Chapter 5
Methods for the Design and Development

Harald Anacker, Michael Dellnitz, Kathrin Flaßkamp, Stefan Groesbrink, Philip Hartmann, Christian Heinzemann, Christian Horenkamp, Bernd Kleinjohann, Lisa Kleinjohann, Sebastian Korf, Martin Krüger, Wolfgang Müller, Sina Ober-Blöbaum, Simon Oberthür, Mario Porrmann, Claudia Priesterjahn, Rafael Radkowski, Christoph Rasche, Jan Rieke, Maik Ringkamp, Katharina Stahl, Dominik Steenken, Jörg Stöcklein, Robert Timmermann, Ansgar Trächtler, Katrin Witting, Tao Xie, and Steffen Ziegert

Abstract. After the domain-spanning conceptual design, engineers from different domains work in parallel and apply their domain-specific methods and modeling languages to design the system. Vital for the successful design, are system optimization methods and the design of the reconfiguration behavior. The former methods enable the parametric adaption of the system's behavior, e.g. an adaption of controller parameters, according to a current selection of the system's objectives. The latter realizes structural adaption of the system's behavior, e.g. the exchange of software or hardware parts. Altogether, this leads to a complex system behavior that is hard to overview. In addition, self-optimizing systems are used in safety-critical environments. Consequently, the system's safety-critical behavior has to undergo a rigorous verification and testing process. Existing design methods do not address all of these challenges together. Indeed, a combination of established design methods for traditional technical systems with novel methods that focus on these challenges is necessary. In this chapter, we will focus on such new methods. We will introduce new system optimization and design methods to develop reconfigurations of the software and the microelectronics. In order to ensure the correctness of safety-critical functionality, we propose new testing methods and formal methods to ensure safety-properties of the software. We show how to apply virtual prototyping to deal with the complexity of self-optimizing systems and perform an early analysis of the overall system. As each domain applies its own modeling languages, the result of these methods are several overlapping models. In order to keep these domain-specific models consistent among all domains, we will introduce a

J. Gausemeier et al. (eds.), *Design Methodology for Intelligent Technical Systems*,
Lecture Notes in Mechanical Engineering,
DOI: 10.1007/978-3-642-45435-6_5,

new semi-automatic model synchronization technique. Each of these design methods are integrated with the reference process for the development of self-optimizing systems.

The principle solution forms the basis of the design and development. Engineers of the involved domains derive their domain-specific models from the it. This is, however, an error-prone and tedious task. Therefore, we will introduce a semi-automatic model transformation techniques (cf. Sect. 5.1) that enables engineers to, e.g. derive an initial controller hierarchy or an initial software architecture. Afterwards, each domain details these models. This may involve changes that have an impact on the other domains. In order to keep the models of all domains consistent, we will propose a model synchronization technique (cf. Sect. 5.1.3).

The system must consider several concurrent objectives in different Application Scenarios.

This requires methods for optimizing the system with respect to these objectives and appropriate adaption methods. System optimization methods origin from the research areas of applied mathematics and artificial intelligence. The methods determine the optimal system behavior or a set of optimal compromises for several concurrent objectives. Practically, this is a formalism to compute optimal controller parameters or optimal configurations of the system structure (cf. Sect. 5.3). Then, it is the task of engineers from the domains mechanical, electrical/electronic, control, and software engineering to specify the corresponding change of the system's behavior, i.e. the reconfiguration of the system.

The system can perform reconfigurations on every system level (cf. Sect. 1.4.3). In particular, this requires new design methods for the application software, the system software, and the hardware modules to specify reconfiguration. Furthermore, reconfiguration is often safety-critical and must fulfill hard real-time constraints. Consider the RailCab's reconfiguration behavior to build a convoy as an example (cf. Sect. 2.1.7): The RailCab must reconfigure the controller behavior to consider the distance to the preceding RailCab if the RailCab joins a convoy as a member. In fact, if this function is not free from design faults or the system cannot execute the reconfiguration within a certain time, a crash may happen. Therefore testing and formal verification methods are crucial to ensure the safety of the system's complex behavior and its real-time properties.

On the level of the application software, software engineers specify the communication behavior and the switching between alternative behavior implementations. We apply a component-based design method called MECHATRONICUML that considers hard-real time constraints for the communication behavior, the reconfiguration of controllers, and the reconfiguration of software components. In MECHATRONICUML, formal verification techniques are applied to ensure safety constraints and the real-time properties of the system.

As a consequence of reconfigurations of the application software, the software's resource and performance demands changes. Usually, the system must reconfigure hardware modules to meet the changed requirements of the application software

again. For instance, a change of the communication behavior may require a change of the physical communication topology or the implementation of a communication protocol on the hardware to meet the performance requirements. Different hardware techniques such as FPGAs or multi-processor platforms are capable to realize these reconfigurations. We will present design methods, architectures, and modeling approaches to design dynamically reconfigurable hardware for different techniques and enable flexible and robust implementation of dynamically reconfigurable hardware (Sect 5.4). In particular, a layered architecture such as PALMERA (Paderborn Layer Model for Embedded Reconfigurable Architectures) can be applied to abstract from the different hardware techniques. Based on PALMERA the design-flow INDRA guides engineers through the different steps towards the realization of information processing systems based on dynamically reconfigurable hardware.

The system software forms the interface between the application software and the dynamic reconfigurable hardware. Hence, the system software must define a common interface to trigger changes of the hardware. Furthermore, it must adapt to changing available resources and changing resource demands while operating under hard real-time constraints. This requires new concepts and design methods for the system software. ORCOS (Organic Reconfigurable Operating System) is a real-time operating system that provides operating system services and an architecture to master these challenges (cf. Sect. 5.5). For instance, the FRM (flexible resource management) allows an overallocation of resources to optimize the resource availability under changing resource demands (cf. Sect. 5.5.2).

The result of the design and development is a complex composed behavior developed by different engineers. This leads to a specification, that is hard to overview. In addition, engineers must ensure the correctness of safety-critical functionality as early as possible during the design. One solution to cope with the complexity is to build and test a virtual prototype. Virtual prototyping enables engineers to perform experiments during early development phases. It requires models of the system that are often created by several tools. We will introduce a concept of a virtual environment and methods to extend the environment and integrate models of the domains involved.

This chapter is structured as follows: First, we will describe the model transformation techniques to derive the domain-specific models from the principle solution and the model synchronization technique to keep the domain-specific models consistent (cf. Sect. 5.1). In Sect. 5.2, we will introduce the design of the communication software and reconfiguration behavior with MECHATRONICUML. Novel system optimization approaches that origin from mathematics and artificial intelligence follow in Sect. 5.3. We will describe technologies and design methods for dynamic reconfigurable hardware in Sect. 5.4. In Sect 5.5, we will focus on the system software and introduce the self-optimizing real-time operating system ORCOS. Finally, we will introduce virtual prototyping and advanced testing methods in Sect. 5.6.

5.1 Automatic Model Transformation and Synchronization

Jan Rieke

In the domain-spanning conceptual design, experts from all domains have elaborated the principle solution. This principle solution covers all domain-spanning relevant information, i.e. all interfaces and overlaps between different domains are described in this model. Thus, the principle solution can serve as a starting point for the domain-specific design and development.

In this section, we will show how *model transformation techniques* can be applied to *automatically derive initial domain-specific models* that are consistent with the principle solution and all other domain-specific models. Basically, these initial models contain skeletons that are filled by the domain engineers in the design and development phase. We will explain a model transformation that generates software engineering models from a principle solution in Sect. 5.1.2.1. In Sect. 5.1.2.2, we will also show how initial control engineering models can be generated.

Ideally, the principle solution covers all domain-spanning aspects. Thus, there should be no need for further domain-spanning coordination. However, in practice, the principle solution rarely captures every domain-spanning concern. Additionally, changes to the overall system design may become necessary later on, e.g. due to changing requirements. Therefore, cross-domain changes may become necessary during the domain-specific design and development. Sect. 5.1.3 explains in detail how *model synchronization techniques* can be applied in such a scenario.

Before describing the model transformation and synchronization technique in detail, let us have a closer look at an example.

5.1.1 Example Scenario

As a running example, let us consider the RailCab system (cf. Sect. 2.1). When driving in a convoy, all RailCabs (except for the convoy leader) control their velocity based on the distance to the RailCab traveling in front of them. To measure the distance, a distance sensor is mounted in the front of each RailCab. Figure 5.1 shows how the different models evolve in the exemplary scenario described below.

To illustrate transformations and synchronizations that may become necessary throughout the development, assume the following exemplary process. After the system engineers design the principle solution, we apply model transformations to the different domain-specific models (step 1). The control engineers then start implementing the controllers (step 2). In particular, they elaborate on the velocity control strategies for driving in a convoy as a follower based upon the distance measured by the distance sensor. This is a domain-specific refinement that has no influence on the models of other disciplines.

Modern mechatronic systems incorporate self-healing to repair the system in case of failure. As described in Sect. 5.2.7, the software engineers perform an analysis of the self-healing operations in order to determine whether they reduce the probability of hazards successfully. In our example, the distance sensor could fail or send bad

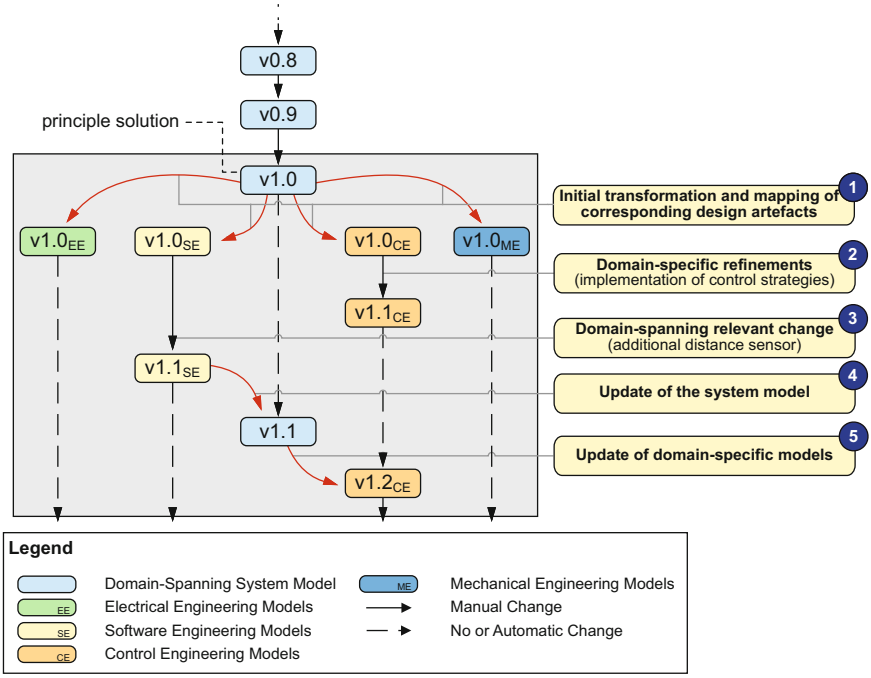

Fig. 5.1 Evolution of different models during the development process

data. It may turn out that even with self-healing the hazard probability can not be reduced to an acceptable level: The hazard of two RailCabs colliding during convoy mode due to a failing distance sensor exceeds the acceptable hazard probability of the system. Thus, software engineers should propose adding redundancy by adding a second distance sensor. They add a new sensor measurement component to their software model (step 3). This is a domain-spanning relevant change, i.e. it affects the domain-spanning system model as well as several domain-specific models. In particular, the velocity control strategy must be modified.

Thus, we use **model synchronization techniques** to propagate the change to the system model (step 4). In contrast to model transformation, which translates complete models, the idea of model synchronization is to modify only the model elements that have been changed after the initial model transformation. Thus, model synchronization is also called an *incremental update*. Version 1.1 of the system model now contains a second distance sensor. To allow all engineers to react to change, it is propagated further to all affected domain-specific models. For instance, the control engineering model is updated, again using model synchronization techniques (step 5).

The control engineers can now modify their control strategy to use both sensor data as input. In step 5, it is crucial that the domain-specific model is updated in a

way, so that all refinements and implementations that have been added to it in the meantime (see step 2) are retained.

Next, we will describe in detail a) how to derive initial domain-specific models, and b) how to propagate changes to the system model and further on to domain-specific ones.

5.1.2 Deriving Initial Domain-Specific Models from the System Model

In this section, we will present two example transformations from the system model to domain-specific models. We will use the principle solution as the input to these transformations to create models for the domain-specific design and development. However, these transformations can also already be used during the conceptual design to generate domain-specific models, for instance, for early simulation and verification.

First, we will describe how the active structure can be used to derive initial software component models in MECHATRONICUML[9], and how the Behavior–States model is transformed to an initial software statechart. Next, we will show how control engineering models (MATLAB/Simulink and Stateflow) can be derived.

5.1.2.1 Transformation from CONSENS to Software Engineering Models

Figure 5.2 shows the basic principles of the transformation from CONSENS to MECHATRONICUML software models. In the active structure, you can see small colored annotations above the system elements. These so-called *relevance annotations* define which element is relevant to which domain-specific model. For instance, "SE" and "CE" denotes software and control engineering, respectively.

The central idea of mapping is that every system element that has a software engineering relevance annotation (i.e. it fulfills software functions) should be represented by a software component in the MECHATRONICUML model. The information flows between system elements are mapped to ports and connectors in MECHATRONICUML.

Generally, we distinguish between continuous and discrete components. *Continuous components* are typically controllers that continuously process input data from sensors to compute outputs for actuators Typically, control engineers implement them. However, MECHATRONICUML allows integrating them as continuous components. **Continuous components** are black-box components, i.e. no actual behavior is attached to continuous components in MECHATRONICUML. In this way, they define the interface to control engineering in a MECHATRONICUML software model.

In contrast, the behavior of *discrete components* is implemented using MECHATRONICUML (cf. Sect. 5.2). **Discrete components** communicate with each other via discrete ports using asynchronous, message-based communication defined in

[9] See Sect. 5.2 for a detailed explanation of MECHATRONICUML.

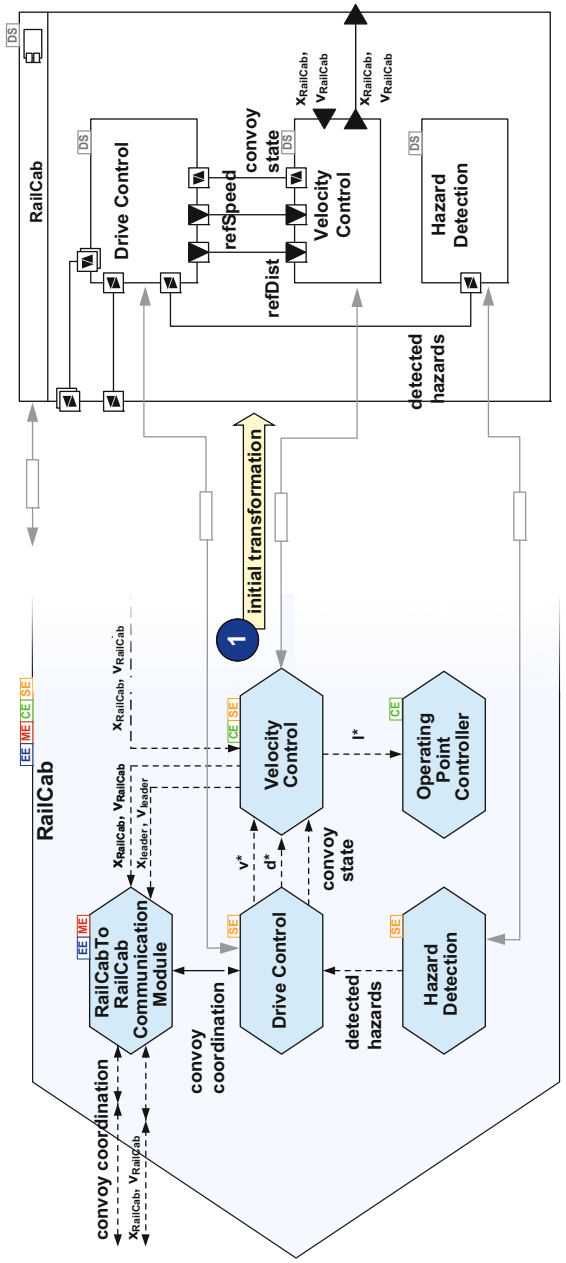

Fig. 5.2 Initial transformation from the active structure to a software component diagram (adapted from [70])

real-time statecharts. Discrete components can also send or receive signals to or from continuous components using hybrid ports. In Fig. 5.2, the components *Drive Control* and *Velocity Control* have both discrete and continuous ports.

The transformation creates discrete components in MECHATRONICUML for every system element that is relevant for software engineering. We use a technique called *Triple Graph Grammars* (TGG) for defining the model transformations to the different domain-specific models. TGGs are a graph-based, declarative technique to define mappings between two models, invented by Schürr (1994) [188].

Figure 5.3 shows a TGG rule that is part of the TGG rule set that implements this mapping.

A TGG rule describes which model elements in one or more source models relate to which model elements in one or more target models. In Fig. 5.3, the two source models are the two left columns, *AS Annotation* and *active structure*.[10] The target models are located in the right columns, *Component Diagram* and *UML Annotation*. In the middle column, the so-called *correspondence model* is described, which is a kind of trace model, storing relations between the models. It is used to identify corresponding model parts when incrementally updating models. The green parts of the rule, additionally marked with "++"s, is the actual mapping, stating that a *SystemElementInstance* that has a *Relevance* annotation must be mapped to *Property*, also with a *Relevance* annotation. The b/w part of the rule is the *context*, defining in which situations the mapping must be valid.

We use our TGG Interpreter Tool Suite [203] to define and automatically execute these TGG rules. Given a domain-spanning system model and a TGG rule set, the TGG Interpreter can automatically create the corresponding domain-specific models. We refer to Greenyer and Kindler (2010) [78] for further details on TGGs. Gausemeier et al. (2009) [70] describe the principles of the transformation from the active structure to software models in detail.

5.1.2.2 Transformation from CONSENS to Control Engineering Models

Figure 5.4 shows the basic principles of the transformation from CONSENS to MATLAB/Simulink control engineering models.

Generally, every system element that is relevant to software or control engineering is mapped to a Simulink block. The system elements relevant only to software engineering, however, are just placeholders. When the software engineers finish the actual implementation, this implementation is inserted. This is because we use a MATLAB/Simulink model at the end of the development process as a combined software/control engineering model from which code is generated. Thus, all artifacts of the software engineering domain are integrated into this MATLAB/Simulink model.

To allow the integration of discrete software components that use asynchronous, message-based communication and reconfiguration, we use a message bus

[10] Due to technical reasons (e.g. to allow easy extensibility), annotations are stored in two separate models. Thus, they are located in the separate columns *AS Annotation* and *UML Annotation* in Fig. 5.2.

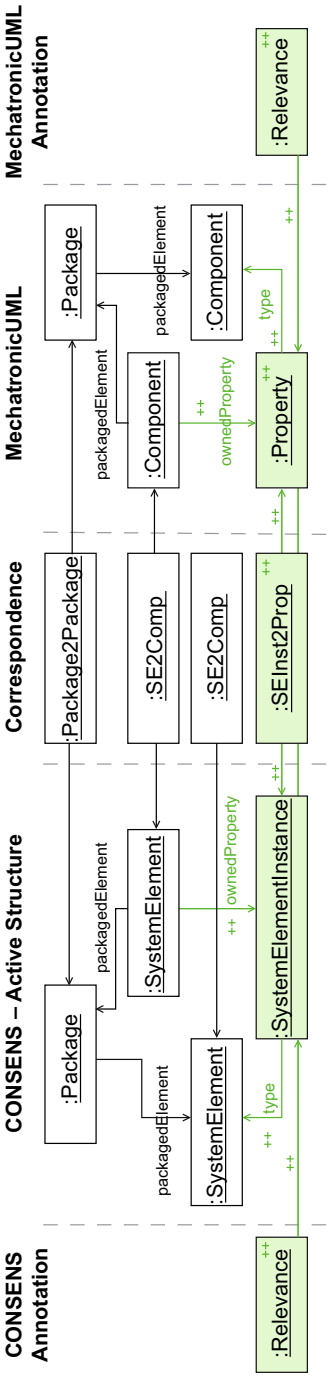

Fig. 5.3 TGG rule for mapping system elements to software components (adapted from [70])

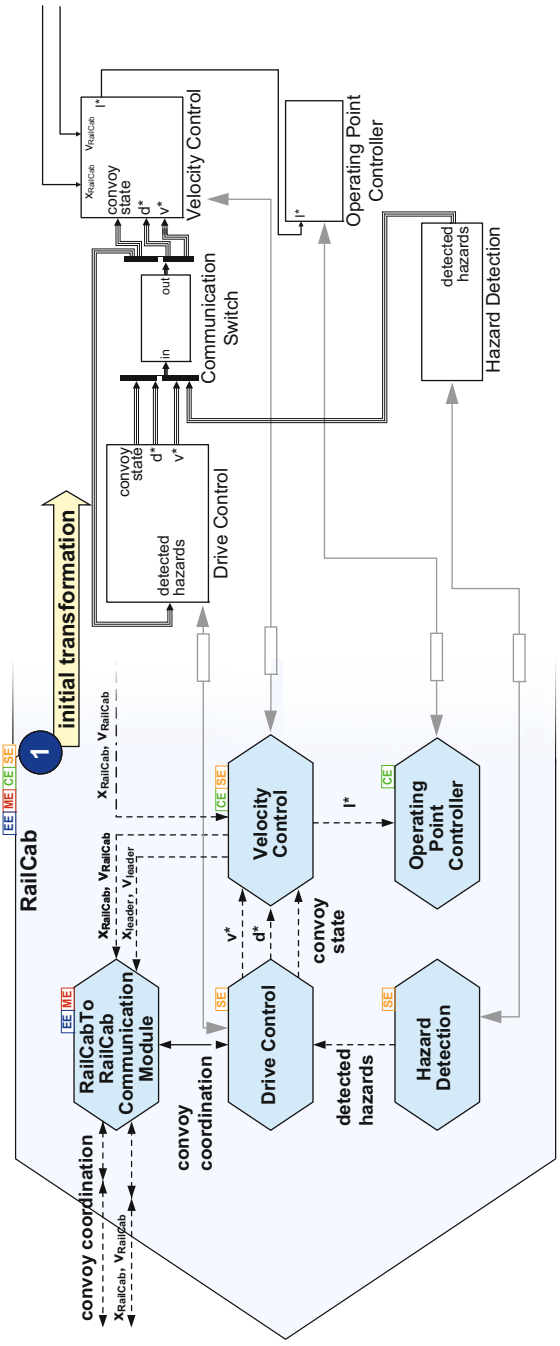

Fig. 5.4 Initial transformation from the active structure to a MATLAB/Simulink control engineering model

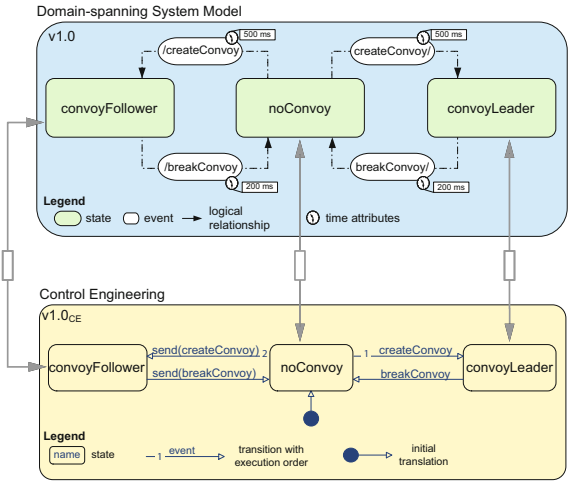

Fig. 5.5 Initial transformation from Behavior–States to a MATLAB/Stateflow model

approach. The communication between two discrete components is implemented using a *Communication Switch*. This switch connects every component and is responsible for forwarding sent messages to the correct recipient. This is necessary to allow changing communication structures as required when reconfiguring a system. Signal-based information flow, like the I^* value signal from *Velocity Control* to *Operating Point Controller*, is mapped to connected outputs and inputs of the respective blocks.

Furthermore, we use behavioral models of the principle solution to generate MATLAB/Stateflow control engineering models. Figure 5.5 shows such a transformation of behavioral models.

Rieke et al. (2012) [180] describe the principles of the transformation of state-based models. Heinzemann et al. (2012) [93] give technical details on the generation of MATLAB/Stateflow and Simulink models.

5.1.3 *Synchronizing Models during the Domain-Specific Refinement Phase*

Although most domain-spanning relevant information should already be present at the end of the conceptual design, changes to the system in development may become necessary during the domain-specific design and development. For instance, requirements may still change during later phases, or it may turn out that some aspect of the system must be implemented in another way. This easily leads to changes that affect both the domain-spanning system model and several domain-specific models. Furthermore, engineers may have already generated early domain-specific models during conceptual design, to allow early checks and simulations of different concepts and ideas. It is reasonable to keep these early models and to reuse and refine them during the design and development.

This requires keeping the development models consistent during all phases of the development. Manually checking and restoring the consistency of all models is a time-consuming and error-prone task. Therefore, we apply similar methods as with the derivation of initial models (described in the previous section) to synchronize models during the development.

First, we will describe how the system model is updated when changes in a domain-specific model occur. Next, we will show how domain-specific models can be updated with respect to these system model changes.

5.1.3.1 Updating the System Model

As described in Sect. 5.1.1, an extra distance measurement component is added to the software model (step 3 in Fig. 5.1). Our model transformation approach forwards this change to the system model (step 4 in Fig. 5.1). Figure 5.6 shows the result of this step.

We again use TGGs to perform such model synchronization operations. TGG rules can be applied bidirectionally, i.e. transformation and synchronization operations can be performed both from the system model to the software model and vice versa. Here, we apply the TGG rules reverse, propagating the change from the software model to the system model. The added *Distance Measurement* system element is shown on the left side of Fig. 5.6.

We do not want to simply run the transformation again in backwards direction, as this would completely re-create one model. Thus, the core idea is to only update modified model parts and leave everything else untouched. For every model element, we check whether mapping of this element is still valid. To do so, our approach uses the existing trace information that is stored inside the correspondence graph. Using this correspondence graph, it can identify corresponding model elements in the two models and then check the consistency of these model elements by testing whether the TGG rule that was applied there still holds. The approach only modifies a model element if a rule does not hold any more relevance. Such an approach is called *incremental model transformation* or *model synchronization*.

We have developed a new, improved model synchronization algorithm that is tailored for the use in mechatronic system design. More specifically, it prevents the loss of information in models during the synchronization process. This is especially required when synchronizing development models of mechatronic system, as these models have different abstraction levels and/or different views: The system model is usually more abstract than the domain-specific models that contain concrete implementation details. Thus, the domain-specific models may contain information that is not part of the system model. For instance, the Stateflow model shown in the lower part of Fig. 5.5 is later refined such that it contains details of controller reconfigurations that happen when switching convoy states. Thus, the Stateflow model now contains information that is not present in the abstract system model. When the system model is changed, this change may affect parts of the Stateflow model that has been refined. Our synchronization algorithm avoids affecting these

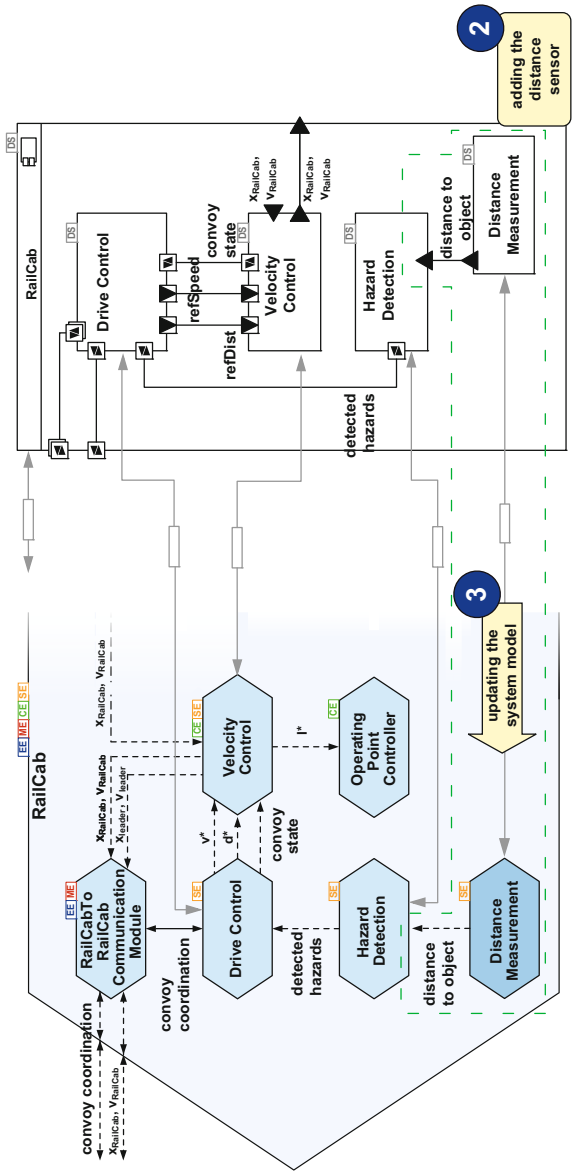

Fig. 5.6 Updating the active structure using the altered software component diagram (from [70])

refinements when updating this domain-specific model. Rieke et al. (2012) describe such a change scenario in detail [180]. For details on the improved model synchronization algorithm, see Greenyer et al. (2011) [79].

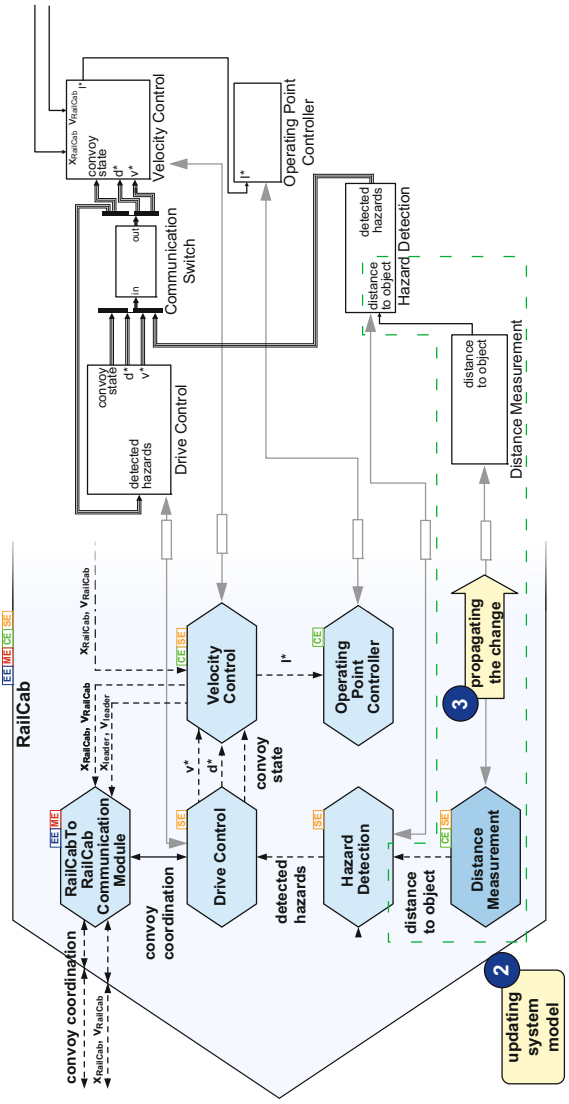

Fig. 5.7 Updating the MATLAB/Simulink control engineering model using the updated active structure diagram

5.1.3.2 Updating Control Engineering Models

After updating the system model, these changes must be propagated to other affected domain-specific models (step 5 in Fig. 5.1). Figure 5.7 shows how the added *Distance Measurement* system element can also be added to the control engineering model.

This is again achieved by rerunning the transformation incrementally, leaving the unaffected parts untouched and only adding a new block with its respective inputs, outputs and lines.

When changes to a model occur, we are able to update other affected models automatically in most cases, using these improved model transformation and synchronization techniques. However, there might be cases where user decisions are indispensable, for instance when there are different possibilities to propagate a specific change. Thus, it is reasonable to combine this technique with means for user interaction [79].

5.2 Software Design

Christian Heinzemann, Claudia Priesterjahn, Dominik Steenken, and Steffen Ziegert

Self-optimizing mechatronic systems execute a great amount of software to coordinate the operations of the system. In the following, we will refer to that software as the *discrete* software of the system as opposed to the controller software. The RailCab demonstrator for example (cf. Sect. 2.1) needs discrete software to manage the necessary communication for getting admission to drive onto a track section and, especially, for driving in convoy mode. In convoy mode, RailCabs need to execute complex coordination behavior for maintaining the convoy when the convoy consists of more than two RailCabs. Since RailCabs can join or leave a convoy during a journey, a flexible structure for the specification of the coordination is needed. The required small distances between RailCabs in a convoy imply real-time coordination between the speed control units of the RailCabs. This is safety-critical and requires the software engineer to address a number of constraints when designing the RailCabs' control software.

In the design and development, the software engineers apply the MECHATRONICUML method [53, 75] for designing the discrete software of mechatronic systems, especially of self-optimizing mechatronic systems (cf. Sect. 3.3.3). **MECHATRONICUML** enables a component-based specification of the discrete software with a special focus on specifying the communication and reconfiguration behavior of a self-optimizing mechatronic system. The development process for developing with MECHATRONICUML in the course of the design and development is shown in Fig. 3.11 on Page 84. We illustrate the development with MECHATRONICUML by providing an overview of the general concepts of MECHATRONICUML [1, 43, 75] and recent extensions [53, 54, 91, 196] in the course of this section. The complete, technical language specification of MECHATRONICUML can be found in [18].

The software engineers start the development with MECHATRONICUML by deriving a component model for the discrete software as discussed in Sect. 5.2.1. In the next step, the communication requirements need to be decomposed based on the components of the component model as described in Sect. 5.2.2. The communication protocols that define the message-based communication of the components are specified formally by using real-time coordination patterns and verified with our design-time verification procedure as explained in Sect. 5.2.3. Afterwards, the component's discrete communication behavior is specified as described in Sect. 5.2.4.

In Sect. 5.2.5 we will outline how the complete hybrid system is simulated. When the simulation is successful, the deployment of software components to hardware is specified as explained in Sect. 5.2.6. Finally, we will outline an analysis of self-healing operations in Sect. 5.2.7 and the code generation in Sect. 5.2.8.

5.2.1 Component Model

The software development with MECHATRONICUML starts by deriving an initial component model for the system, because MECHATRONICUML follows the component-based approach [198] for developing software. Each component encapsulates part of the system functionality and the components only interact via well-defined interfaces, called ports. An initial component model is derived from the Active Structure by using the transformation presented in Sect. 5.1.2.1. Since the Active Structure only contains components that affect more than one discipline, it might be necessary to refine the component model by splitting the behavior of a component into several subcomponents. That reduces the complexity of the single components which, in turn, enables the reuse of existing components and makes their verification more efficient.

Fig. 5.8 shows the *DriveControl* component of the RailCab that has been derived from the system element *DriveControl* as shown in Fig. 5.2. The *DriveControl* component encapsulates the software controlling the driving operations of the RailCab. In our example, a RailCab will either be a coordinator or a member of a convoy, but not both at the same time. Therefore, the developer may decide to split the behavior of the RailCab component into subcomponents. The two components *ConvoyCoordination* and *MemberControl* encapsulate the behavior of being coordinator and of being member respectively. In addition to these components, each RailCab requires a component *SpeedControl* which defines the speed for the RailCab which serves as the reference speed for the controller. If the RailCab is a convoy member, the reference speed and an additional reference distance to the preceding RailCab in the convoy are received by the *MemberControl* and propagated by *SpeedControl* to the controller.

In self-optimizing mechatronic systems, the components interact by means of message passing via their ports. In MECHATRONICUML, the behavior that defines an interaction between two components is specified by so-called real-time coordination patterns (cf. Sect. 5.2.3). In Fig. 5.8, the *DriveControl* interacts with other components using the ports *coordinator*, *member*, *hazardReceiver*, *convoyState*, *refSpeed*, and *refDist*. The former four ports are discrete ports that execute a state-based communication protocol specified by a real-time coordination pattern (cf. Sect. 5.2.3). The *refSpeed* and *refDist* ports are so-called hybrid ports which are used for providing a value, in this example the reference speed and reference distance for the RailCab, to a controller.

The behavior of components and ports is defined using a state-based approach called **real-time statecharts** (RTSC). RTSCs are a combination of UML

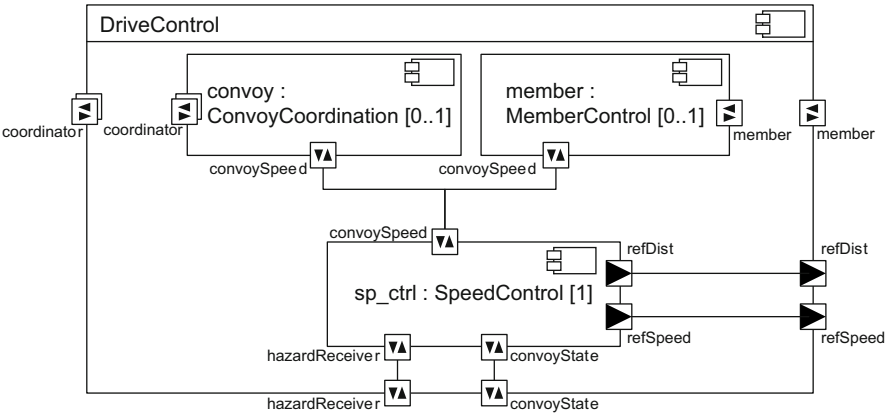

Fig. 5.8 *DriveControl* component of the RailCab

statemachines and timed automata [8]. We will provide more information on RTSCs using an example in Sect. 5.2.3.

The MECHATRONICUML component model distinguishes between components and component instances. A component instance is the occurrence of a component in a system. Component instances are connected via their ports for specifying a concrete system architecture, called component instance configuration.

Fig. 5.9 shows a component instance configuration that consists of three instances of the component *RailCab* (cf. Fig. 5.2). The *RailCabs* drive in a convoy because they execute the real-time coordination pattern *ConvoyCoordination* which we will introduce in detail in Sect. 5.2.3.

5.2.2 Decompose Communication Requirements

In a self-optimizing mechatronic system, the single components often interact and exchange different kinds of data. In the example in Fig. 5.9, RailCabs interact with each other for two reasons. First, they communicate to coordinate the convoy drive and, second, a RailCab needs to transmit its current position to adjacent RailCabs in the convoy for controlling the distance. The communication protocols defining the necessary message exchange are specified by real-time coordination patterns of MECHATRONICUML. A developer should specify one real-time coordination pattern for each reason for interaction to achieve separation of concerns. This in return will reduce the complexity of the single real-time coordination patterns, allow a more efficient verification, and enable their reuse in different systems.

The requirements for the real-time coordination patterns are specified by means of **Modal Sequence Diagrams** (MSDs) as described in Sect. 4.3. The MSD specification, however, does not distinguish the different communication protocols. Therefore, the developer needs to decompose the MSDs according to the communication protocols that are needed in the system. For the example in Fig. 5.9, we obtain

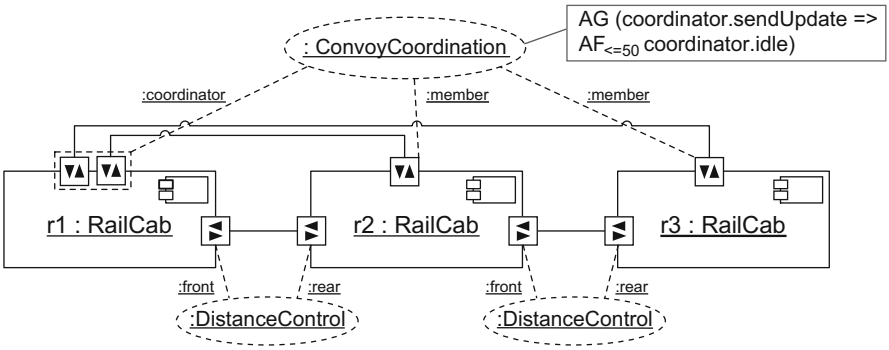

Fig. 5.9 Component instance configuration of a convoy with three RailCabs

one set of MSDs for the *ConvoyCoordination* and one set of MSDs for the *DistanceControl*. Then, the developer needs to define a real-time coordination pattern as described in Sect. 5.2.3 for each of the communication protocols. These real-time coordination patterns are then associated with the ports and connectors of the component model as shown in Fig. 5.9.

In addition, the developer may split components into several subcomponents as illustrated in the *DriveControl* component in Fig. 5.8. In this case, the interactions need to be associated with subcomponents that will implement the interaction. This step might require a further derivation of MSDs that define the requirements for the communication within a component. In the *DriveControl* component, the developer needs to specify MSDs for the interaction of *SpeedControl* with *ConvoyCoordination* and *MemberControl*.

5.2.3 Real-Time Coordination Patterns

The communication behavior of the components is specified formally by using real-time coordination patterns. The developer needs to specify a real-time coordination pattern for each connector between components in the component model. In MECHATRONICUML, real-time coordination patterns are specified independent of a concrete component to allow reusing them in different systems. Thus, the developer either needs to specify a new real-time coordination pattern based on the communication requirements as described in Sect. 5.2.3.1 or he may reuse an existing real-time coordination pattern. The real-time coordination pattern is refined to the specific components as part of process step "Specify Discrete Behavior" (cf. Sect. 5.2.4).

The communication behavior is safety-critical. In our example, errors in the communication between convoy coordinator and convoy members may lead to an accident. If a RailCab still operates in convoy mode while the convoy coordinator assumes that it has left the convoy, a crash may occur if the convoy brakes because

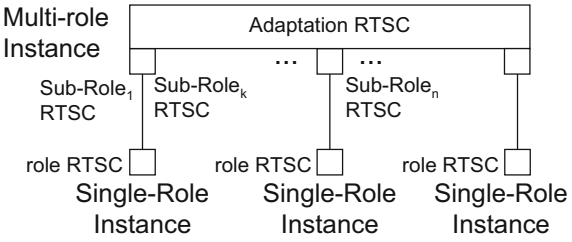

Fig. 5.10 Instance of a real-time coordination pattern with a multirole

the RailCab will not be notified. We can prove the correctness of the communication behavior by using our design-time verification procedure outlined in Sect. 5.2.3.2.

5.2.3.1 Specification of Real-Time Coordination Patterns

A real-time coordination pattern defines the required communication between two communications partners independent of a concrete component implementation. We call the communication partners *roles*. In this section, we focus on 1:*n* communication where one role communicates with *n* other roles all executing the same behavior [53]. In a RailCab convoy (cf. Fig. 5.9), one RailCab serves as a coordinator and needs to communicate with the *n* other members of the convoy. The coordinator is required, e.g. for defining a reference speed for the whole convoy and to coordinate acceleration and braking maneuvers.

RailCab *r1* is the coordinator of the convoy, while *r2* and *r3* are members. Therefore, *r1* executes an instance of the *coordinator* role of the *ConvoyCoordination* real-time coordination pattern. RailCabs *r2* and *r3* execute an instance of the *member* role. The instances of the *DistanceControl* real-time coordination pattern are used for controlling the distance between two successive RailCabs in a convoy.

Since the *coordinator* role instance communicates with *n member* role instances, we call it a *multirole*. The *member* role instance, which communicate with only one *coordinator* role instance is called a *singlerole*. Fig. 5.10 shows the general structure of an instance of a real-time coordination pattern with a multirole instance.

The multirole instance consists of an adaptation real-time statechart and *n subrole* real-time statecharts. Each of the subrole instances manages the communication with exactly one singlerole instance. The adaptation real-time statechart is responsible for creating and deleting subrole instances, e.g. if RailCabs join or leave a convoy. In addition, the adaptation real-time statechart is used to coordinate the subrole real-time statecharts, e.g. to trigger that they send data to the member RailCabs in a defined order.

Fig. 5.11 shows the real-time statechart that defines the behavior of the singlerole *member*. The real-time statechart starts its execution in the initial state *waitUpdate*. It waits for 500 time units for an *update* message to arrive. Messages are sent asynchronously between different roles, i.e. the receiver stores the message in a buffer and may process it at a later point in time. If the message arrives in time, the real-time statechart switches to *sendAck* thereby resetting the clock *c* to 0. If the message

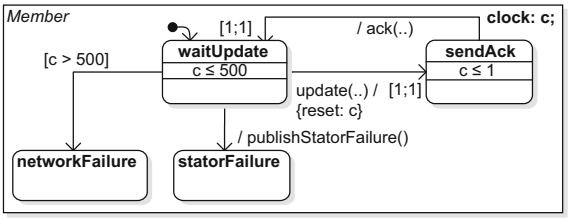

Fig. 5.11 Real-time statechart of a convoy member

does not arrive in time, it switches to state *networkFailure*. The state *sendAck* is left after 1 time unit by sending a message *ack* and switching to *waitUpdate*.

Fig. 5.12 shows the real-time statechart that defines the behavior of the multirole *coordinator*. The real-time statechart of a multirole always consists of one state that contains two parallel regions, which is *Coordinator_Main* in the example. One region contains the adaptation real-time statechart while the other contains the subrole real-time statechart that is executed by all subrole instances. At run-time, we obtain one real-time statechart instance of the subrole real-time statechart for each subrole instance.

The *coordinator* subrole real-time statechart in the lower region of *Coordinator_Main* is the pendant to the *member* real-time statechart of Fig. 5.11. It is initially in state *idle*. The transition from *idle* to *sendUpdate* is triggered by a synchronous internal event *next* which is parameterized by an integer. Synchronous events cause

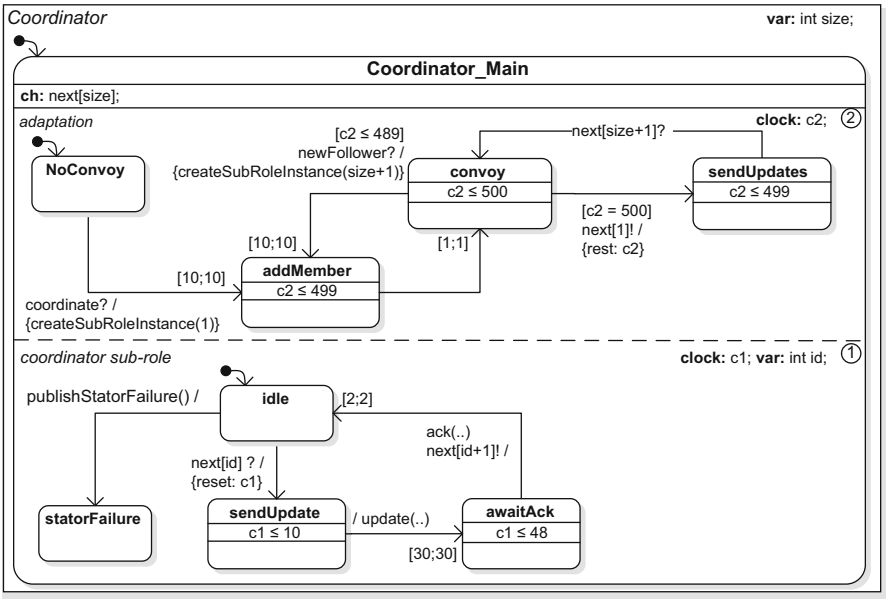

Fig. 5.12 Real-time statechart of the convoy coordinator

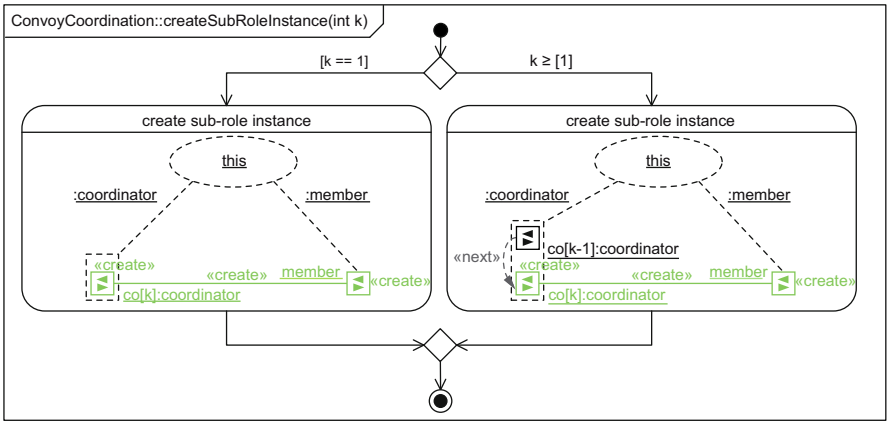

Fig. 5.13 Component story diagram modeling the creation of a subrole instance

the sender transition (event suffixed by !) and the receiver transition (event suffixed by ?) to fire simultaneously. Additionally, sender and receiver must provide and expect the same integer parameter. In the state *sendUpdate*, the real-time statechart may spend up to 10 time units before it sends the *update* message and switches to the state *awaitAck*. Executing this transition takes a minimum and a maximum of 30 time units which is indicated by the deadline in square brackets. The transition from *awaitAck* to *idle* is triggered by the receipt of the *ack* message from the *member* role. It triggers the next subrole using the synchronous event *next*, incrementing the expected integer by 1.

The adaptation real-time statechart in the upper region of *Coordinator_Main* starts in state *noConvoy*. If the RailCab is chosen to coordinate the convoy, it is triggered by the synchronous event *coordinate* and switches to *addMember*. The side effect *createSubRoleInstance* at the transition triggers the component story diagram of Fig. 5.13 that creates a new subrole instance in the *coordinator* multirole instance. Then, the real-time statechart switches to the state *convoy*. In the state convoy, the real-time statechart triggers the first subrole instance every 500 time units using the synchronous event *next*. The transition from *sendUpdate* back to *convoy* synchronizes with the last subrole instance after it has successfully received the *ack* from the *member* role. Back in the state convoy, the real-time statechart can only be triggered by the synchronous event *newFollower* and switch to *addMember*. Again, the side effect *createSubRoleInstance* of the transition executes the component story diagram of Fig. 5.13 for creating a new subrole instance.

Component story diagrams [200] are a special kind of graph transformation rules [184] that use the concrete syntax of MECHATRONICUML. We use component story diagrams for specifying run-time reconfiguration operations, i.e. the creation and deletion of component instances and connections. The component story diagram of Fig. 5.13 instantiates a new connection to a new member that wants to join the convoy at position *k*. In the component story diagram, we distinguish

between creating the first connection and creating further connections. In the activity node on the left, we create the first connection between coordinator and a member in the real-time coordination pattern. The *this*-variable represents the instance of the *ConvoyCoordination* real-time coordination pattern which called the component story diagram from the adaptation real-time statechart of its multirole instance. In addition, the multirole instance, modeled by the dashed rectangle, is bound. Then, the parts of the rule annotated with «create» are created and the connection is established. In the activity node on the right, the subrole instance with index $k-1$ is bound additionally and the new subrole instances is created as a successor to this subrole instance.

5.2.3.2 Design-Time Verification of Real-Time Coordination Patterns

The correctness of software for self-optimizing mechatronic systems is often safety-critical, especially if the software influences the physical movement of the system. That requires the software to meet high quality standards to ensure its safe operation. Traditional testing-based development approaches are not able to guarantee functional correctness. Design-time verification, however, is a method to give a mathematical proof that a software is functionally correct with respect to a formal specification [13]. In this section, we will illustrate how design-time verification can be used to ensure that the communication within a self-optimizing mechatronic system modeled by real-time coordination patterns is safe.

The real-time coordination patterns used in self-optimizing mechatronic systems are often subject to run-time reconfiguration. An example is given by the *Convoy-Coordination* real-time coordination pattern introduced in Sect. 5.2.3. In such real-time coordination patterns, the behavior is defined by a syntactical combination of real-time statecharts and component story diagrams. Consequently, a verification procedure needs to take both into account.

Existing approaches and corresponding tools for design-time verification do not provide sufficient support for self-optimizing mechatronic systems that adhere to real-time constraints and use run-time reconfiguration. Graph-based tools like GROOVE are very effective for verifying untimed graph transformation systems (GTS) [113], but are still limited, especially with respect to verification of timing properties. Timed model checkers such as Kronos [31] or UPPAAL [21], which support the verification of real-time statecharts, provide no means for specifying dynamic object creation and deletion.

Our method for design-time verification combines the strengths of both approaches for verifying real-time coordination patterns with run-time reconfiguration. It is executed at design-time by the software engineer while creating the real-time coordination patterns as opposed to run-time verification which is performed while the mechatronic system is running [69, D.o.S.O.M.S. Sect. 3.2.14] .

Fig. 5.14 provides an overview of the single steps of our verification procedure. It requires two inputs: a real-time coordination pattern and a set of requirements that need to be verified. The textual requirements informally state the properties that the behavior modeled in MechatronicUML needs to fulfill. Then, we perform two

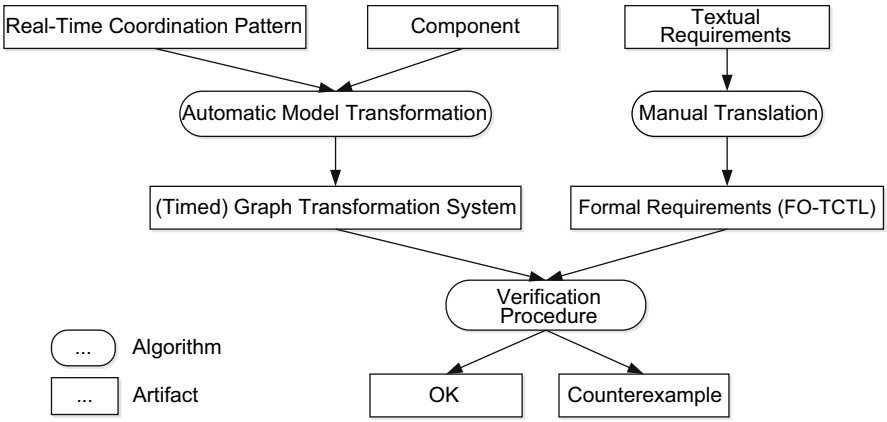

Fig. 5.14 Overview of the design-time verification procedure

transformation steps that transform the inputs such that they can be processed by the verification procedures. The transformation of the real-time coordination pattern to a (timed) graph transformation is completely automatized. The transformation of textual requirements to formal requirements is a manual task. After explaining them in the following subsections, we will describe our verification procedure.

The result of applying this method to a MECHATRONICUML model is either that the model fulfills the formalized requirements or a counterexample. A counterexample is an execution of the system that leads to a state that violates the specified requirement. The counterexample is intended to support an engineer in locating and correcting the cause of an error in the model. After correcting the error, this method needs to be applied again until no more errors are found in the model. Then, the model is correct with respect to the formal requirements that have been verified. The verified model is the input for a code generator that generates the source code for the system.

From Real-Time Statecharts to Graph Transformation Systems

Design-time verification of MECHATRONICUML models needs to capture the behavior of the real-time statecharts as well as their run-time reconfiguration operations in terms of component story diagrams. This is because both strongly influence each other. As an example, consider the real-time coordination pattern *ConvoyCoordination* shown in Fig. 5.9. The *coordinator* needs one subrole instance for each convoy member. Therefore, the multirole real-time statechart calls a reconfiguration operation as a side effect (cf. Fig. 5.12). The reconfiguration operation, in turn, creates a new subrole instance including an instance of the subrole real-time statechart. The execution of the new real-time statechart instance contributes to the behavior of the real-time coordination pattern instance and, thus, needs to be analyzed by the verification procedure. To cope with this strong interconnection between timed

state-based behavior and reconfiguration, we use **timed graph transformation systems** (timed GTS), as shown in [53].

At this point, graph transformations [184] play a double role in our approach [53]. While they are used to model reconfiguration operations formally in terms of component story diagrams, they are also used as a meta-language to define the semantics of MECHATRONICUML. Such formally defined semantics is the basis for an automated verification procedure. A key extension necessary for self-optimizing mechatronic systems is the annotation of time, which is needed to capture the semantics of real-time statecharts. Therefore, we use timed GTS as the basis for our verification procedure. The use of timed GTS at this level, however, is hidden from the modeler who gives a MECHATRONICUML specification to the model checker which performs the translation automatically.

A timed GTS [53] consists of a start graph, a type graph, and a three different types of rules, namely timed graph transformation rules (timed GT rules), clock instance rules, and invariant rules. The start graph defines the starting point for the execution of the timed GTS and the type graph defines the types of nodes and edges for all graph generated by the timed GTS. Timed GT rules change a timed graph, but may neither add nor remove clock instances. Clock instance rules are used to add all clock instances that are possibly required for the application of a timed GT rule. Invariant rules forbid the existence of a subgraph of a timed graph after a certain time bound. For a formal definition of timed GTS, we refer to our technical report [196].

The translation of a MECHATRONICUML model into a timed GTS needs to encode the behavior of the real-time statecharts by timed graph transformation rules. We use objects representing the instances of the real-time statecharts including their states. Transitions cause a change of the active state of a real-time statechart. Consequently, we create a timed GT rule for each transition of a real-time statechart. State invariants forbid that a state is active beyond a specified point in time. They are translated to invariant rules. The clocks that are used by the real-time statecharts are created using clock instance rules. We refer to [53] and our technical report [92] for more information on the translation.

From Textual Requirements to Formal Requirements

As shown in Fig. 5.14, a second translation is required for translating the textual requirements into formal requirements. Informal requirements in natural language are not suitable for being processed by an automatic verification procedure. An automatic verification procedure requires a formal specification of the requirements. Such translation needs to be carried out manually by an engineer. For timed automata, TCTL [6] has been introduced as a formal language for expressing such requirements.

In our *ConvoyCoordination* example, operating in convoy mode requires one RailCab to operate as a coordinator and periodically send reference data updates to all other convoy members. There, we need to ensure, e.g. that after the coordinator sends an update to a member, the coordinator must receive an acknowledgement within 50 ms. This constraint is formalized by the TCTL property

$$\mathbf{AG}(\text{coordinator.sendUpdate} \Rightarrow \mathbf{AF}_{<=50}\text{coordinator.idle})$$

shown in Fig. 5.9. This property obviously needs to be valid *for all* subrole instances of the *coordinator* and, in particular, must be valid throughout the reconfiguration.

In our verification procedure, we use FO-TCTL which is an extension of TCTL [6] by constructs of first-order logic. It enables specification of properties on graph structures in a much more user-friendly way compared to plain TCTL. In particular, it supports specifying a property that needs to be valid *for all* subrole instances of a real-time coordination pattern. To achieve this, we introduce variables that range over the nodes of a graph, constants that represent particular nodes that are known at design-time, predicates representing types of nodes and edges, and quantifiers. Variables allow the formulation of properties concerning nodes without knowing which particular nodes exist during run-time. Expressing the same property using the normal TCTL requires knowledge of all nodes that may exist during the execution of the system.

Verification Procedures

We defined two verification procedures for verifying properties specified in FO-TCTL based on a timed GTS that we will introduce in the following subsections. The first verification procedure, called FO-TCTL model checking, uses a state-exploration technique that enumerates the run-time states of the timed GTS, thereby considering the timing conditions of the timed GTS. Consequently, it supports timed GTS with a finite number of run-time states. Our second verification procedure applies a shape analysis technique. It supports timed GTS with an infinite number of run-time states, but it does not consider the timing conditions.

Verification Procedure 1 – FO-TCTL Model Checking: Our FO-TCTL model checking procedure consists of three steps that are visualized in Fig. 5.15. The key idea of our approach is a reduction of the model checking problem for a timed GTS and a FO-TCTL specification to the well-studied TCTL model checking problem for timed automata [6, 7, 21]. Then, a standard timed model checking tool answers the question whether the MECHATRONICUML model fulfills its formal requirements.

In the first step, a so-called Gt-automaton is computed for the timed GTS. The Gt-automaton is a timed automaton where each of its states corresponds to a timed graph which can be derived from the initial graph of the timed GTS. Transitions result from derivations using the timed GT rules and are labeled with the guard and reset of the timed GT rule that was used for the derivation. We label each state with the clock constraints of the invariant rules that can be matched to the state. Each node in a state is labeled with a unique identifier that is preserved by the derivation. The set of clocks of the Gt-automaton corresponds to the union of the clock instances that have been created by the clock instance rules.

In the second step, we use the Gt-automaton to reduce the FO-TCTL formula to a standard TCTL formula. In particular, we exploit the identifiers of the nodes for replacing quantifiers and variables by boolean expressions with constants, only. An $\exists$ quantifier is replaced by a disjunction replacing the occurrences of the quantified

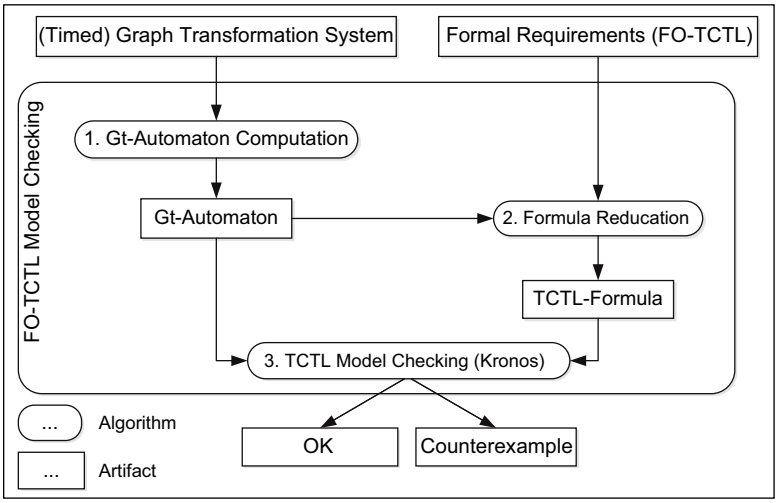

Fig. 5.15 Overview of FO-TCTL model checking

variable by all possible node identifiers occurring in the Gt-automaton. A $\forall$ quantifier is replaced analogously by a conjunction. Finally, we encode the identifiers by atomic propositions that can be processed by the timed model checker.

In the third step, we use the Gt-automaton and the TCTL formula as inputs to a standard timed model checking tool. We propose using Kronos [31] because Kronos provides a full TCTL model checking. UPPAAL [40], on the contrary, only supports a simple subset of TCTL. UPPAAL can be used with our method as well, if the supported TCTL subset is sufficient for the verification task.

Verification Procedure 2 – Shape Analysis: Commonly, models contain behavior that allows the runtime structure specified by the model to grow. An example of this is convoy coordination in the RailCab system, where new RailCabs can join existing convoys. Usually, and in this example as well, there is no natural limit to this growth.

There are two ways out of this. One is to use bounded model checking, which is what the method detailed above amounts to. Instead of checking the entire system, a finite subsystem is identified by bounds, such as maximum convoy size, and then checked. In order to construct the Gt-automaton, the entire state space of the system must be constructed, and thus all possibilities of infinite growth pruned at some arbitrary bound, like, e.g. 10 RailCabs. Any behavior within that subsystem is safe, yet nothing is known of the remainder. That means that any correctness result obtained using this method is only valid as long as there is no convoy longer that 10 RailCabs. As soon as there is, all verification results obtained with this bound in place are lost.

In this particular case, there might be some merit to limiting the number of RailCabs that can take part in a convoy a priori to a constant number. This is because the communication range of RailCabs is limited, as is the maximum deceleration

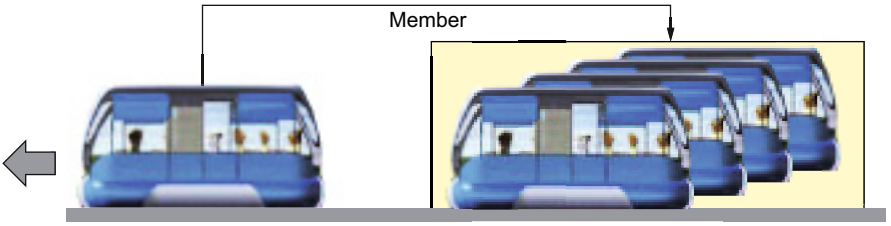

Fig. 5.16 An abstracted convoy

a RailCab is capable of, which limits the length of a potential convoy. However, such limitations usually only apply to hardware structures, such as convoys. Also, as the system evolves, physical parameters change. Better transmitters might extend a RailCabs WiFi range, improved brakes might improve maximum deceleration. Most successful distributed systems eventually outgrow any bound on size.

The second way to deal with infinite growth is called overapproximation, and that is what *shape analysis* essentially does. Instead of looking at a subset of the behavior of a system, in shape analysis one looks at a superset of it which has the property of being compactly (finitely) representable. This is done in such a way that safety properties that can be shown for the overapproximation, are also guaranteed to be valid for the original system.

Shape Analysis was initially a formalism used to abstractly describe heap structures in imperative programs [185]. In our work we have utilized the concepts developed for that formalism to create a verification algorithm that applies them to GTS [193, 194, 211]. This algorithm is generic and applies to all GTS. It is an instance of a class of algorithms performing abstract graph transformations. Other instances include [25, 137, 176].

At its core, the algorithm works by identifying groups of nodes in a given graph that have similar properties and grouping them together into one *summary node*. The resulting graph then is a *representative* for the set of all graphs where the summary node is replaced by a particular number of nodes. Thus a single abstract graph, called *shape*, can represent an infinite number of actual, concrete graphs. As an example, consider Fig. 5.16. Here, the rectangle represents an arbitrary number of follower RailCabs. The entire shape therefore represents a convoy of arbitrary size.

Such shapes can now be subjected to dynamic behavior, just as the original graphs were. If the abstraction was chosen well, we obtain a finite representation of the entire state space and can check then whether the given safety properties are valid or not. If they are, we have just proven the safety of the original system in its unconstrained form, e.g. the safety of convoy coordination regardless of the number of participants. If they are not, we get a counterexample. This counterexample can either be genuine, or it can be an artifact of the abstraction. This can be decided by retracing the counterexample obtained on the shape level on concrete graphs. If the counterexample is genuine, we need to fix the system, if it is not, we must refine the abstraction to remove the artifact that produced the counterexample.

The ability to verify infinite systems does of course not come without a price. The two main drawbacks of this method are increased complexity and undecidability. Abstraction introduces a lot of complex definitions and properties that make it hard to enrich with additional properties. This is the reason Shape Analysis is currently unable to take time into account in any form (unlike the method described above). Undecidability means that in its finished form, the algorithm will run fully automatically, but in the absence of human intervention there is the possibility that the algorithm will run forever. It is however possible to reduce the probability of this by allowing the algorithm access to as much domain specific information as possible to help it guide its abstraction refinement process.

5.2.4 Discrete Behavior

After proving the correctness, the real-time coordination patterns are integrated into the component implementation and refined if necessary. We provide an algorithm for checking the correctness of the refinement in Sect. 5.2.4.1. Additionally, we may integrate existing legacy components into a MECHATRONICUML model such that they meet the system's safety and liveness requirements; this is presented in Sect. 5.2.4.2. In Sect. 5.2.4.3, we will provide an automatic synthesis of component behavior to resolve dependencies that might exist between different real-time coordination patterns when they are combined in a component. Sect. 5.2.4.4 describes the specification of reconfiguration behavior of components. Finally, we will outline a planning technique that selects which runtime reconfigurations to apply to reach the system's objectives at runtime in Sect. 5.2.4.5.

5.2.4.1 Refinement of Real-Time Coordination Patterns

Real-time coordination patterns as introduced in Sect. 5.2.3 aim at reusing the modeled interaction in different applications. Therefore, real-time coordination patterns are specified independent of a concrete component implementation. A concrete implementation often has to refine this behavior, e.g. add internal computations or access internal variables, thereby introducing new internal states and/or transitions. Such modifications of the behavior may invalidate the formal requirements that have been proven for the real-time coordination pattern using the design-time verification procedure (cf. Sect. 5.2.3.2).

As described in Sect. 5.2.3.1, real-time coordination patterns may specify 1:*n* communication with runtime reconfiguration. Then, the reconfiguration operations need to be considered when checking for a correct refinement [91]. This problem is more difficult than the 1:1 communication case as additionally the creation and deletion of the protocols and the dependencies between the instances has to be considered. Since the abstract real-time coordination patterns are verified formally beforehand, the refinement must preserve these verified properties.

The overall refinement approach is shown in Fig. 5.17. First, a real-time coordination pattern is modeled as described in Sect. 5.2.3.1. Afterwards, we verify this real-time coordination pattern using the verification approach outlined in Sect. 5.2.3.2

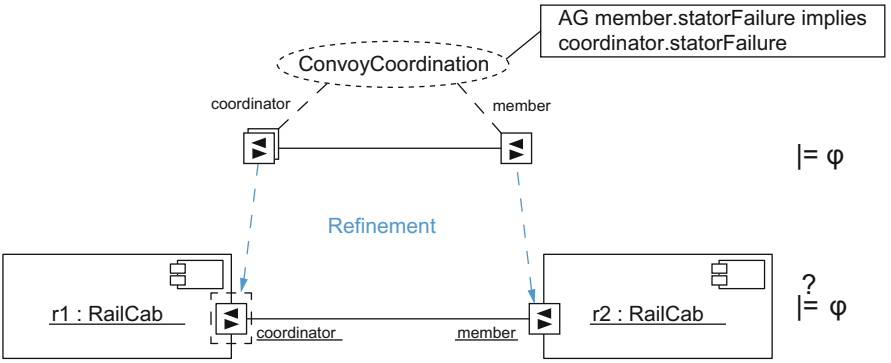

Fig. 5.17 Refinement approach

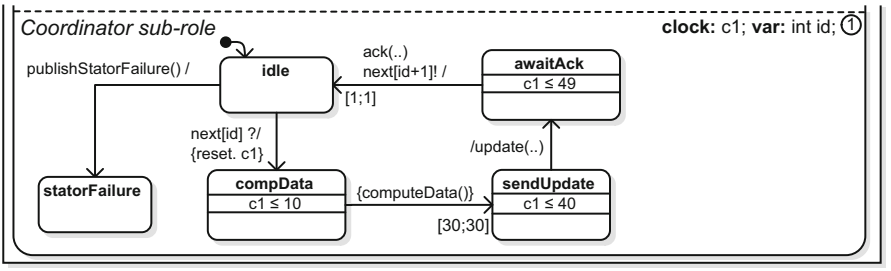

Fig. 5.18 Excerpt of the refined real-time statechart of the convoy coordinator

for proving that the specified properties φ are valid. Then, both roles are refined to a port as part of a component implementation. Finally, we check the conformance of the component implementation to the roles of the real-time coordination pattern by checking for a correct refinement.

As an example, consider the excerpt of a *coordinator* real-time statechart shown in Fig. 5.18. The real-time statechart has been refined with respect to the real-time statechart shown in Fig. 5.12 by inserting a new state *compData*. The transition from *compData* to *sendUpdate* specifies a side effect that computes new data to be sent via the *update* message which causes the transition to consume 30 time units of computation time. Since the timing values have been changed and a new state has been added, it is not clear whether the refined *coordinator* multirole still fulfills all verified properties.

In the literature, two basic types of refinements are defined: *simulation* and *bisimulation* that exist for untimed systems as well as for real-time systems [37, 202]. These standard refinement definitions are based on automata and disregard run-time reconfiguration that is used in our approach. Additionally, simulation is a very weak condition as it does not require the refined system to specify all communications being specified in the abstract real-time coordination pattern. Obviously, this is not sufficient for safe protocol reuse. In contrast, bisimulation is a very strong condition

as it requires the refined system to perform exactly the same in exactly the same time as the abstract real-time coordination pattern. This does not allow applying changes to the protocol, thereby limiting the set of component implementations complying to the abstract real-time coordination pattern.

Therefore, we introduce a refinement definition called *relaxed weak timed bisimulation* [91] that relaxes the strict conditions of a bisimulation by using information of our component model. We assume that each port has an unbounded input buffer for received messages that can accept messages at any time. If a statechart receives a message, the message is taken out of the input buffer. Using such an input buffer, the point in time, at which a message is consumed by a real-time statechart, does not matter for a communication partner. Therefore, we allow that the refined role processes messages later than the abstract role. Delaying a sent message is not allowed as we only consider one role in the refinement and we cannot assume that the receiver of the message can still receive it after the time interval specified by the abstract real-time coordination pattern has elapsed.

We consider the run-time reconfiguration of real-time coordination patterns by a so-called structural refinement as defined in [90]. It ensures that the refined real-time coordination pattern executes its reconfiguration operations in the right time intervals by relating the subrole and singlerole instances including their connections in both the abstract and refined real-time coordination pattern instance.

Checking for a correct refinement requires checking the refined role implemented in a port of a component against the abstract role of the real-time coordination pattern. This requires exploring the state-spaces of both and to compare the intervals in which messages are sent or received. The refinement check algorithm is based on the same implementation as the verification procedure introduced in Sect. 5.2.3.2. In [91], we showed for the RailCab example that this is more efficient than verifying all properties for the refined real-time coordination pattern again. The reason is that we do not need to consider the connector, but only one role at a time.

5.2.4.2 Integration of Legacy Components

The software of self-optimzing mechatronic systems is usually a network of components. By MECHATRONICUML we provide a sound method that guarantees a high quality of this software. However, in domains like the automotive industry the development of new functions is an exception rather than the norm. In many cases, components exist and have to be reused where no model or only incomplete models exist. On the one hand, reuse accelerates the development of the system. On the other hand, one can rely on the quality the component has proven in the past. Both saves development costs.

These so-called *legacy components* must be integrated into the newly built system such that they meet the system's safety and liveness requirements. Therefore, we reconstructed a real-time statechart that specifies the communication behavior of the legacy component. The reconstructed real-time statechart is used to verify the correct integration of the legacy component into a MECHATRONICUML model [96, 97].

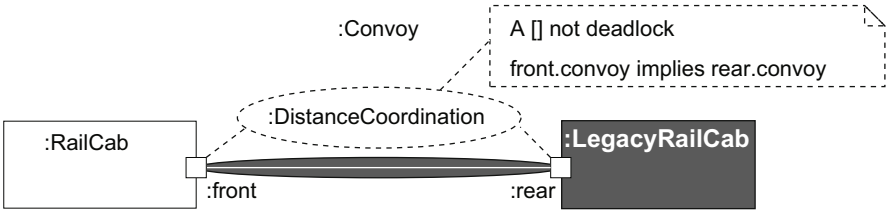

Fig. 5.19 Architecture with legacy RailCab

Fig. 5.19 shows a scenario where an old RailCab *LegacyRailCab* communicates with a *RailCab* developed with MechtronicUML. Here we assume that the developer does not have a MechtronicUML model of the communication software of the *LegacyRailCab*. Both RailCabs shall communicate using the *DistanceCoordination* Pattern. The communication behavior of the rear role must satisfy the liveness constraint that no deadlock occurs (`A[] no deadlock`) and the safety contraint that both RailCabs drive in convoy mode (`front.convoy implies rear.convoy`) when applying the *DistanceCoordination* pattern.

The role behavior with which the legacy component has to interact is called *context*. In Fig. 5.19 the context is the front role of the component RailCab. An integration is successful, if the communication between the legacy component and the context is error-free. This is specified by safety properties and liveness properties. Moreover we need to guarantee that, depending on the communication behavior, the correct control behavior is executed. The continuous behavior is identified by system identification.

In order to integrate a legacy component into a MECHATRONICUML model, the following requirements must be met. The legacy component must provide an interface that is accessible by the developer. This interface must define all incoming and outgoing messages used for communication, all signals used by embedded feedback controllers, and all information which is relevant for executing the component (e.g. execution periods). This, however, does not require additional effort in the domain of safety-critical systems, as this is typically part of the system specification.

Moreover, we assume that initially the component is in its starting state or in a quiescent state (cf. [127, 213]). We further assume that the developer is able to put the component in such a state.

The information provided by the interface of the legacy component may differ substantially. We distinguish three cases. First, *(1)* the interface provides functionality to query its current state. If this is not the case, we distinguish the cases where *(2)* the source code of the interface is provided and *(3)* no source code is provided.

Depending on the provided information, we apply different methods to integrate the legacy component. For case *(1)* we apply grey-box-checking, for case *(2)* white-box-checking, and for case *(3)* black-box-checking. We will shortly explain these methods below. We will introduce the basic approach of iterative learning by grey-box-checking. Thereafter we will point out how the other methods differ from grey-box-checking.

Grey-Box-Checking

We start grey-box-checking with a *chaotic closure*. This chaotic closure is an over-approximation of the actual communication behavior. The chaotic closure is a behavior model that enables all possible communication behaviors and also a deadlock of the legacy component at any time. However, not all of this behavior may be implemented in the legacy component. Therefore, the behavior is defined step-by-step by limiting the behavior of the chaotic closure until it conforms to the behavior of the legacy component.

First, we verify the safety and liveness properties on the combination of the chaotic closure and the context. If the verification yields a counterexample, the counterexample is used to generate a test case for the legacy component. The test case is generated by extracting all inputs and outputs including their time or appearance. The legacy component is executed with the extracted inputs. The test has passed, if the extracted outputs are observed from the legacy component at identical points of time as in the counterexample. Otherwise, the test has failed.

If the test case passed, we have found a valid counterexample. This means, one of the required safety and liveness properties are not satisfied. At this point, reverse engineering either stops or the requirements on the system need to be relaxed. If the test case failed, the observed behavior is used to refine the chaotic closure. Therefore, the current state is requested from the legacy component. If a new state is found, a new state is created for the chaotic closure. Edges are built according to the observed transitions. This process continues until a valid counterexample is found or all traces of the context have been taken into account.

Black-Box-Checking

Black-box-checking also uses a counterexample guided refinement. But here the legacy component does not provide the functionality to request its current state. Our solution is to construct a candidate for the behavior of the legacy component. The candidate is constructed by an extension of the learning algorithm of Angluin (1987) [9], an efficient approach for learning a deterministic finite automaton of a black-box. We extended the algorithm of Angluin (1987) to take into account incoming and outgoing messages and time.

The candidate and the context are verified by model checking with respect to the safety and bounded liveness properties of the legacy component. If the verification is successful, it is proven that the candidate is equivalent to the behavior of the legacy component. Otherwise, the counterexample is used to improve the candidate.

White-Box-Checking

For White-box-checking, we assume that we know the source code of the legacy component. To safely integrate the legacy component into the system, we generate source code from the context model. The source code of the context and the legacy component are embedded into a framework. The framework simulates scheduling,

message exchange and timed behavior. The resulting system is verified by a source code model checker with respect to the safety and bounded liveness properties.

5.2.4.3 Synthesis of Component Behavior

As described in Sect. 5.2.4.1, the roles of a real-time coordination pattern are refined to ports of a component. Often, a component needs to engage in more than one interaction in order to fulfill its function, i.e. it refines roles of several real-time coordination patterns.

In the RailCab example, a RailCab interacts with other RailCabs for building convoys, but it also needs to register at the track section it is currently driving on. The registration at track sections is required to ensure that each track section is only accessed by RailCabs driving in the same direction. Up to this point, all interactions defined by real-time coordination patterns are operating independent of each other. For a safe convoy operation, however, we need to fulfill the requirement that "in convoy operation mode, each participating RailCab has to be registered to a track section" [54]. Thus, there may exist dependencies between real-time coordination patterns when they are combined in a component.

In our previous works, we used a so-called **synchronization real-time statechart** in a component for resolving such dependencies [75]. The specification of such a synchronization real-time statechart was subject to the developer. Specifying a synchronization real-time statechart, however, is a difficult and error-prone task. This is because, on the one hand, it needs to resolve the dependency and, on the other hand, it must not remove communications specified by one of the roles. If communications specified by one of the roles was removed, the results of the design-time verification will not necessarily be valid anymore.

As a solution, we provide an automatic synthesis of a component behavior that automatically resolves the dependencies and ensures the *role conformance* of the resulting behavior [54]. The dependencies are either specified by *state-composition rules* referring to the states of the roles or by *event-composition automata* referring to the sent and received messages of the roles. To perform the synthesis, first, a product automaton including the behavior of all ports is constructed and the dependencies are resolved automatically by applying the state-composition and event-composition rules. Then, the role conformance check ensures that all communications originally specified by both roles are still available in the synthesized behavior. That, in turn, ensures that all verified properties are still valid in the synthesized behavior.

We illustrate our approach by a simplified example using UPPAAL timed automata [21]. Fig. 5.20a) shows a simplified convoy behavior consisting of the two states *noConvoy* and *convoy*. Fig. 5.20b) shows a timed automaton for registering at a track section.

In the following, we will first introduce state-composition rules. Thereafter, we will explain event-composition automata before outlining the synthesis algorithm.

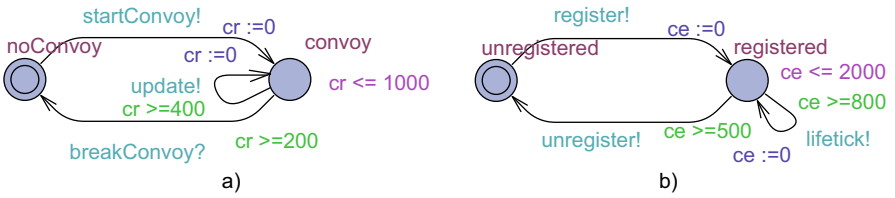

Fig. 5.20 Simplified behavior models for a) convoy coordination and b) registration

State-Composition Rules

State-composition rules define restrictions for the component behavior based on the states of the role automata. In particular, they specify forbidden state combinations of the input automata. The state information encoded in state-composition rules includes timing information that forbids certain state combinations only for a specific time interval. The state-composition rules need to be specified by a developer when creating the component.

An example of a state-composition rule for the automata in Fig. 5.20 is given by:

$$r_1 = \neg((unregistered, true) \wedge (convoy, true)).$$

The state-composition rule r_1 formalizes the requirement that a RailCab may only drive in convoy mode while it is registered at a track section. Consequently, r_1 forbids the component behavior to be in states *unregistered* and *convoy* at the same time. A reference to a state, e.g. $(unregistered, true)$, is a tuple where the first entry refers to the name of the state and the second entry defines a clock restriction. In this case, the clock restriction is *true* for both states which means that the state combination is not allowed for all possible clock values of all used clocks.

Event-Composition Automata

Event-composition automata define restrictions for the component behavior based on sequences of messages that the role automata may send or receive. They monitor the messages that are sent or received by the role automata. Consequently, only messages used in one of the role automata may be used in an event-composition automaton.

In our example, we assume another requirement for the RailCab component which states that a RailCab needs to be registered at a track section for at least 2500 time units before it can start at convoy. This requirement refers to a *message* of the automaton and, thus, can not be specified by state-composition rules. The event-composition automaton for the requirement is shown in Fig. 5.21.

The automaton starts in state *ec_initial*. If it monitors that the RailCabs sends *register*, it switches to *ec_registered* thereby setting the clock *ec_c1* back to 0. The message *startConvoy* may be monitored, at the earliest, 2500 time units later which is specified by the time guard of the corresponding transition. The automaton stays in *ec_registeredConvoy* until the RailCab sends an *unregister* message.

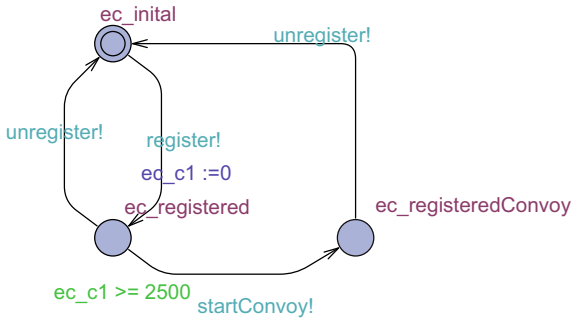

Fig. 5.21 Example of an event composition automaton eca_1

Synthesis Algorithm

The synthesis algorithm takes three inputs. These are the automata of the roles that should be synthesized to a component behavior as well as the state-composition rules and the event-composition automata that define the restrictions for the synthesis. Based on these inputs, the resulting automaton for the component is synthesized in four steps. We briefly outline these steps in the following. For a complete description, we refer to Eckardt and Henkler (2010) [54].

Step 1 – Computing the parallel composition: In the first step, we compute the parallel composition of the role automata. The parallel composition is derived from the parallel composition operator of CCS (Calculus of Communicating Systems, [144]) which is also used in UPPAAL [21]. The parallel composition contains the complete behavior of the role automata.

Step 2 – Applying state-composition rules: In the second step, the state-composition rules are applied to the parallel composition resulting from step 1. Iteratively, each state-composition rule is applied to each state of the parallel composition automaton. If the state fulfills the state conditions imposes by the state-composition rule, the time condition is added to the invariant of the state. If the resulting invariant of the state is false, the state is removed from the parallel composition along with all its incident transitions.

The state $(convoy, unregistered)$ of the product automaton fulfills the state conditions of r_1. Consequently, the time conditions $\neg((true) \wedge (true))$ is added to the invariant which makes it $false$ and causes the state to be removed.

Step 3 – Applying event-composition automata: In the step 3, all event-composition automata are applied iteratively to the automaton resulting from step 2. Since the event-composition automaton is a timed automaton, the application is similar to the parallel composition of step 1. The difference is that the event-composition automaton only monitors the messages that are sent and received by the role automata.

After the parallel composition, each state refers both, to the states of the initial parallel composition and the state of the event-composition automaton. All states resulting from the parallel composition that are not reachable from the initial location are removed from the automaton.

Step 4 – Checking Behavior Conformance: In step 4, we check for behavior conformance of the synthesized automaton resulting from steps 1-3. In step 2 and step 3, every behavior not allowed by the state-composition and event-composition rules have been removed from the parallel composition of the role automata. Due to the removal of behavior, it is not ensured that the communications specified by the roles of the real-time coordination pattern are still contained in the synthesized automaton. As a consequence, it is not guaranteed that the synthesized behavior still fulfills all properties that have been verified for the real-time coordination pattern.

By checking the synthesized automaton for behavior conformance to the roles of the real-time coordination pattern, we ensure that the verified properties still be valid. *"In order to preserve the relevant role behavior, we need to ensure that in the refined component behavior, every timed safety properties and every untimed liveness properties are preserved. This would imply that no deadlines of the original role automata are violated while all events of the original automata are (in the correct order) still visible within the original time interval. If both of these properties are preserved, we say that the refined component behavior is role conform."* [54]

If the synthesized automaton resulting from steps 1-3 is not behavior conform, the synthesis reports an error. In this case, it is not possible to synthesize an automaton that fulfills the state-composition and event-composition rules while specifying the behavior of the roles of the real-time coordination pattern. In this case, the engineer needs to specify the synchronization real-time statechart manually and ensure correctness by repeating the verification steps introduced above.

We refer to our technical report [55] for a detailed proof of the correctness of the synthesis.

5.2.4.4 Modeling Component Reconfiguration

In Sect. 5.2.3 we showed how reconfiguration is specified for real-time coordination patterns. Additionally, we can specify reconfiguration behavior for the components of our component model [43]. For the specification of reconfiguration behavior, we use component story diagrams [200] again. We specify component story diagrams for each component of the component model that needs to perform reconfiguration.

Figure 5.22 shows an example of a component story diagram that specifies the behavior for becoming a convoy member for the *DriveControl* component in Fig. 5.8. Becoming a convoy member requires to instantiate the subcomponent *MemberControl* and to connect it to the *SpeedControl* such that it can provide the reference speed for the speed controller.

Since we model the component reconfiguration by component story diagrams, we can use our design-time verification procedure introduced in Sect. 5.2.3.2 for verifying the reconfiguration behavior. We refer to [94] for more technical information on executing reconfiguration in a hierarchical component model.

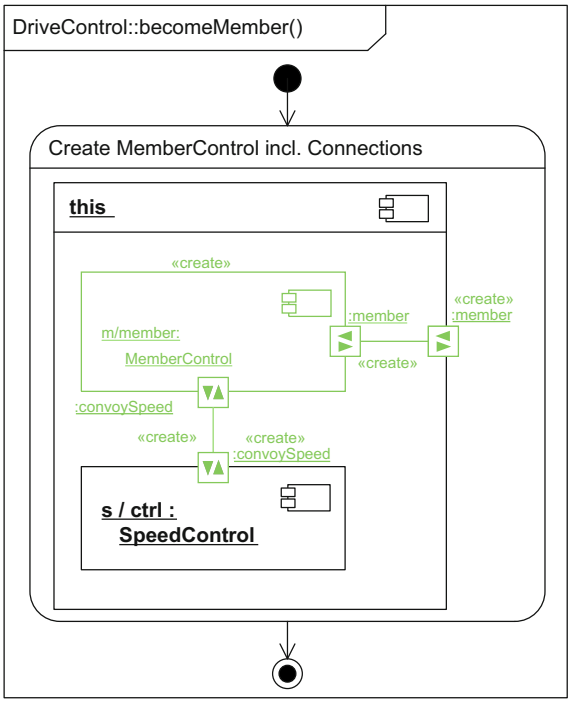

Fig. 5.22 Reconfiguration rule of *DriveControl* to become a convoy member

5.2.4.5 Safe Planning

In each configuration of a self-optimizing mechatronic system, a large set of runtime reconfigurations can be applied to adapt the system to changes in its environment at runtime. Selecting which runtime reconfigurations to apply can be a complex task. Self-optimizing systems often have superordinated objectives that should be reached during execution, like optimizing the energy consumption or achieving user-specified objectives. These objectives have to be respected when selecting which runtime reconfigurations to apply. However, selecting runtime reconfigurations that are likely to help to achieve the objective is no trivial task. Since the selection of runtime reconfigurations is supposed to happen autonomously (a human intervention would not meet the response time requirements of self-optimizing mechatronic systems), it has to be planned by a software system.

To prevent unsafe configurations, e.g. an inadequate safety distance between two RailCabs, from occurring in a plan, the planning system should further take safety requirements into account. The safety requirements restrict the set of valid configurations, i.e. they specify which configurations are not allowed to occur in a resulting plan. In contrast to the verification of runtime reconfigurations (cf. Sect. 5.2.3.2), where the absence of unsafe states is guaranteed categorically, this technique allows unsafe states to exist in the reachability graph, but plans the reconfigurations in such a way that no unsafe state is passed through. The latter approach is chosen for specific safety requirements that can not be verified by the design-time verification

or impose to many restrictions to the specification of the runtime reconfigurations. In [69, D.o.S.O.M.S. Sect. 3.2.9] , we present a technique that considers these safety requirements when planning runtime reconfigurations.

Our approach uses GTS as an underlying formalism. The transition system of the GTS can be constructed by successively applying the graph transformations to the initial configuration and its successor configurations. The planning task is to find a path in this transition system so that a target configuration is reached. A safe planning task is basically the same, but includes the requirement that no potentially unsafe configuration is passed through. In our case, the initial configuration corresponds to a UML object diagram and each transition is the result of applying a graph transformation rule to the configuration.

To solve these planning tasks, different algorithms and techniques exist. One of the approaches is to translate the planning problem into an available off-the-shelf planning system. These traditional planning systems, however, employ models different from GTS. They employ models with first-order literals that are usually compiled into a propositional representation by grounding predicates and actions. While a translation is basically possible, there are some restrictions because typical planning languages, like the Planning Domain Definition Language (PDDL), which is the current *de facto* standard in academia, has a different expressive power than GTS. By planning directly in the transition system defined by the GTS, we avoid these problems.

Given a goal specification, our model can be fed into a planning system, e.g. [57], that directly plans on the transition system that results from the model. Therefore, no translation to a dedicated planning language and thus no restriction to the expressive power of GTS is necessary. Unsafe configurations are recognized by the planning system and not allowed in a valid plan. The resulting plan specifies a sequence of runtime reconfigurations that safely turn the system from its initial configuration into a target configuration.

5.2.5 Simulation of Hybrid Behavior

The software of a self-optimizing mechatronic system consists of discrete software developed with MECHATRONICUML and continuous controllers developed with a tool like MATLAB/Simulink. That results in a so-called hybrid behavior specification [98]. The design-time verification procedures described in Sect. 5.2.3.2 can only be applied to the discrete software. Corresponding hybrid verification techniques [5] do not scale sufficiently for complex mechatronic systems.

Therefore, we use simulation for testing the complete hybrid system and, in particular, the correct integration of discrete software and controllers. We provide an automatic model transformation for transforming the MECHATRONICUML model into an input of a simulation tool, namely MATLAB/Simulink [93, 95].

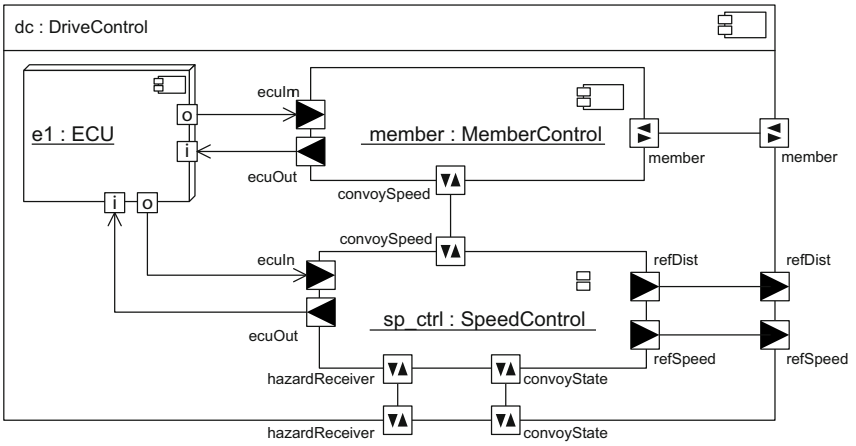

Fig. 5.23 Deployment diagram

5.2.6 *Specification of Deployment*

The software components that have been created with MECHATRONICUML need to be deployed on a hardware. This hardware comprises sensors that provide input signals, controllers that control actuators and computing hardware that executes the software components. In MECHATRONICUML the deployment of software to hardware is specified by deployment diagrams. Hardware entities are represented by hardware nodes which communicate unidirectionally via hardware ports.

Figure 5.23 shows an example of a deployment diagram which specifies the deployment of an instance of the component type *DriveControl* (cf. Fig. 5.8) to an ECU. In deployment diagrams hardware nodes are drawn as boxes. Hardware ports are drawn as squares that contain either an "i" for incoming signals or an "o" for outgoing signals. The embedded component instances *member:Member* and *sp_ctrl:SpeedControl* of *dc:DriveControl* are both connected to the hardware node *e1:ECU* which represents the ECU that executes these component instances.

5.2.7 *Integration of Self-healing Behavior*

The self-optimization capabilities of self-optimizing mechatronic systems can be used to repair systems in case of failures at runtime. This so-called self-healing can be used to reduce occurrence probabilities of hazards in systems which are applied in safety-critical environments. Self-healing systems react to failures by a reconfiguration of the system architecture during runtime.

Take for example the speed control of the RailCab. The electric current to be set on the linear drive depends on the speed of each wheel which is measured by speed sensors. If a failure occurs in at least one of the speed sensors, a wrong value is passed to the current controller. This causes the RailCab to drive at a wrong speed which can result in a collision. To prevent such a situation, a self-healing operation

can be specified in form of a reconfiguration which replaces the faulty sensor by a spare which is still working.

This reaction is subject to hard real-time constraints because reacting too late does not yield the intended self-healing effects. Consequently, it is necessary to analyze the propagation times of failures and the effect of a reconfiguration on the propagation of failures [166]. In [69, D.o.S.O.M.S. Sect. 3.2.13] , we present an approach for the analysis of self-healing operations which specifically considers these properties.

Not all parameters which are needed to analyze self-healing operations, e.g. the concrete system architecture, are known at design time. When, for example, RailCabs have become ready for the market, they will be produced by more than one manufacturer. Then it will be possible that two vehicles that come from different manufacturers meet on the track. In order to build a convoy they need to establish a connection. This connection leads to a system architecture that was unknown at design time, because the system architecture of the unknown vehicle was, of course, unknown to the developers of the RailCab.

Consequently, the effect of self-healing operations needs to be analyzed during runtime. We developed an approach to analyze self-healing operations at runtime. It prevents the construction of system architectures at runtime where self-healing operations can not reduce the occurrence probabilities of hazard so that they become acceptable.

Based on the system's current architecture, we compute each reachable system architecture for a fixed number of subsequent reconfigurations at runtime. We then analyze the self-healing operations. If the hazard occurrence probability of a reachable system architecture exceeds the system's acceptable hazard occurrence probability event after the application of a self-healing operation, the reconfiguration rule that constructs this system architecture is locked.

5.2.8 Code Generation

We use the models that have been created using MECHATRONICUML for an automatic generation of the source code of the self-optimizing mechatronic system. An approach for code generation has been introduced in [1]. Alternatively, we can use the MATLAB/Simulink code generation facilities to generate code out of the MATLAB/Simulink model that we created for simulation (cf. Sect. 5.2.5).

5.3 System Optimization

Harald Anacker, Michael Dellnitz, Kathrin Flaßkamp, Philip Hartmann, Christian Horenkamp, Bernd Kleinjohann, Lisa Kleinjohann, Martin Krüger, Sina Ober-Blöbaum, Christoph Rasche, Maik Ringkamp, Robert Timmermann, Ansgar Trächtler, and Katrin Witting

In order to develop self-optimizing systems, optimization plays a crucial role. In the following section, a couple of methods are presented which allow a systematical and

formal optimization of the system behavior. In contrast to successive improvement, which often has to be done manually, these methods aim at automatically seeking the optima, i.e. points of no further improvement. During the conceptual design, the relevant objectives are identified and a general control structure is designed, that is capable to alter the fulfillment of the objectives, cf. Sect. 3.2. At the beginning of the system's design and the development, concrete mathematical models of the system behavior are created in the respective domains as well. These are the inputs for the methods of **model-based self-optimization** described in the first seven sections. The following sections deal with behavior-oriented self-optimization which describes methods without an explicit physical model of the system or process. Instead, these approaches work on mapping input values to output values. The actual system and the considered process are observed as a black box.

In the first section, we get back to multiobjective optimization which has already been introduced in Sect. 1.4.1.1 and present some more details about novel set-oriented algorithms for solving multiobjective optimization problems (MOP) in Sect. 5.3.1. The algorithms can be used to compute optimal system configurations that considers several conflicting objectives in one single MOP.

Self-optimizing systems are often complex systems consisting of several subsystems which are hierarchically structured (see Sect. 1.3 for an introduction into the structuring concept). If each system comes with its own objectives, one also gets a hierarchy of MOPs that has to be solved. Section 5.3.2 describes an approach on how to handle such optimization problems. The following section, Sect. 5.3.3, is closely related to hierarchical optimization. A so-called hierarchical model is introduced that can be used to significantly reduce the model complexity of hierarchical systems by means of parametric model-order reduction.

The following two Sections 5.3.4 and 5.3.5, deal with MOPs which also depend on continuous external parameters. Two numerical methods are presented that are used to solve such problems efficiently and to identify so-called robust Pareto points.

Optimal Control, a different aspect of system optimization, is addressed in Sections 5.3.6 and 5.3.7. In optimal control problems the goal is to compute time-dependent steering maneuvers as introduced in Sect. 1.4.1.2. In Sect. 5.3.6, the optimal control technique DMOC is presented which is especially tailored for the optimal control of mechanical systems. In order to improve the solvability of optimal control problems by creating efficient initial guesses, a motion planning technique based on motion primitives is described in Sect. 5.3.7. Within this approach, several short pieces of simply controlled trajectories are sequenced to longer trajectories.

In Sect. 5.3.8 one approach for decision making (cf. Sect. 1.4.1.3) is presented that is called hierarchical hybrid planning. The hierarchical model is used to simulate the prospective system behavior and a discrete planning problem is defined based on the simulation results as well as on a pre-computed Pareto set.

All these different methods of system optimization need a physically motivated mathematical model of the self-optimizing system. If such a detailed model is not

available for a specific task, methods of the **behavior-oriented self-optimization** can be used (see Sect. 1.4.2 for an introduction). Statistical Planning, described in Sect. 5.3.9, is one of these methods. It uses statistical data to compute plans for mechatronic systems based on an environmental model given by a discrete finite nondeterministic Markov decision process. A different approach is presented in Sect. 5.3.10. A discrete planning problem is defined that can be used to find a sequence of operation modes which describe a transition from an initial state to a predetermind goal state. Section 5.3.11 presents an approach to realize a multiagent system by behavior planing, to open up the advantage given by the possibility for intelligent communication of individual subsystems. Finally we will present the application of solution pattern presented in Sect. 4.5 to make the method hybrid planning available to the developers.

5.3.1 Set-Oriented Multiobjective Optimization

Michael Dellnitz, Kathrin Flaßkamp, and Christian Horenkamp

The demand for multiobjective optimization in the context of self-optimizing systems was already shown in Sect. 1.4.1. Here, we review algorithms developed and applied within the CRC 614 for the computation of the entire Pareto set of multiobjective optimization problems. The basic idea of these methods is to use set-oriented algorithms for dynamical systems (cf. [46]).

We reconsider the multiobjective optimization problem (MOP Eq. (1.1)) introduced in Sect. 1.4.1

$$\min_{\mathbf{p}\in S\subset\mathbb{R}^n} \mathbf{F}(\mathbf{p}), \tag{5.1}$$

where $\mathbf{F}:\mathbb{R}^n\to\mathbb{R}^k$, $\mathbf{F}(\mathbf{p})=(f_1(\mathbf{p}),...,f_k(\mathbf{p}))^T$ is the vector of $k\in\mathbb{N}$ objective functions, $\mathbf{p}$ the optimization variable or design variable of dimension $n\in\mathbb{N}$, and S denotes the feasible set. The necessary conditions for Pareto optimality are given by the Karush-Kuhn-Tucker (KKT) equations (cf. Sect. 1.4.1). Here, we consider the left hand side of the KKT equations as a map $\mathbf{H}:\mathbb{R}^{k+n}\to\mathbb{R}^n$, with $\mathbf{H}(\beta,\mathbf{p})=\sum_{i=1}^k \beta_i\nabla f_i(\mathbf{p})$ and $\beta=(\beta_1,...,\beta_k)$ with $\beta_i\geq 0$ for all $i\in\{1,...,k\}$ and $\sum_{i=1}^k\beta_i=1$ (cf. Eq. (1.2) in Sect. 1.4.1). By finding zeros of the map H, we identify points that satisfy the necessary optimality conditions. Therefore, the use of zero finding strategies as well as the minimization of the function H are essential steps in many techniques for solving multiobjective optimization problems.

5.3.1.1 Set-Oriented Solution Techniques for Multiobjective Optimization

The set-oriented solution techniques for multiobjective optimization are implemented in the software package GAIO[11]. They can be divided into two approaches: subdivision methods and recovering methods, which we shortly introduce in the following (cf. [190] for a detailed overview).

[11] Global Analysis of Invariant Objects, see `www.math.upb.de/~agdellnitz`

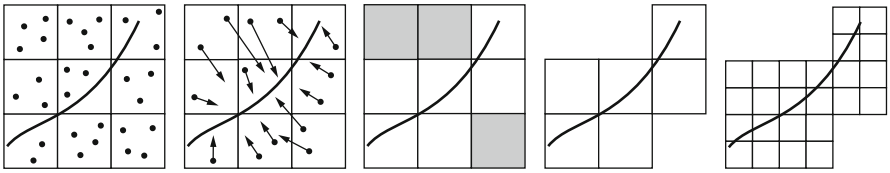

Fig. 5.24 Illustration of the subdivision algorithm: it alternates between subdivision and selection steps to approximate the Pareto set by a box covering

The **subdivision** procedure (cf. Fig. 5.24 for a sketch) starts with a box that covers the admissible set of optimization parameters and approximates the Pareto set by a successive refinement and selection of boxes. After every subdivision step, a gradient method is applied to chosen test points in all boxes. This iterates the test points forward, possibly into other boxes. The selection step deletes all boxes that do not contain iterated test points and only keep the other boxes for further subdivision. This scheme generates a box covering of the Pareto set with desired refinement. A sampling algorithm (cf. [190]) that does not require gradient information can be used instead of the gradient step.

Recovering techniques are applied to fill gaps in the covering of the Pareto set. Under certain conditions, the Pareto set (locally) forms a manifold [100], i.e. in the neighborhood of already known Pareto points further points can be found. The recovering algorithm is similar to a predictor corrector method, which is typically used for numerical integration. Based on an initial partial box covering, new test points are generated nearby (prediction step) and then iterated until they fulfill the KKT conditions (correction step). In this way, connected components of the Pareto set can be found if at least one Pareto point of this component is already known (cf. Fig. 5.25).

In the following two sections, we present two basic strategies to use the recovering technique. Firstly, the recovering techniques are applied in the preimage space (space of optimization parameters). This approach reaches its limitations when the number of design variables is high. In such a case one can pursue a second strategy, for which the recovering techniques are applied in the image space (space of objective functions). This is more suitable if the number of design variables is high but the number of objectives is small as discussed in Sect. 5.3.1.3.

5.3.1.2 Set-Oriented Recovering Methods in the Preimage Space Applied to the Multiobjective Optimization of the Test Vehicle Chameleon

In this subsection we review a recovering technique for the approximation of the Pareto set based on a predictor corrector method working in the preimage space. This method is a very efficient tool for the approximation of a finite representation of the entire Pareto set and has been successfully applied for the multiobjective optimization of the test vehicle Chameleon within the CRC 614.

In principle, starting with a point $\mathbf{p}^*$ of the Pareto set, the following steps are performed:

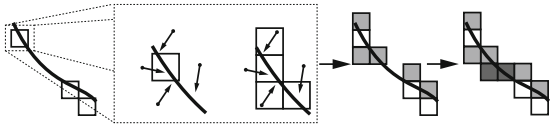

Fig. 5.25 Illustration of the recovering algorithm: starting with an initial partial covering of the Pareto set, a full covering is computed by a generation and mapping of test points near existing boxes

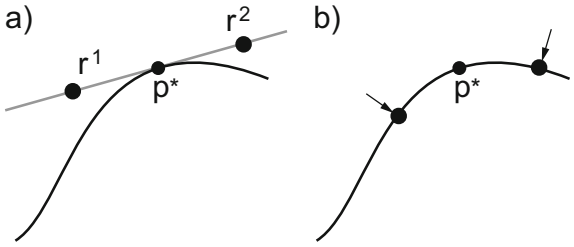

Fig. 5.26 Illustration of the predictor-corrector method: (a) Predict points $\mathbf{r}^1$ and $\mathbf{r}^2$ in the neighborhood of $\mathbf{p}^*$. (b) Correct points such that they lie on the Pareto set.

1. Predict points $\mathbf{r}^1, ..., \mathbf{r}^m$ in the neighborhood of $\mathbf{p}^*$.
2. Correct the points $\mathbf{r}^1, ..., \mathbf{r}^m$ such that they lie on the Pareto set by minimizing the norm of the KKT equations and adding the boxes containing the corrected points.

An illustration of this technique can be found in Fig. 5.26. The number of predicted points $m \in \mathbb{N}$ has to be sufficiently large in order to cover the Pareto set after the correction step well. Therefore, the bottleneck of this method is the prediction step where new test points are generated near an initial solution $\mathbf{p}^*$. A common way for the generation of new test points is to linearize the Pareto set around the initial solution $\mathbf{p}^*$ by an approximation of the tangent space of the Pareto set in $\mathbf{p}^*$ (*grey line in Fig. 5.26*). In general, one can use the Hessians of all objectives, but this approach reaches its limitation when high-dimensional models, where n is large, are considered. In the course of the research of the CRC 614, a novel method has been developed for the treatment of high-dimensional MOPs by successive approximation of the tangent space [181]. In detail, a new algorithm has been stated, where the tangent space is approximated by secants. This algorithm leads to an efficient approximation of the Pareto set of high-dimensional MOPs. Table 5.1 shows the CPU time for a scalable multiobjective optimization problem with three objectives which are taken from [189] for the recovering algorithm using the tangent space approximation (R_C) and the new algorithm (R_S) developed in [181]. For the test problem a significant speedup can be obtained for large n.

Within the corrector step in which the predicted points are corrected such that they lie on the Pareto set, many efficient minimizers make use of derivatives of the objective function $\mathbf{F}$. In many applications only program code for the objective $\mathbf{F}$ is provided and the corresponding derivatives, if existent, can not be determined analytically, thus other techniques are required. For example, finite differences can

Table 5.1 Comparison between the classical recovering algorithms R_C and a method using a successive approximation of the tangent space R_S

dimension of MOP		R_C	R_S
100	CPU time	2.9	2.9
200	CPU time	14	11.9
500	CPU time	134	91
1000	CPU time	965	500

be used, however, this approach leads to inaccurate derivatives which slows down the correction step. Alternatively, algorithmic differentiation (also called automatic differentiation) can be used (cf. [81]). These techniques automatically compute formulas for the derivatives based on the program code of the optimization problem for example.

In [182] the recovering technique of [181] has been combined with algorithmic differentiation. In more detail, the feasible set S of a MOP has been described as a zero set and the recovering procedure is adapted as follows: Let $\mathbf{p}^*$ be a solution of the MOP (1.1), then for the prediction step select neighboring points of $\mathbf{p}^*$ along the feasible set S and correct them to points on S. After the correction step a non-dominance test is performed to ensure that only the Pareto set is approximated as a subset of the feasible set S. For all non-dominated points, the predictor-corrector step is repeated until a covering of the Pareto set is reached. For the correction of the predicted points, the derivatives involved are calculated by an algorithmic differentiation method.

This method was successfully applied for the multiobjective optimization of the distribution of the tire forces for a braking maneuver of the test vehicle, Chameleon, which is described in more detail in Sect. 2.3. The tire forces of the Chameleon can be influenced individually within the physical and technical restrictions [175], hence there are a multitude of possibilities to realize a braking maneuver with the same braking force. In [182] the slip λ_i and the slip angle α_i for each wheel $i = 1, ..., 4$ are the optimization parameters. The objectives are to avoid tire wear by minimizing the squared sum of the slip angles (f_1) and for each tire the minimization of the distance between the tire force and the adhesion limit for safety reasons ($f_2, ..., f_5$). Fig. 5.27 shows projections of the resulting Pareto set and Pareto front.

Another extension of both the recovering and subdivision algorithms is the use of parallelization techniques. This is motivated by time-consuming function evaluations of a sufficiently high amount of test points involved in the algorithms. In [26], for instance, a multiobjective optimization problem is solved for the resource efficient design of integrated circuits. In more detail, the dimensions of transistors in simple logic cells are optimized with respect to noise margin, propagation delay and dynamic energy consumption. A function evaluation in this setup is a

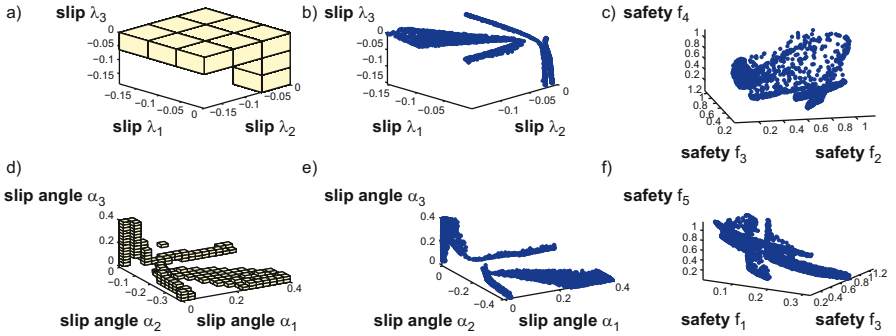

Fig. 5.27 Projection of the resulting Pareto set and Pareto front of a multiobjective optimization problem of the distribution of the tire forces for a braking maneuver of the test vehicle Chameleon. The Pareto set and Pareto front were computed with the algorithmic differentiation approach: (a), (d) A set of boxes covering smooth connected parts of the Pareto set. (b), (e) Corresponding Pareto points. (c), (f) Corresponding Pareto front. Figure from [182].

one to three seconds simulation of an integrated circuit. Using a parallelization infrastructure, it is possible to obtain good approximations of the Pareto set within adequate computational time.

5.3.1.3 Set-Oriented Recovering Methods in the Image Space Applied to the Multiobjective Optimization of the Active Guidance System of the RailCab

The previously described recovering method reaches its limitations if the number of design variables is high. In such a case, the approximation of the tangent space in the predictor step is computationally costly, therefore the recovering method will be applied in the image space Image space (cf. [41]). The principal procedure is the same as shown in Fig. 5.26. This approach is a good alternative for the case when the dimension of the parameter set is high and only a few objectives are considered.

This method was applied to find trajectories of the RailCab vehicle (cf. Sect. 2.1) in the rails [72, 206]. The control of the RailCab vehicle is done by the active guidance system that controls the displacement of the vehicle in the rails. It controls the position of the front and rear axles. The computed trajectories should maximize safety (f_1) and passenger comfort (f_2) and minimize the average energy consumption (f_3) of the hydraulic actuators. Naturally, this problem is an optimal control problem but due to the fact that the problem underlies a certain structure it can be transformed into a multiobjective optimization problem with a high number of parameters (cf. Sect. 1.4.1.2 and 5.3.6). In [72] the trajectories of the front and rear axles of a fixed rail track with respect to f_1, f_2 and f_3 have been optimized. Due to the high amount of parameters, a recovering method in the image space Image space is necessary. In Fig. 5.28 the computed Pareto front is shown. Two optimal compromise solutions were selected (marked by a circle and a rectangle). In Fig. 5.29 the

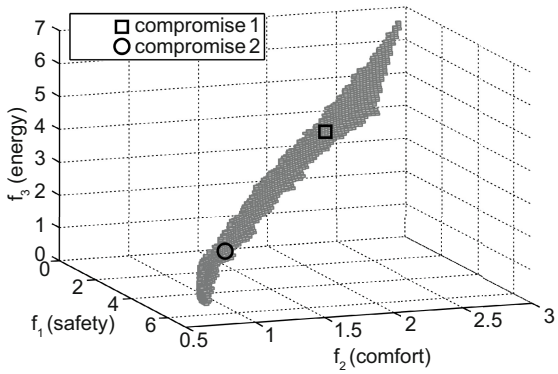

Fig. 5.28 Pareto front for the three objectives safety, comfort and energy. The trajectories corresponding to the Pareto points marked with a rectangle and a circle are shown in Fig. 5.29. Figure from [72].

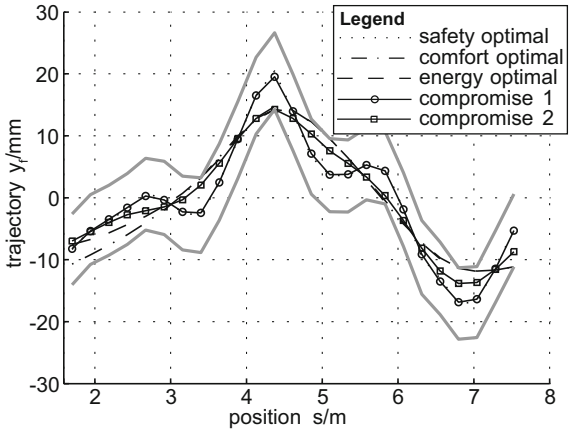

Fig. 5.29 Different reference trajectories for the front axle. While comfort prioritizing trajectories are apparently smooth, safety prioritizing trajectories try to follow vertical displacement to stay near the middle line. Figure from [72].

corresponding trajectories of the position of the front axle are shown. The recovering techniques in the image space are also suitable to find well-distributed Pareto points in the image space. In [183] such a method was applied to design an operating strategy for the Energy Management of a Hybrid Energy Storage System combining batteries and double layer capacitors.

To sum up, various applications have shown the great suitability of the set-oriented mutliobjective optimization methods in the design of self-optimizing technical systems.

5.3.2 Hierarchical Multiobjective Optimization

Michael Dellnitz and Maik Ringkamp

Modeling of self-optimizing systems often leads to hierarchical multiobjective optimization problems. These kinds of problems consist of several MOPs instead of

just one MOP. All MOPs are related to each other by a hierarchy. The solutions of a lower level MOP restrict the preimage space of the next higher level MOP in the sense that the feasible set of the higher level MOP is a subset of the lower level Pareto set. Each level of the hierarchy consists of one MOP.

Consequently, in the case of two MOPs two levels of hierarchy exist. Such a problem is also called bilevel MOP and is defined as follows:

$$\begin{aligned} \min_{(\mathbf{p},\mathbf{p}^1)\in\mathbb{R}^n\times\mathbb{R}^{n_1}} \quad & \mathbf{F}(\mathbf{p},\mathbf{p}^1) \\ s.t. \quad & (\mathbf{p},\mathbf{p}^1)\in S \\ & \mathbf{p}^1 \in \mathscr{P}_{\mathbf{f}^1}(\mathbf{p}) \end{aligned} \tag{5.2}$$

Here, the $\mathbf{p}$-dependent Pareto set $\mathscr{P}_{\mathbf{f}^1}(\mathbf{p})$ is defined as the solution of the MOP of the lower level:

$$\mathscr{P}_{\mathbf{f}^1}(\mathbf{p}) := \arg\min_{\mathbf{p}^1\in\mathbb{R}^{n_1}} \quad \mathbf{f}^1(\mathbf{p},\mathbf{p}^1) \tag{5.3}$$

$$s.t. \quad (\mathbf{p},\mathbf{p}^1)\in S^1 \tag{5.4}$$

with feasible sets $S, S^1 \in \mathbb{R}^n \times \mathbb{R}^{n_1}$, objective functions $\mathbf{F} : \mathbb{R}^n \times \mathbb{R}^{n_1} \to \mathbb{R}^k$, and $\mathbf{f}^1 : \mathbb{R}^n \times \mathbb{R}^{n_1} \to \mathbb{R}^{k_1}$.

Under given regularity conditions, bilevel MOPs can be solved using the Karush-Kuhn-Tucker equations (Eqs. (1.2)) of the lower level MOP as additional equality constraints for the upper level MOP as described in detail in [42].

The hierarchical structure of the optimization problems derived from the OCM structure allows to consider a special case of the general bilevel MOP (5.2). Instead of computing one general MOP on the lower level, we consider problems where the lower level MOP can be separated into several independent MOPs, i.e. each MOP has a different set of optimization parameters.

5.3.2.1 Hierarchical Multiobjective Optimization by Parametrization of the Lower Levels

More specifically, the kind of problems we consider are given as

$$\begin{aligned} \min_{(\mathbf{p},\mathbf{p}^1,\ldots,\mathbf{p}^l)\in S} \quad & \mathbf{F}(\mathbf{p},\mathbf{p}^1,\ldots,\mathbf{p}^l) \\ s.t. \quad & \mathbf{p}^j \in \mathscr{P}_{\mathbf{f}^j}, j \in \{1,\ldots,l\}, \end{aligned} \tag{5.5}$$

where $l \geq 1$ is the number of independent lower level MOPs and $S \subseteq \mathbb{R}^n \times \mathbb{R}^{n_1} \times \ldots \times \mathbb{R}^{n_l}$ the feasible set as in Eq. (5.2) with independent Pareto sets $\mathscr{P}_{\mathbf{f}^j}, j \in \{1,\ldots,l\}$, as solutions of the l lower level MOPs

$$\mathscr{P}_{\mathbf{f}^j} := \arg\min_{\mathbf{p}^j\in\mathbb{R}^{n_j}} \mathbf{f}^j(\mathbf{p}^j) \tag{5.6}$$

with objective functions $\mathbf{F} : \mathbb{R}^n \times \mathbb{R}^{n_1} \times \ldots \times \mathbb{R}^{n_l} \to \mathbb{R}^k$ and $\mathbf{f}^j : \mathbb{R}^{n_j} \to \mathbb{R}^2, \forall j \in \{1, \ldots, l\}$.

Under certain regularity conditions, the resulting Pareto sets $\mathscr{P}_{\mathbf{f}^j}$ of the lower level MOPs are 1-dimensional submanifolds of $\mathbb{R}^{n_j}$ for each $j \in \{1, \ldots, l\}$. Thus, these sets can be parametrized by variables $\alpha^j \in [0, \alpha_{max}]$ and a map $\varphi^j : [0, \alpha_{max}] \to \mathscr{P}_{\mathbf{f}^j}$. The parametrization reduces the complexity of the upper level MOP, it can be described with the help of an auxiliary objective $\tilde{\mathbf{F}} : \mathbb{R} \times [0, \alpha_{max}] \times \ldots \times [0, \alpha_{max}] \to \mathbb{R}^k, \tilde{\mathbf{F}}(\mathbf{p}, \alpha^1, \ldots, \alpha^l) := \mathbf{F}(\mathbf{p}, \varphi^1(\alpha^1), \ldots, \varphi^l(\alpha^l))$ as

$$\begin{aligned} \min_{(\mathbf{p}, \alpha^1, \ldots, \alpha^l) \in \mathbb{R}^n \times [0, \alpha_{max}] \times \ldots \times [0, \alpha_{max}]} \quad & \tilde{\mathbf{F}}(\mathbf{p}, \alpha^1, \ldots, \alpha^l) \\ s.t. \qquad (\mathbf{p}, \alpha^1, \ldots, \alpha^l) \in S. & \end{aligned} \tag{5.7}$$

For problem (5.5) we propose the following solution strategy:

1. Compute the Pareto sets of all independent lower level MOPs (5.6) by using the methods explained in Sect. 5.3.1.
2. Parametrize the resulting Pareto sets by the map φ.
3. Use the parametrization variables as parameters for the MOP on the next higher level and solve the auxiliary problem (5.7).

This method was successfully applied for example in [131] or [102] to solve bilevel MOPs derived by the OCM structure. In the latter work, the considered application examples are an active suspension system and a linear drive with an active air gap adjustment which both represent a module of the rail-bound vehicle RailCab. Hierarchical optimization is used to combine the module-related optimal operating strategies. In Fig. 5.30 the computed Pareto front of the upper level MOP is shown.

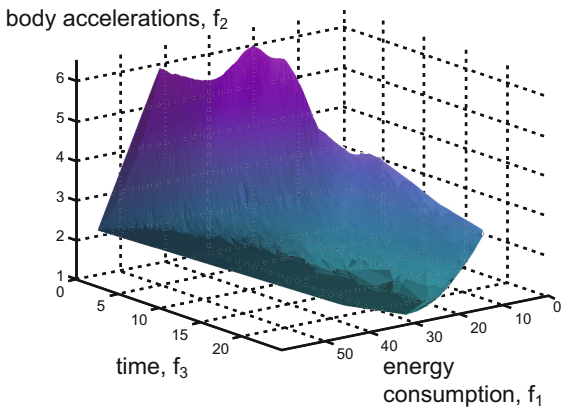

Fig. 5.30 *Active suspension system and linear drive:* Computed Pareto front for the hierarchical model of the combination of the active suspension system and the linear drive with an active air gap adjustment (original figure from [102]).

5.3.3 Hierarchical Modeling of Mechatronic Systems

Martin Krüger and Ansgar Trächtler

The hierarchical modeling is based on the hierarchical OCM structure presented in Sec. 1.3. Mathematical models of the dynamical behavior are needed for several methods in the design process of self-optimizing systems. Such methods are for example, the design of feed-forward or feedback controllers, identification and observation of system parameters respective states or model-based optimization. Complexity of the models rapidly increases at higher levels of the system hierarchy. The modeling approach described in the following sections yields a so-called hierarchical model which uses the hierarchical structure to reduce the model complexity in a systematic way. Particularly, in combination with hierarchical multiobjective optimization (cf. 5.3.2) a novel approach for parametric model-order reduction can be used.

5.3.3.1 Hierarchical Model

Each element of the system hierarchy is equipped with its own information processing described by an OCM. In general, this reduces the complexity of the information processing, as several tasks can be encapsulated. However, the dynamical behavior of a subsystem depends on the underlying elements (subsystems) in the hierarchy. Hence, the behavior of the underlying subsystems has to be taken into account in the modeling process.

The idea of the hierarchical model is to include the dynamics of the underlying systems in a simplified form, rather than considering all details. This reduces the complexity of the resulting model while ensuring that models have an appropriate amount of detail that can be used by model-based methods. Figure 5.31 illustrates the general idea.

Additionally, if each element of the hierarchy is seen as a self-optimizing system with its own objectives, a Pareto set, i.e. a set of optimal compromises, can be computed by applying multiobjective optimization. This Pareto set can then be used as additional information for the simplification of the system before it is transferred to the superordinated element. The parametric model-order reduction approach described below has been developed especially for this task. The result is a simplified respective reduced model which can, for example, be simulated much faster than the original model while maintaining a certain variability in view of the objectives.

5.3.3.2 Parametric Model-Order Reduction

In the following we will give a short overview about a particular parametric model-order reduction approach which yields parametric reduced models for the Pareto-optimal systems that was first published in [129]. The Pareto-optimal systems are those that correspond to the Pareto-optimal parameters $p^\star \in \mathscr{P}_F$. The general goal is to construct an approximation of these Pareto-optimal systems in terms of the parameterization variable α, limited to the case of two objective functions. This

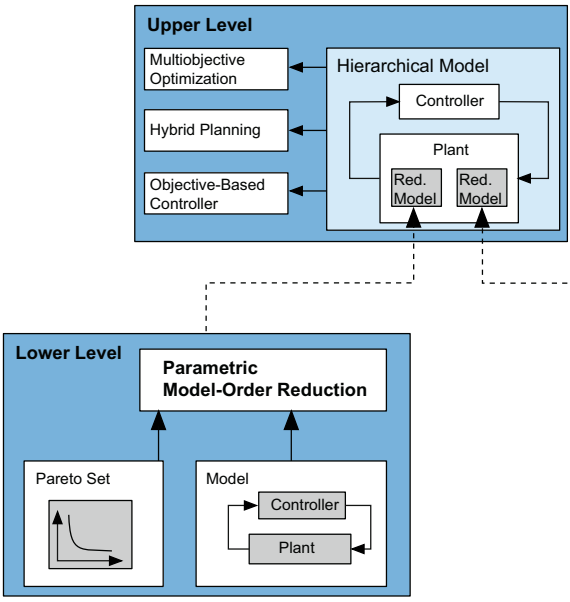

Fig. 5.31 Hierarchical modeling principle for self-optimizing systems. First, the optimal configurations of the lower level module are computed. The resulting Pareto set is parameterized and the system model corresponding to the optimal configurations is reduced by parametric model-order reduction. On the upper level the reduced models are integrated in the hierarchical model which can then be used for following tasks as e.g. (hierarchical) optimization (Sect. 5.3.2) hybrid planning (Sect. 5.3.8) or the design of an objective-based controller (Sect. 2.1.4).

kind of model-order reduction can also be beneficial for analyzing the objective-based controller described in Sect. 2.1.4 where α is the control variable.

Interpolation of Pareto-Optimal Systems

The parameterization of the Pareto set described in Sect. 5.3.2 also defines a parameterization of the Pareto-optimal systems. Assuming a linear closed-loop system

$$\dot{\mathbf{x}} = A(\varphi(\alpha))\mathbf{x} + B\mathbf{u}, \tag{5.8a}$$

$$\mathbf{y} = C\mathbf{x}, \tag{5.8b}$$

with $\mathbf{u}$ being the vector of external inputs and $\mathbf{y}$ being the output vector for calculating the objectives, the dynamics depend on the parameterization function φ. Since a higher number of parameters complicates the reduction process for almost all parametric model-order reduction algorithms we do not directly use this kind of parameterization. Instead, we create an interpolation, of the Pareto-optimal systems and not of the Pareto set, that depends directly on α.

The first step is to define a sequence of knots $0 = \alpha_1 < \alpha_2 < \ldots < \alpha_k = \alpha_{max}$. Then, a component-wise linear spline interpolation can be applied to the Pareto-optimal systems that yields

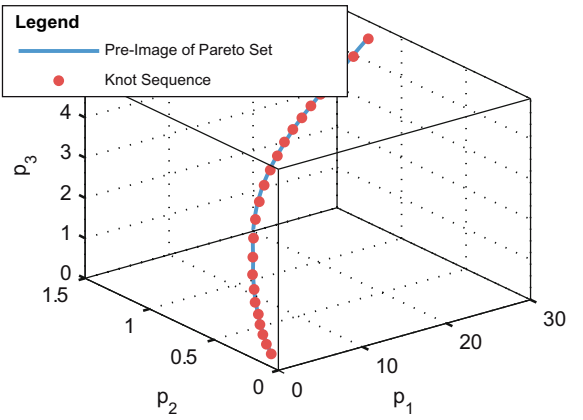

Fig. 5.32 Pareto set of the active suspension system and results of the knot placement. Optimization parameters are given by three variables, which define the sky-hook damping of the system. A number of ten equidistantly placed knots has been used as input for the algorithm leading to a knot sequence of 24 knots placed along the Pareto set to reach the given error bound.

$$A(\alpha) := \underbrace{A(\varphi(\alpha_i))}_{A_i} + \frac{\alpha - \alpha_i}{\alpha_{i+1} - \alpha_i} \left[A(\varphi(\alpha_{i+1})) - A(\varphi(\alpha_i)) \right] \tag{5.9}$$

for $\alpha \in [\alpha_i, \alpha_{i+1})$.

The number of knots α_i as well as their positions can be chosen automatically by an algorithm that is described in more detail in [129]. It consists of two parts. One part improves the knot positions of an existing sequence by means of the classical FORTRAN algorithm *newnot* [30] that has been extended to the matrix case. The second part compares the linear matrix-valued spline with a cubic one to estimate the approximation quality and inserts additional knots if necessary. Both parts are executed alternately until a given error bound is reached. Figure 5.32 shows the results of the knot placement for a Pareto set of the active suspension system, introduced in Sect. 2.1.4 using the same objectives energy consumption and level of comfort.

Parametric Model-Order Reduction

The result of the aforementioned interpolation is a piecewise matrix polynomial $A(\alpha)$ and a corresponding parametric system

$$\dot{\mathbf{x}} = A(\alpha)\mathbf{x} + B\mathbf{u}, \tag{5.10a}$$

$$\mathbf{y} = C\mathbf{x}, \tag{5.10b}$$

with the states $\mathbf{x} \in \mathbb{R}^{n_x}$ and system matrices $A(\alpha), B$ and C of appropriate dimensions that can be reduced by parametric model-order reduction. The first step of the reduction procedure comprises of a non-parametric reduction of the systems corresponding to the knots α_i. Any projection-based reduction method that yields two projection matrices $V_i, W_i \in \mathbb{R}^{n_x \times q}$, can be used for this task, e.g. the IRKA (Iterative Rational Krylov Algorithm) to get an $\mathscr{H}_2$-optimal interpolation [10]. This leads to the reduced systems of order q

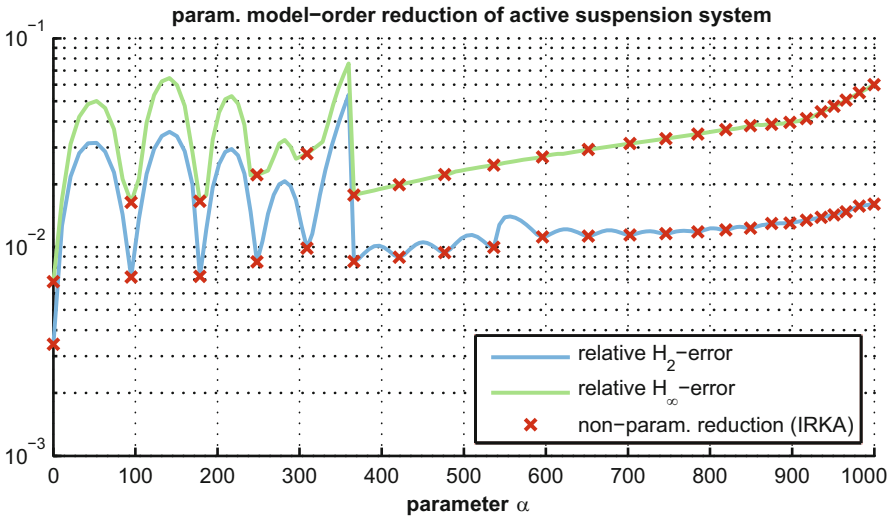

Fig. 5.33 Relative error of the reduced system compared to the original system (5.10).

$$\underbrace{W_i^T V_i}_{\tilde{E}_{r,i}} \dot{\mathbf{x}}_r = \underbrace{W_i^T A_i V_i}_{\tilde{A}_{r,i}} \mathbf{x}_r + \underbrace{W_i^T B}_{\tilde{B}_{r,i}} \mathbf{u}, \tag{5.11a}$$

$$\mathbf{y} = \underbrace{C V_i}_{\tilde{C}_{r,i}} \mathbf{x}_r \quad (1 \le i \le k). \tag{5.11b}$$

Secondly, we apply a method called matrix interpolation to compute a parametric reduced system, see [160] for more details. Using Matrix Interpolation, the reduced matrices are compatible to one another by means of a reprojection to a common subspace, given by the columns of an orthonormal matrix $R \in \mathbb{R}^{n_x \times q}$. This matrix is computed by means of a singular value decomposition of the concatenation of the projection matrices $[V_1, \ldots, V_k]$. Each reduced system is then transformed by means of two quadratic matrices

$$M_i = (W_i^T R)^{-1} \text{ and } T_i = R^T V_i. \tag{5.12}$$

The parametric reduced system consists of an interpolation of the transformed reduced matrices

$$E_{r,i} = M_i \tilde{E}_{r,i} T_i,\ A_{r,i} = M_i \tilde{A}_{r,i} T_i,\ B_{r,i} = M_i \tilde{B}_{r,i},\ C_{r,i} = \tilde{C}_{r,i} T_i \tag{5.13}$$

In our case we use a simple weighted sum depending on α, i.e.

$$A_r(\alpha) = \left(1 - \frac{\alpha - \alpha_i}{\alpha_{i+1} - \alpha_i}\right) A_{r,i} + \frac{\alpha - \alpha_i}{\alpha_{i+1} - \alpha_i} A_{r,i+1} \tag{5.14}$$

for the system matrix to give one example. The results of the parametric model-order reduction of the active suspension system are shown in Fig. 5.33.

5.3.4 Parametric Multiobjective Optimization

Michael Dellnitz, Christian Horenkamp, and Katrin Witting

Many mechatronic systems are subject to external forces or time-varying parameters. In many cases, such dependencies cannot be directly modeled in the optimization problem (5.1) in Sect. 5.3.1. Therefore, in this section, we extend the optimization problem (5.1) in such a way that it additionally depends on an external parameter $\lambda \in [\lambda_{start}, \lambda_{end}]$:

$$\min_{\mathbf{p} \in S} \mathbf{F}(\mathbf{p}, \lambda), \tag{5.15}$$

where $\mathbf{F} : \mathbb{R}^n \times [\lambda_{start}, \lambda_{end}] \to \mathbb{R}^k$, $\mathbf{F}(\mathbf{p}, \lambda) = (f_1(\mathbf{p}, \lambda), ..., f_k(\mathbf{p}, \lambda))^T$ is the vector of objective functions. The parameter λ can model the dependence on time or any other external parameter of the objectives. In such situations, instead of choosing a single Pareto point, the decision maker has to choose a whole curve $\mathbf{p}(\lambda)$ describing for each λ a Pareto optimal solution for the MOP. Similar as in Sect. 5.3.1, for each fixed $\lambda \in [\lambda_{start}, \lambda_{end}]$ the necessary optimality conditions are given by the Karush-Kuhn-Tucker equations, and the underlying optimization problem can be solved separately for each parameter value.

Consider the following parameter dependent MOP with $\lambda \in [0,1]$ and the two objective functions $f_1, f_2 : \mathbb{R}^2 \times [0,1] \to \mathbb{R}$ defined as

$$f_1(\mathbf{p}, \lambda) = \lambda \left((p_1 - 2)^2 + (p_2 - 2)^2\right) + (1 - \lambda) \left((p_1 + 2)^4 + (p_2 - 2)^8\right) \text{ and}$$
$$f_2(\mathbf{p}, \lambda) = (p_1 + 2\lambda)^2 + (p_2 + 2\lambda)^2.$$

Fig. 5.34 (a) shows the Pareto sets for different values of λ and Fig. 5.34 (b) shows the entire λ-dependent Pareto set.

Calculating for each parameter value the entire Pareto set is numerically very costly and therefore, this approach is not suitable for applications, for which the solution has to be computed online. Thus, we propose a solution method which alternates between Pareto set computations and numerical path following of single Pareto points and therefore prevent the computation of the entire Pareto set. The proposed algorithm is designed for online use and works as follows:

1. Compute the entire Pareto set for a fixed parameter value λ_1 and select a point $\mathbf{p}(\lambda_1)$ on the Pareto set.
2. Compute the solution curve $\mathbf{p} : [\lambda_1, \lambda_2] \to S$ up to a fixed parameter value λ_2.
3. Compute the entire Pareto set for the parameter value λ_2 and select a point $x(\lambda_2)$ on the Pareto set. Proceed with step 2.

For the computation of the solution curve, in step 2 a predictor corrector method along the curve direction is involved.

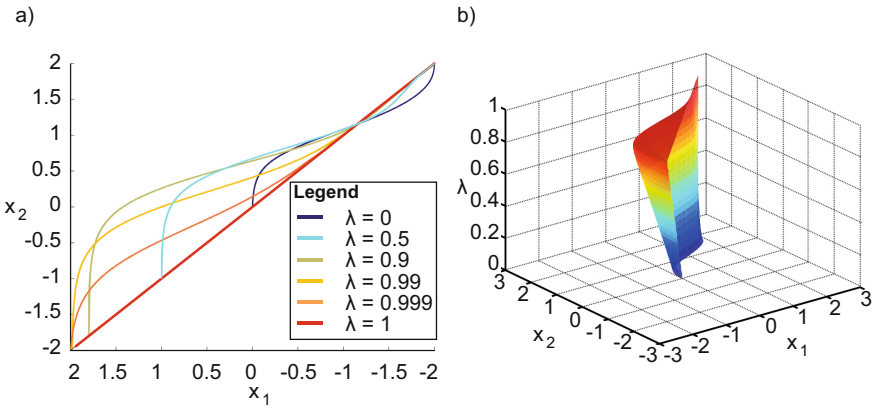

Fig. 5.34 (a) Pareto sets for some specified values of λ. (b) entire λ dependent Pareto set. Figure from [206].

The parameter dependent approximation of the Pareto optimal solutions was developed in [206]. In [208] and [187], it has been successfully applied to the optimization of the operating point assignment of the linear-motor of the driven railway system RailCab (cf. 2.1). It was also successfully applied to the active suspension system of the RailCab. In this application the crosswind has an influence and it was modeled as a parameter (see also Sec. 5.3.5.1).

5.3.5 Computation of Robust Pareto Points

Michael Dellnitz, Robert Timmermann, and Katrin Witting

One important question in the context of multiobjective optimization problems (cf. Sect. 1.4.1.1) is the choice of the actual optimal configuration for one specific application, the so-called **decision making**. In this section we address this problem by defining the **robust Pareto points** and give a brief overview of two methods for the computation of such points. For a more detailed explanation, the reader is referred to [206]. We consider a Pareto point to be robust, if it varies as little as possible under variation of the external parameters of the parametric multiobjective optimization problem Eq. (5.15). Here, we additionally have the choice to regard the variation in parameter space or objective space.

Computation is based on two approaches: The first approach to the computation of robust Pareto points is based on numerical path following methods (cf. Sect. 5.3.4). First, a λ-dependent Pareto set for $\lambda = \lambda_{start}$ is computed. Secondly, λ is varied from λ_{start} to λ_{end} for a subset of points of the Pareto set and the lengths of the resulting paths, which then run from the λ_{start}-Pareto set to the λ_{end}-Pareto set, are calculated. Finally, these path lengths can be taken into account when choosing one of the Pareto optimal operating points, since robust Pareto points are those with minimal path length. This enables the decision maker to choose points, which vary

as little as possible under the influence of λ. If λ, for example, describes the influence of a change of temperature, one can chose an operating point, such that varying temperature has little effect on the system.

The second approach is based on the calculus of variations. The problem of finding the shortest path from a point on the Pareto set for λ_{start} onto the Pareto set for λ_{end} can be formulated as the variational problem

$$\min_{(\mathbf{p}(\lambda),\alpha(\lambda))} \int_{\lambda_{start}}^{\lambda_{end}} \|\mathbf{p}'(\lambda)\|_2^2 \, d\lambda \qquad (5.16)$$
$$s.t.\ \mathbf{H}_{KT}(\mathbf{p}(\lambda),\alpha(\lambda),\lambda) = 0$$

where the constraint $\mathbf{H}_{KT} = 0$ represents the necessary Kuhn-Tucker equations for optimality (cf. Eq. (1.2)) in Sect. 1.4.1.1 and Sect. 5.3.1). The integral means, that the energy of the λ-dependent curve of Kuhn-Tucker points is minimized. If points exist in which all Pareto sets intersect, both approaches lead to the same robust Pareto points. Otherwise those points may differ. The main advantage of the second concept over the first one is that the starting point on the Pareto set needs not to be fixed in advance but is implicitly calculated during the minimization. Unfortunately, this concept is computationally more expensive, so if the underlying models are very complex or if execution time is critical (e.g. if the robust points are calculated in real time), the first concept is more suitable.

A much more detailed explanation of the path following approach can be found in [47], and two applications are presented in [26] (transistor sizing of CMOS logic standard cells) and [201] (robust Pareto points for the Active Suspension Module). For further reading about the variational method we refer to [207]. Both methods are also presented in [69, D.o.S.O.M.S. Sect. 3.1.8] .

5.3.5.1 Application

The second concept has been successfully used to compute robust Pareto points for the Active Suspension Module (ASM, cf. Sect. 2.1.4) in [130].

In this work, an external parameter λ is used to model varying crosswind conditions which affect the ASM's behavior. A parametric multiobjective optimization problem was formulated using a simple ASM model with three degrees of freedom p_1, p_2, p_3 and with the two objectives comfort and energy consumption.

Figure 5.35 shows three Pareto sets for three different crosswind values and two robust Pareto points which were computed using the variational method. The robust point at $(0,0,0)$ corresponds to the energy optimal solution and could be expected in advance, the second point is nontrivial though and was not expected before the calculations. It can be used when designing the system such that it exhibits similar behavior in a variety of crosswind situations.

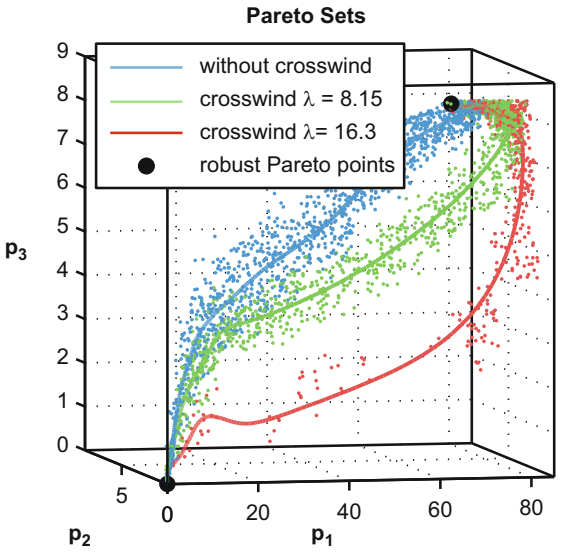

Fig. 5.35 Application of the second concept (based on the calculus of variations) to compute robust Pareto points for the Active Suspension Module. This figure shows Pareto sets for three specific crosswind values and two robust Pareto points. Figure from [130].

5.3.6 Optimal Control of Mechanical and Mechatronic Systems

Kathrin Flaßkamp and Sina Ober-Blöbaum

As introduced in Sect. 1.4.1, an optimal control problem seeks a control trajectory which steers the dynamical system in an optimal way with respect to a given cost functional. This is a challenging task for complicated nonlinear dynamical systems and thus has to be addressed by numerical techniques. In this section, we present an optimal control technique which is especially developed for the optimal control of mechanical systems (including mechatronic systems with additional electronic subsystems). For this class of systems, the equations of motion in the optimal control problem (OCP), cf. Eq. (1.3b), can be specified to the forced Euler-Lagrange equations, i.e.

$$\min_{\mathbf{x}(t),\mathbf{u}(t)} J(\mathbf{x},\mathbf{u}) = \int_0^T C(\mathbf{x}(t),\mathbf{u}(t))\,dt \tag{5.17a}$$

$$\text{with respect to } \frac{\partial L}{\partial \mathbf{q}}(\mathbf{q},\dot{\mathbf{q}}) - \frac{d}{dt}\frac{\partial L}{\partial \dot{\mathbf{q}}}(\mathbf{q},\dot{\mathbf{q}}) + \mathbf{f}(\mathbf{q},\dot{\mathbf{q}},\mathbf{u}) = 0 \tag{5.17b}$$

$$\mathbf{r}(\mathbf{x}(0),\mathbf{x}(T)) = 0, \text{ and} \tag{5.17c}$$

$$\mathbf{h}(\mathbf{x}(t),\mathbf{u}(t)) \leq 0 \text{ with } \mathbf{x} = (\mathbf{q},\dot{\mathbf{q}}). \tag{5.17d}$$

Here, the system's state $\mathbf{x} = (\mathbf{q},\dot{\mathbf{q}})$ consists of configurations $\mathbf{q}$ and corresponding velocities $\dot{\mathbf{q}}$, $L(\mathbf{q},\dot{\mathbf{q}})$ is the Lagrangian of the system (closely related to the system's

energy) and $\mathbf{f}$ a control dependent forcing[12]. All possible configurations of a system form the configuration manifold[13] Q such that the system's state space is given by the tangent bundle TQ.

To numerically solve an OCP, direct optimal control methods directly discretize the differential equations (5.17b). This can be done by integration schemes, i.e. the continuous state $\mathbf{x}(t)$ is replaced by a sequence of discrete states $\{\mathbf{x}_d\}$ in the same manner as discretized trajectories are generated by numerical integration (simulation) of dynamical systems. An optimal solution has to fulfill the discretized differential equations (and additional constraints) and it is optimal with respect to the discretized cost functional, i.e. it is a solution to a nonlinear optimization problem and approximates the solution of the original OCP.

5.3.6.1 The Direct Optimal Control Technique DMOC

DMOC (*Discrete Mechanics and Optimal Control*, [151]) is a direct optimal control method tailored to the special structure of mechanical systems . The forced Euler-Lagrange equations (5.17b) are derived from a variational principle: the Lagrange-d'Alembert principle ([140]). DMOC is based on a direct discretization of the Lagrange-d'Alembert principle of the mechanical system. The goal of this discrete variational mechanics approach is to derive discrete approximations of the solutions of the forced Euler-Lagrange equations that inherit the same qualitative behavior as the continuous solution. For the discretization, the state space TQ is replaced by $Q \times Q$ and the discretization grid for the time interval $[0,T]$ is defined by $\Delta t = \{t_k = kh \,|\, k = 0,\dots,N\}$, $Nh = T$, where N is a positive integer and h is the step size. The path $\mathbf{q} : [0,T] \to Q$ is replaced by a discrete path $\mathbf{q}_d : \{t_k\}_{k=0}^N \to Q$, where $\mathbf{q}_k = \mathbf{q}_d(kh)$ is an approximation of $\mathbf{q}(kh)$ [141, 151]. Similarly, the control path $\mathbf{u} : [0,T] \to U$ is replaced by a discrete one. The discrete Lagrange-d'Alembert principle then leads to the **discrete forced Euler-Lagrange equations**

$$D_1 L_d(\mathbf{q}_k, \mathbf{q}_{k+1}) + D_2 L_d(\mathbf{q}_{k-1}, \mathbf{q}_k) + \mathbf{f}_k^- + \mathbf{f}_{k-1}^+ = 0 \tag{5.18}$$

for each $k = 1,\dots,N-1$, where D_i denotes the derivative w.r.t. the i-th argument. That means, solution curves of the differential equation (5.17b) can be approximated by discrete solution trajectories of the set of algebraic equations. In other words, for given control values u_k, equation (5.18) provides a time stepping scheme for the simulation of the mechanical system which is called a variational integrator (cf. [141]). Since these integrators, derived in a variational way, are structure-preserving, important properties of the continuous system are preserved (or change consistently with the applied forces), such as symplecticity or momentum maps induced by symmetries (e.g. the linear or angular momentum of a mechanical system). In addition,

[12] Confere e.g. [140] for a general introduction into the theory of mechanical systems, in particular regarding Lagrangian mechanics.

[13] Simply speaking, a manifold is a generalization of the vector space $\mathbb{R}^n$ including e.g. tori, but readers non-familiar with differential geometry can replace Q by $\mathbb{R}^n$ and TQ by $\mathbb{R}^{2n}$ in the following.

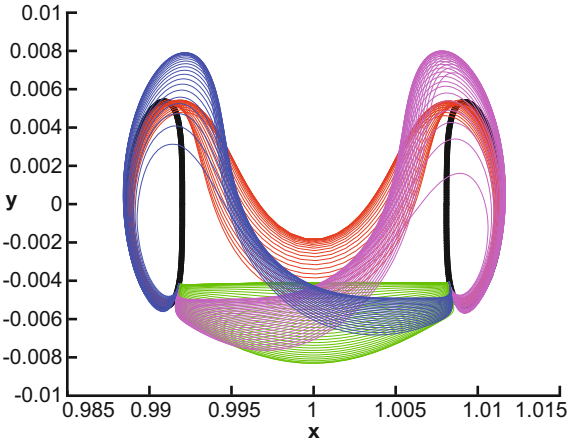

Fig. 5.36 *Space mission design:* Pareto optimal trajectories for minimal control effort and time-minimal transfer between period orbits near sun and earth. Red: high mission times, low control effort. Green: small mission times, high control effort. Blue and magenta: solution between the first two (Figure from [152]).

their long-time energy behavior is excellent. Therefore, variational integrators can be used with relatively large step sizes. However, rather than solving initial value problems, an optimal control problem has to be solved, which involves the minimization of a cost functional $J(\mathbf{x},\mathbf{u}) = \int_0^{t_f} C(\mathbf{x}(t),\mathbf{u}(t))\,dt$. Thus, in the same manner, an approximation of the cost functional generates the discrete cost functions C_d and J_d, respectively. The resulting nonlinear restricted optimization problem reads

$$\min_{\mathbf{q}_d,\mathbf{u}_d} J_d(\mathbf{q}_d,\mathbf{u}_d) = \min_{\mathbf{q}_d,\mathbf{u}_d} \sum_{k=0}^{N-1} C_d(\mathbf{q}_k,\mathbf{q}_{k+1},\mathbf{u}_k) \tag{5.19}$$

subject to the discrete forced Euler-Lagrange equations (5.18) together with discretized boundary and (in-)equality constraints for states and/or controls. Thus, the discrete forced Euler-Lagrange equations serve as equality constraints for the optimization problem which can be solved by standard optimization methods like SQP (cf. e.g. [76]). In [151], a detailed analysis of DMOC resulting from this discrete variational approach is given. The optimization scheme is symplectic-momentum consistent, i.e. the symplectic structure and the momentum maps corresponding to symmetry groups are consistent with the control forces for the discrete solution independent of the step size h. Thus, the use of DMOC leads to a reasonable approximation of the continuous solution, also for large step sizes, i.e. a small number of discretization points. Furthermore, constraints of mechanical systems can be included in DMOC such that it is applicable to constrained systems, which often occur in multi-body dynamics ([134]).

5.3.6.2 Extensions and Applications

Typically, in particular for self-optimizing systems, there is more than one single objective that has to be optimized, hence we are faced with **multiobjective optimal control** . Problems of this kind, i.e. with a vector $\mathbf{J}(\mathbf{x},\mathbf{u}) = (J_1(\mathbf{x},\mathbf{u}),\ldots,J_m(\mathbf{x},\mathbf{u}))$

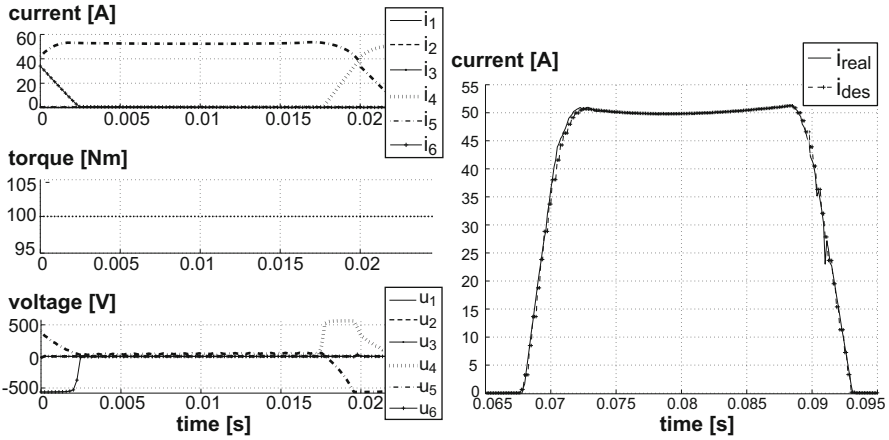

Fig. 5.37 *Switched reluctance drive:* Optimal profile for current and voltage computed by DMOC. Note that the constraint of a fixed motor torque is fulfilled for every discretization point. The profile is combined with a feedback controller. It can be followed at the real test bench very well as shown in the right plot (original Figure from [63])

of cost functionals can be solved by a combination of multiobjective optimization methods and optimal control techniques. Since the discretization of the differential equations, e.g. by DMOC as described above, leads to a high-dimensional multiobjective optimization problem (i.e. a high number of optimization parameters $\mathbf{q}_d, \mathbf{u}_d$), image space oriented methods should be applied. In [152], this method has been applied to an optimal control problem in space mission design, cf. Fig. 5.36. Here, the concurring objectives are the control effort and the transfer time, which should both be simultaneously minimized. Thus, the solution of the multiobjective optimal control problem results in a number of very different Pareto optimal trajectories. A mission designer would now choose one of the correspondent control trajectories dependent on current aims and restrictions on the mission for a thorough analysis and further optimization with more detailed models.

As proposed above, the DMOC method is not restricted to purely mechanical systems since many electrical (sub)systems can be modeled by Lagrangian functions as well . In the course of the CRC 614, the optimization of the Hybrid Energy Storage System (cf. Sect. 5.3.1 and Sect. 2.1.5) has been repeated with additional (final) constraints on the optimal control problem. Furthermore, DMOC has been successfully used for the optimal control of a switched reluctance drive (cf. [63] and Sect. 2.1.1 above for a description of the test bed). The optimal current profiles have to fulfill two aims: maximizing the efficiency of the engine and guaranteeing a constant torque of the drive. In this application, the torque restriction is modeled as an equality constraint and the resulting single objective optimization problem is solved by DMOC. The resulting feedforward control is combined with a feedback controller. In Fig. 5.37, results are shown from the successful application to the real test bed.

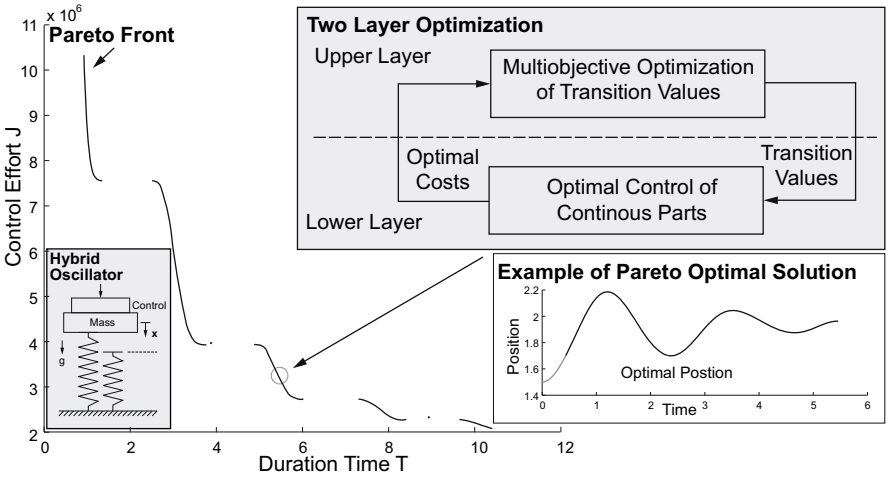

Fig. 5.38 *Hybrid single mass oscillator:* in the two layer optimization approach, a multiobjective optimization problem arises since both the control effort and the time of the maneuver (into the equilibrium position) have to be optimized. The resulting Pareto front is shown with an example solution for the resulting hybrid position trajectory (Subimages from [61])

Hybrid mechanical systems are described by continuous-time dynamics in combination with discrete events to model e.g. impacts, varying topologies of interacting robots, or a changing environment. From the perspective of optimal control, the switching times at which the discrete events occur, become new design variables. The **optimal control of hybrid systems** is an active field of research. Promising results can be achieved by approaches that split the problem up into several layers (cf. Fig. 5.38 and [61] for a detailed discussion). It is then possible to solve ordinary optimal control problems on a lower layer with well established methods while on the upper layer, the switching time optimization can be performed with other appropriate techniques. Figure 5.38 shows an example with a multiobjective optimization of a hybrid single mass oscillator, which has to be steered into its equilibrium position. Switching time optimization as a specific, isolated optimal control problem for hybrid systems has been studied in [60].

The optimal control method DMOC has been extended in several directions to improve performance and applicability even further. For the computation of gradients that are used for the optimization, e.g. by the SQP algorithm, DMOC can be combined with ADOL-C [153], a tool for algorithmic differentiation. In applications where subsystems with different time scales are interacting, the variational integrator can be extended to a multirate integration scheme [132, 133] that allows for an accurate integration with acceptable computational effort for combined fast and slow dynamical systems. The accuracy of numerical integration and thus of optimal solutions as well depend on the order of the approximation scheme. Therefore, higher order schemes can be used for the discrete Lagrangian [34].

Direct optimal control methods are based on local optimizers for the nonlinear optimization problem and therefore, they strongly rely on good initial guesses. Since minimal control effort is often a desired aim, it is a fruitful approach to use **inherent dynamical properties** of the uncontrolled system to generate such initial guesses. In space mission design, it has become state of the art to use trajectories on the system's invariant manifolds to design energy efficient control maneuvers (cf. e.g. [45, 146, 199] for applications using DMOC as the optimal control method). This approach can be used for technical systems as well, in [64] it is shown that, compared to black box optimizations with simple initial guesses, better (local) optima can be found with the help of initial guesses on the stable manifold of the final equilibrium position for a planar double pendulum. In more detail, this idea is explained in the broader context of motion planning with motion primitives in the following section.

5.3.7 Motion Planning with Motion Primitives

Kathrin Flaßkamp and Sina Ober-Blöbaum

Solving optimal control problems which arise in real applications is a challenging task for current numerical techniques. Since many optimal control techniques are based on local optimization methods, they strongly depend on good initial guesses to provide (local) optimal solutions which are also globally efficient. **Motion planning with motion primitives** – going back to [66] – tackles these difficulties with a two phase approach. In the first step, several short pieces of simply controlled trajectories are collected in a motion planning library, typically represented as a graph. These motion primitives can be sequenced to longer trajectories in various combinations. In the second phase, for a given optimal control problem, the optimal sequence of motion primitives is determined from the motion planning library. Such motion primitives originate from inherent **symmetries**, i.e. the dynamical system is equivariant with respect to certain transformations and certain system properties turn out to be invariant with respect to these symmetries[14]. Typically, mechanical systems naturally exhibit symmetries as translational or rotational invariance. By consequence, controlled maneuvers, that have been computed for a specific situation, are suitable in many different (equivalent) situations as well. Recently (cf. [62]), this motion planning technique has been extended by a new kind of primitives, namely trajectories on (un)stable manifolds of the natural system dynamics. In space mission design, such trajectories on invariant manifolds have already been successfully used (cf. e.g. [146]). This approach is especially tailored to the computation of energy efficient (minimal control effort) solutions, which is often a major objective for technical systems.

We formally introduce symmetry and motion primitives for Lagrangian systems (cf. 5.3.6), although the basic approach holds for general dynamical systems (cf. [66]). Assume that a Lie group G is acting on the configuration manifold Q by

[14] Again, we recommend [140] for an introduction to the role of symmetries in mechanical systems.

a so called left-action $\Phi : G \times Q \to Q$ ($\Phi(\mathbf{g},\cdot) =: \Phi_\mathbf{g}$ is a diffeomorphism for each $\mathbf{g} \in G$). It can be lifted to the tangent space: $\Phi^{TQ} : G \times TQ \to TQ$ for $(\mathbf{q},\mathbf{v}) \in TQ$ given by $\Phi_\mathbf{g}^{TQ}(\mathbf{q},\mathbf{v}) = T(\Phi_\mathbf{g}) \cdot (\mathbf{q},\mathbf{v})$. Then, symmetry corresponds to the invariance of the Lagrangian under the group action, i.e. $L \circ \Phi_\mathbf{g}^{TQ} = L$ for all $\mathbf{g} \in G$. In other words, two trajectories $\pi_1 : t \in [t_{i,1}, t_{f,1}] \mapsto (\mathbf{q}_1, \dot{\mathbf{q}}_1, \mathbf{u}_1)(t)$ and $\pi_2 : t \in [t_{i,2}, t_{f,2}] \mapsto (\mathbf{q}_2, \dot{\mathbf{q}}_2, \mathbf{u}_2)(t)$ are equivalent, if it holds that (1) $t_{f,1} - t_{i,1} = t_{f,2} - t_{i,2}$, both have the same time duration and (2) there exists $g \in G, T \in \mathbb{R}$, such that $(\mathbf{q}_1, \dot{\mathbf{q}}_1)(t) = \Phi_\mathbf{g}^{TQ}((\mathbf{q}_2, \dot{\mathbf{q}}_2)(t-T))$ and $\mathbf{u}_1(t) = \mathbf{u}_2(t-T) \forall t \in [t_{i,1}, t_{f,1}]$. All equivalent trajectories can be summed up in an equivalence class, the motion primitive. The number of candidates for the motion planning library can be immensely reduced by exploiting the system's invariance, i.e. only a single representative is stored that can be used at many different points when transformed by the lifted symmetry action. Induced by the symmetry, trim primitives are a special class of motion primitives. They are constantly controlled solutions which are generated solely by the symmetry action, i.e. $(\mathbf{q}, \dot{\mathbf{q}})(t) = \Phi^{TQ}(\exp(\xi\, t), (\mathbf{q}_0, \dot{\mathbf{q}}_0)), \mathbf{u}(t) = \mathbf{u}_0 = \text{const.} \forall t \in [0, T]$ with $\xi \in \mathfrak{g}$, the corresponding Lie algebra and $\exp : \mathfrak{g} \to G, \xi \mapsto \exp(\xi) \in G$ (cf. [140] for an introduction to mechanical systems and symmetry from a differential geometric perspective). Trim primitives can be found analytically or numerically based on the symmetry action. For mechanical systems, they are identical to relative equilibria and can be computed by symmetry reduction procedures (cf. [62]).

The second type of primitives, trajectories on (un)stable manifolds are computed for fixed points (or equilibria) $\bar{\mathbf{x}} = (\bar{\mathbf{q}}, 0)$ of the uncontrolled system. The local stable manifold for a neighborhood U of $\bar{\mathbf{x}}$ is defined as $W^s_{loc}(\bar{\mathbf{x}}) = \{\mathbf{x} \in U \,|\, \mathbf{F}_L(\mathbf{x},t) \to \bar{\mathbf{x}} \text{ as } t \to \infty \text{ and } \mathbf{F}_L(\mathbf{x},t) \in U \,\forall t \geq 0\}$. Then, the global stable manifold can be obtained by the union of the (pre)images of the Lagrangian flow $\mathbf{F}_L$. A stable manifold consists of all points in state space flowing towards the equilibrium. The corresponding trajectories are promising candidates for energy efficient steering maneuvers to operation points which are often the fixed points. The unstable manifold consists of all points that show the same behavior in backward time. Their existence is guaranteed by the stable manifold theorem [84]. In general, the (un)stable manifolds have to be computed numerically, e.g. by set-oriented methods [44].

As a third class of primitives, short controlled maneuvers between trims and manifold trajectories are required such that the primitives can be sequenced. They can be computed by DMOC (cf. Section 5.3.6) for example. The computed primitives are stored in a library. Then, for a specific control problem, i.e. with initial and final points on trims, e.g. operation modes of mechanical systems, in the library it is searched for the optimal sequence of primitives. This can be done based on the graph representation, the so called **maneuver automaton** (cf. [62, 66]). In principle, this second step could be even performed in real time, when using appropriate graph search methods.

We illustrate the approach for a spherical pendulum. Its Lagrangian is given by $L(\varphi, \dot{\theta}, \dot{\varphi}) = \frac{1}{2} mr^2(\dot{\varphi}^2 + \dot{\theta}^2 \sin^2(\varphi)) - mgr(\cos(\varphi) + 1)$ and we assume forcing in both directions. The system is symmetric with respect to rotations about the vertical axis. Trims are horizontal rotations with constant velocity. Contrarily, the (un)stable manifolds of the upper equilibrium are purely vertical motions. For an example

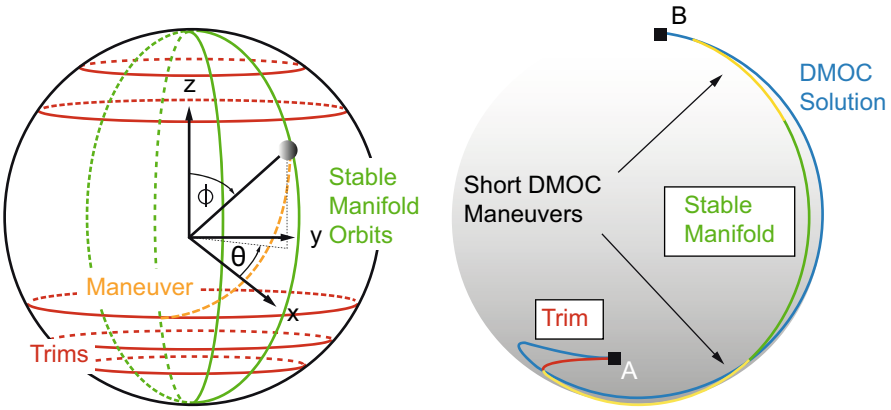

Fig. 5.39 *Sphercial pendulum:* in the motion planning with motion primitives approach, trim primitives are horizontal rotations while orbits on manifolds are purely vertical motions. An optimal sequence has been computed by DMOC and used for a post optimization ("DMOC solution", original subimages from [62])

control problem, the resulting optimal sequence is shown in Fig. 5.39 that consists of five motion primitives: the initial and final trim, two connecting maneuvers, and a trajectory on the stable manifold in between. The sequence has been used for a post-optimization by DMOC.

5.3.8 Hierarchical Hybrid Planning

Bernd Kleinjohann, Lisa Kleinjohann, and Christoph Rasche

Hybrid Planning [3] is based on hierarchical modeling presented in Sect. 5.3.3 and Pareto points calculated by a multiobjective optimization presented in Sect. 5.3.5. Taking the RailCab system (cf. Sect. 2.1) into account, several constraints have to be considered when moving a single RailCab from an initial position to a given goal position. One constraint is that the RailCab has only limited energy resources. To take such constraints concerning discrete as well as continuous system parameters into account, an overall plan for the movement of the RailCab must be computed. The term hybrid planning [2] denotes the integration of discrete and continuous domains in the planning approach. Initially a plan is created offline. It is updated continuously during the movement of the RailCab to ensure that, e. g. environmental influences, like wind do not lead to a violation of the given constraints. Hierarchical hybrid planning [56] denotes a planning approach, which does not only combine discrete and continuous planning but also considers the system's hierarchical decomposition into its single parts (cf. Sect. 5.3.3) during planning.

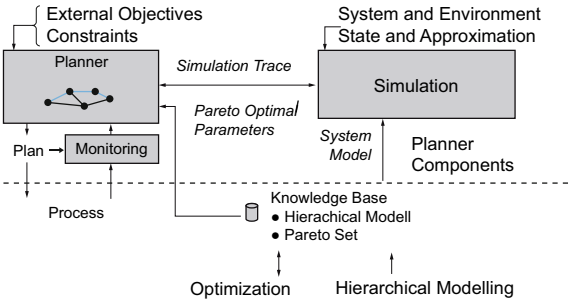

Fig. 5.40 Architecture of the hierarchical hybrid planning system

5.3.8.1 Principle

A plan is computed in order to ensure that a RailCab moves from its initial position to its destination while the requirements are taken into account by the planner. Different parts of a traveling route have diverse properties, like, e. g. slopes which have to be modeled. Thus, to actually create such a plan the complete route between the initial and the end position of the RailCab is subdivided into single track segments. In the first step an initial plan is computed consisting of different parameter settings for the single parts of the system for each track segment. These parameter settings build a discrete dimension of choice for the planner. In the case of the RailCab system considered here several objectives regarding for instance values like passenger comfort and energy consumption, which are in conflict have to be taken into account. For handling such conflictive objectives a multiobjective Pareto optimization is used. The Pareto optimization calculates a Pareto front determining optimal trade offs between parameter settings for each track section of the selected traveling route. Then, the offline planner selects a single Pareto point from the Pareto front for each section. Due to the actual system or environment conditions like wind, abrasion, etc. the forecasted results which selected parameter settings of the plan should lead to, might not be reached. Such deviations between the plan and the actual conditions are detected by continuously monitoring several values that determine the system state, allowing to initiate replanning by the online planner. Fig. 5.40 shows the components used to implement this approach.

As described in detail in [56] the planner is equipped with overall external objectives that need to be fulfilled at any time. In order to forecast the future development of continuous values determining the system state, the planner initiates a simulation with the actually measured system and environment state for the considered actions. The result of the simulation is a number of continuous value traces that are evaluated according to the constraints and objectives. The constraints are used to rule out an action, e.g. if the maximum peak power is too high or the comfort value used by the system is too poor during the simulation. Otherwise the action is considered as possible alternative by the planner. To finally decide for a possible alternative it is further evaluated with respect to the external objectives, for instance regarding the mean comfort or energy consumption of the overall section.

5.3.8.2 Methodology

One important issue of a hierarchical hybrid planner is the computation of a multilevel hierarchical configuration of system parameters during system operation. This configuration is used to improve the offline plan to take additional constraints or external objectives into account when different environment conditions or track segment properties must be considered.

The planner takes its input from the hierarchical optimization (cf. Sect. 5.3.2) and the hierarchical modeling (cf. Sect. 5.3.3). The results of these components are abstract models of the system parts and a set of Pareto-optimal parameter settings. As these components are designed in a hierarchical fashion, the outputs are precalculated on different hierarchical levels.

As an example for illustrating the methodology the Active Suspension Module of the RailCab (cf. Sect. 2.1.4) may serve. The active suspension system of the RailCab can be partitioned and structured hierarchically according to the function of each module. The hierarchy consists of two levels. On the upper level, the entire system which is in charge of the active suspension is considered. Beneath, on the lower level, there are two actuator groups realizing the active suspension by ensuring correct deflections of the fiberglass reinforced polymer springs (cf. Sect. 2.1.4).

The planner computes a multilevel configuration of the parameters of the active suspension system for each track segment based on the inputs described before. If the constraints could not be met, the planner adjusts only the lower level settings to reach the current goals, without affecting the upper level settings. Different objectives can be handled by different hierarchical levels.

During the movement of a RailCab, which executes a given plan, a monitoring of the current system behavior by measuring values like energy consumption and given comfort takes place. The measured data is compared to the data, which were taken into account to compute the initial plan. If the difference between this data is too high, a replanning is necessary. Hence, during system movement alternative Pareto-optimal parameter configurations have to be selected, which take into account these deviations. For this purpose the simulation component is used to predict the system behavior resulting from alternative parameter settings for the next track sections. These settings build a discrete dimension of choice for the planner and can be used for a predictive planning of the next track sections.

5.3.8.3 Application and Evaluation

The approach was evaluated using the Active Suspension Module (cf. Sect. 2.1.4), which is a part of the RailCab. The values for energy and comfort are abstract values without units of measurement. The test track consists of seven track sections. An overall energy consumption with the value 10600 and a comfort constraint for each track section of 49 was given. The two constraints, energy consumption and comfort, are in conflict because a higher comfort leads to a higher energy consumption. Two different types of planning were compared. They also considered the influence of changing environmental conditions, in this case represented by varying crosswind

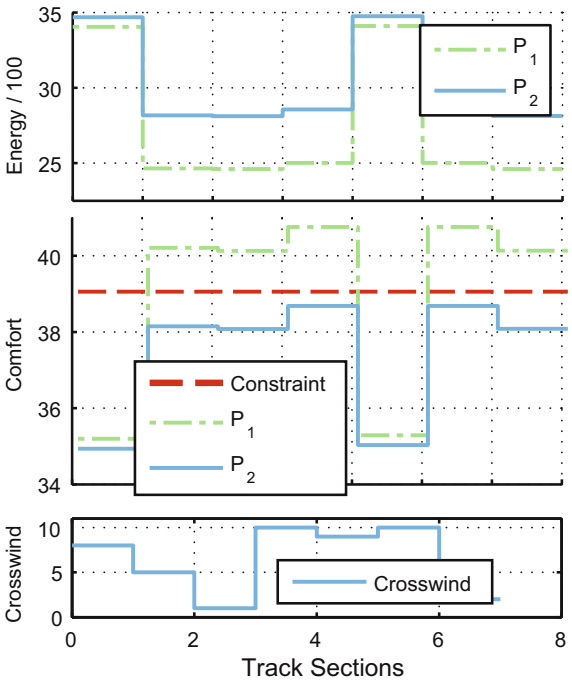

Fig. 5.41 Results of a test run. As higher the energy consumption is as higher is the value. A higher comfort is represented by a lower value.

settings. First, a hybrid planning was performed, which led to a resulting plan P_1. Thereafter, the hierarchical hybrid planning approach was executed using the same constraints on the same test track. While the non-hierarchical approach was not always able to reach the constraints, the hierarchical approach computed a plan P_2 in which each constraint was always reached with only small increases in energy consumption. A more detailed evaluation of the results is represented in [56].

The results in Fig. 5.41 show the energy values and comfort values as well as the crosswind settings on each track segment.

The results show that only the hierarchical approach was able to consistently meet the comfort constraints by finding a feasible plan. The non hierarchical planner violated the constraints at the track segments $2-4$ and $6-7$. The reason is, that the non-hierarchical planner did not have enough options to consider these constraints. The hierarchical planner can change the Pareto points influencing the lower level parts, i.e. the two actor groups realizing the active suspension as mentioned above. In contrast, the non-hierarchical planner can only change to another Pareto point for the overall system in order to satisfy the given constraints. This leads to different and sometimes inadmissible configurations. The concrete values selected by the planners P_1 and P_2 are shown in Table 5.2.

Table 5.2 shows that changes on the upper level have a much higher effect on the resulting values than changes on the lower level. Changes on the lower level also have an effect when both planners choose the same Pareto points for each track segment on the upper level. Only the hierarchical approach was able to meet the

constraints due to its ability to change the Pareto points influencing the lower level parts.

The results show that the hierarchical planning can use a more precise configuration to match the given constraints. Nevertheless, improvements regarding one parameter always imply impairments regarding other conflictive parameters, due to the Pareto-nature of the available planning alternatives.

Table 5.2 Energy and comfort values. Maximum of Total Energy 28.300. Comfort Constraint 39

	Energy		Comfort	
	P1	P2	P1	P2
section1	4812.4	4940.23	35.14	34.87
section2	2930.19	3634.37	40.15	38.09
section3	2920.29	3625.22	40.07	38.02
section4	3001.84	3714.25	40.7	38.62
section5	4823.7	4952.7	35.23	34.97
section6	3001.84	3714.25	40.7	38.62
section7	2920.83	3625.69	40.08	38.02
amount	24411.09	28206.71	272.07	261.21

5.3.9 Statistical Planning

Bernd Kleinjohann, Lisa Kleinjohann, and Christoph Rasche

Taking statistical data into account to compute plans for mechatronical systems results in a self-optimizing behavior. This is due to the fact that observations of previous system behavior are used to improve the so called policy describing the action selection strategy of the system. This principle of learning from observations is used to construct an intelligent self-optimizing system that is able to fulfill several predefined tasks in dynamically changing environments.

5.3.9.1 Principle

The statistical planning approach for mechatronic systems described in this section relies mainly on a statistical data base and rewards; it applies the principles of Reinforcement Learning. The environment is modeled as a discrete finite non deterministic Markov decision process as described by Sutton and Barto (1998) [197]. The mechatronic system measures its current state, selects an action according to a given policy and performs this action. This leads to a transition into a new state and generates a reward signal. Finally, the mechatronic system observes the new state and the reward and compares it with the previous state and the action performed. Based on this comparison the policy is adapted. The objective is to maximize the reward.

5.3.9.2 Methodology

One requirement is that the statistical planning, i. e. the creation of statistical data and the planning, must be done online. This requirement arises since it is hard to create a statistical data base for a real world system where model values appropriately reflect the properties of the environment in which the system works. Hence, the algorithms used for statistical planning must take into account the limited computational power of the mechatronic system.

One algorithm recently investigated by several researchers, which is able to fulfill the requirements is **Q-Learning** [205]. It is an off-policy temporal difference learning algorithm and uses an action-value-function whose update can be expressed recursively. This allows the online execution of the algorithm. The action-value-function $Q(s,a)$ is used to compute the benefit, if a given action in a given state is executed while a fixed policy follows. This action-value-function is similar to the cost functionals presented in Sect. 1.4.1.

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha \left[r_{t+1}(s,a) + \gamma \max_{a'} Q_{t+1}(s',a') - Q_t(s,a) \right] \tag{5.20}$$

Equation 5.20 is a so called sample backup update of the action value function where $r(s,a)$ specifies the reward for taking action a in state s, α denotes a step size parameter and γ a discounting factor used to handle continuous tasks. The step size parameter controls the learning rate while the discounting factor determines the importance of future rewards. $Q(s,a)$ is the quality of a state-action combination. One drawback when using this approach for statistical planning is that it needs a large number of episodes before it converges making it very time consuming. An episode is a single run from the initial configuration until s' is a final state.

To overcome this problem, the **Prioritized Sweeping** algorithm [147] can be used. The main idea is that a model of the environment is maintained. In this context the term model means everything the mechatronic system can use to predict the reaction of the environment when a certain action is performed while the system is in a certain state. In the described case it means that after the performance of an action and updating the action value function, several steps are simulated using the stored model of the environment to predict the outcome. This can speed up the approach by a factor of several magnitudes compared to the classical Q-Learning approach. Additionally, convergence to the optimal policy can be guaranteed, as shown by Li and Littmann (2008) [136].

5.3.9.3 Platforms and Applications

The approach was implemented on the miniature robot platform BeBot [99] (cf. Sect. 2.2) in order to improve the behavior for the application presented in Sect. 2.2.4. The main sensor used to measure the current state is the camera of the BeBot. Moreover, in order to use classical learning algorithms like Q-Learning, a few assumptions concerning the state space of the environment were made. The state

space is assumed to be discrete and of finite size. In addition, a restriction of the set of possible actions took place and the mapping of abstract actions to the actual motor commands was fixed. The used state space and action set is based on Asada et al. (1995) [11].

Image processing is done directly on the BeBot using a color based image segmentation and feature classification approach [111]. The data extracted from the images are then used to determine the current state of the BeBot, for instance, its (discretized) distance to objects in its environment, which could either be goals it has to reach or obstacles it has to avoid. This state information is further used as input of the behavior module, i. e. as input of the statistical planning algorithm.

Practical evaluations of this approach revealed, that noisy images and high sensitivity of the color based feature extraction to illumination changes often lead to the detection of abrupt state changes of the system, e.g. since the detected objects or their positions vary between subsequent images. Another problem is that using the camera a BeBot is not able to perfectly determine its current state. Often several states can be possible due to the limited information the BeBot receives through the use of its camera. The problems were solved by considering a so called hidden state in the model. These models are called partially observable Markov decision process (POMDP) [143]. In POMDPs it is assumed that the actual state (hidden state) of the underlying Markov decision process is not directly observed but the given observations appear with a certain probability in each state. Rather than always having a fully observable state, a belief state probability distribution over all the states has to be maintained. The probability for each state s to be the belief state can be computed recursively, i. e. based on the last belief state, the last action a, the current observation o, and the transition and emission probability parameters of the model as shown in Eq. (5.21).

$$b^{t+1}(s_j) = \Pr\left(s_j|o,a,b^t\right) = \frac{\Pr(o|s_j)\sum_{s_i\in S}\Pr(s_j|s_i,a)\,b^t(s_i)}{\sum_{s_k\in S}\Pr(o|s_k)\sum_{s_i\in S}\Pr(s_j|s_i,a)\,b^t(s_i)} \tag{5.21}$$

The parameters of the model can be computed offline using a modified version of the Baum-Welch algorithm [143].

This leads to a belief state which is no longer discrete. That makes it impossible to find an optimal policy using the described algorithms. Solving POMDPs directly needs a high computational effort. So, only the most likely state and output are considered to be the actual hidden state. The BeBot then takes the selected state as its current state and uses it as the basis to determine its next action. Based on this method only the underlying MDP must be solved using Q-Learning or Prioritized Sweeping as described above.

5.3.10 Behavior Planning in Nondeterministic Environment

Philip Hartmann

In order to increase the dependability of self-optimizing mechatronic systems, **cognitive planning components** with enhanced information processing are also integrated into the system. These components allow mechatronic systems to plan their behavior in order and fulfill individual tasks independently and proactively. A single task represents a sequence of actions executed by the mechatronic system within a limited time frame in order to reach a given goal state. Along with bare fulfillment of that task, i.e. finding an arbitrary sequence of actions to reach the desired goal-state, planning tries to minimize or maximize objectives, such as minimizing energy consumption. For this reason, actions are only selected if their expected results fit the desired objectives. With respect to dependability, it is possible to create alternative plans for critical situations before they arise, i.e. for particular environmental or low energy situations. However, this may decrease the availability of the mechatronic system and the reliability of subsequent task fulfillment. Furthermore, behavior planning considers the continuous and nondeterministic environment of the system (cf. [118]).

When modeling a planning domain for behavior planning of intelligent mechatronic systems (cf. [118, 119, 125]), the main challenge is to map the partial function solutions onto actions within the framework of PDDL (Planning Domain Definition Language, cf. [65]). Depending on the amount of detail desired when modeling these functions, this approach results in a higher or lower abstraction of actions. In case the of behavior planning, the executed partial function solutions are called operation modes. Thus, a planning problem for mechatronic systems can be formulated as follows (adapted from [119]):

- OM is a finite set of available operation modes,
- S is a finite set of possible system states, and
- $\mathbf{s} \in S$ is a state vector with $s(i) \in \mathbb{R}$ for the i-th component.

Furthermore, for each operation mode $om \in OM$:

- $prec^{om} := \{(x_{lower} < s(i) < x_{upper}) | x_{lower}, x_{upper} \in \mathbb{R}\}$ is the set of preconditions which must be true for the execution of operation mode om and
- $post^{om}$ is a set of conditional numerical functions describing the change of influenced state variables. A condition is a logical expression (conjunctions and disjunctions) of comparison operations; if a condition is true, the result of the corresponding numerical function is assigned to state variable in the next state $\mathbf{s}'$ of the plan [119].

A specific planning problem is the finding of a sequence of operation modes which describes a transition from an initial system state $\mathbf{s}_i \in S$ to a predetermined goal state $\mathbf{s}_g \in S$. Thus, a single task of a mechatronic system is given as a 2-tuple $O = (\mathbf{s}_i, \mathbf{s}_g)$. A solution to the planning problem can be determined by applying a

state space search algorithm (cf. [74]), for example. The optimal solution (e.g. minimum of energy consumption) can be found by computing the specific solutions with respect to the given System of Objectives. For this purpose, Ω is a set of objectives and $f : S \times \Omega \rightarrow [0,1]$ is a function that indicates how well the execution of an operation mode in a given state satisfies the objective. Using the weighted sum of the objectives, the optimal sequence of operation modes can be determined (cf. [119]).

During runtime in a non-deterministic environment with continuous processes, behavior planning has to include methods for handling resulting problems. For example, Klöpper (2009) (cf. [118]) uses a modeling approach to integrate continuous processes based on optimal control and continuous multiobjective optimization (also cf. [73]), as well as estimation obtained by fuzzy approximation. To manage planning under uncertain conditions, different techniques can be combined in a hybrid planning architecture (cf. [119]).

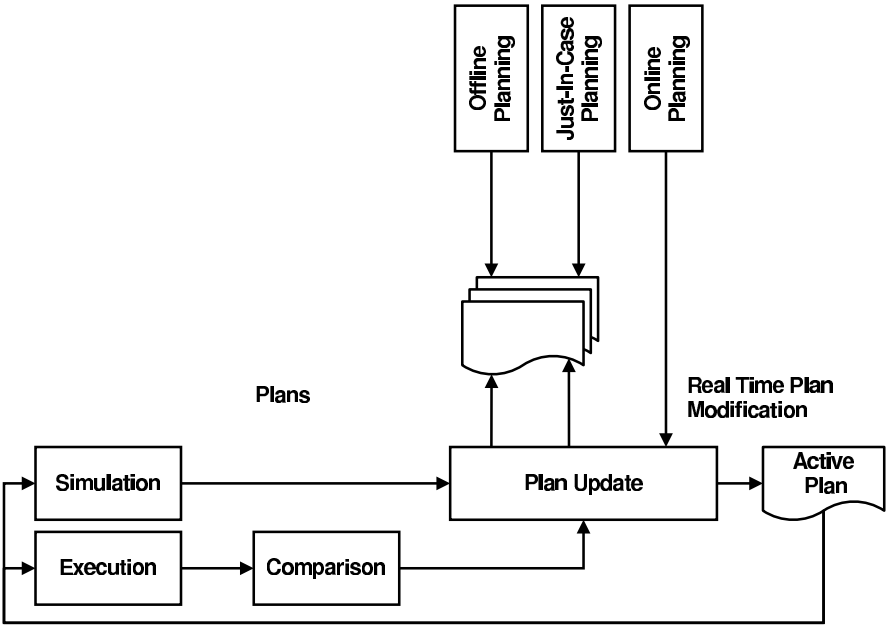

Fig. 5.42 Hybrid planning architecture (source: [119])

Figure 5.42 shows the hybrid planning architecture with the corresponding components for planning, execution and monitoring of plans. The total planning is divided into three separate sections: offline, just-in-case and online planning. The offline planning represents a planning process where, initially, a deterministic and optimal plan in view of the objectives is fully created before execution. The resulting plan is used in the just-in-case planning to do a probabilistic analysis for plan deviations. The present and deterministic plan is examined for estimated variances

in order to proactively generate conditional branches, with alternative plans for critical system conditions. A threshold specifies the maximum probability of state deviations which would result in a generation of conditional branches (see [125], in particular also [125] and [118].)

For this purpose, an additional stochastic planning model is formulated based on the deterministic planning model. This consists of stochastic states $\mathbf{s}^p$ with $|\mathbf{s}^p| = |\mathbf{s}|$, where $range(s^p(i)) \rightarrow P(\mathbb{R})$ is the values range and $distribution(s^p(i))$ the probability distribution of the state variable $s^p(i)$ and a stochastic variant of the operation modes. Let $in_s^{om} \subseteq pre^{om}$ be a subset of input variables and $out_s^{om} \subseteq post^{om}$ a subset of output variables. For each output variable $o \in out_s^{om}$, a Bayesian network (cf. [20]) bn_o^{om} is created to formulate the stochastic relation (cf. [118, 119]; for a concrete example of creating a stochastic model cf. [125]). As a result, it is now possible to use the just-in-case-planning to generate alternative plans for situations that could occur with high probability during operation.

The online planning (cf. Fig. 5.42) serves as a fallback mechanism; it selects the optimal operation mode for the next execution step. Thus, operation in previously unplanned situations is guaranteed. A simulation of the continuous system behavior will check whether the current action of the active plan is executable under the given environmental conditions. If this is not possible, online planning is necessary, e.g. for a situation with extreme environmental influences such as heavy rain. While completing the execution of previously planned operation modes, a comparison of planned and actually reached system states is carried out.

A process for plan updating will check whether a pre-determined plan is available or whether a plan modification by the online planning is necessary. This will guarantee the immediate availability of the next operation mode (cf. Fig. 5.42).

The just-in-case and online planning are implemented as anytime algorithms (for the usage of anytime algorithms in intelligent systems cf. [214]). The planning process can be interrupted at any time to obtain a result, but with increasing time for calculations it provides a higher quality of result, as it is possible to generate more branches and to reach a higher depth of planning.

The dependability our type of system can be influenced by various factors. A major factor is the availability of energy, as this is crucial for the operation of the system. To ensure the dependability of the mechatronic system, it is essential to use the energy storage in a valid range and in particular to continuously observe the state of charge. Energy Management can use behavior planning to proactively schedule future energy demands according to the fulfillment of the current task, which increase the dependability of the mechatronic system (cf. [125]). Table 5.3 shows the values for operation modes derived from the multiobjective optimization from the Active Suspension Module of the RailCab system.

The experiments described here were intended to allow to evaluate three hypotheses (cf. [119]). One of these hypotheses in connection with the dependability was that a lower threshold probability and a higher number of alternative plans increases the reliability of the just-in-case planning ([125]). The simulated experiments included four scenarios (source [119]):

Table 5.3 Values for f_1 (weighted average body acceleration in m/s^2) and f_2 (energy consumption in ws) of operation modes derived from the multiobjective optimization of the Active Suspension Module. (source: [119])

OM	Objective function	Track type I	II	III	IV	V	VI	VII	VIII	IX	X
a	f_1	0.117	0.233	0.350	0.466	0.583	0.699	0.816	0.932	1.049	1.166
	f_2	196	393	589	786	982	1179	1375	1572	1768	1965
b	f_1	0.152	0.304	0.457	0.609	0.761	0.913	1.066	1.218	1.370	1.522
	f_2	165	329	494	659	823	988	1153	1317	1482	1647
c	f_1	0.192	0.385	0.577	0.770	0.962	1.155	1.347	1.540	1.732	1.925
	f_2	142	283	425	567	709	850	992	1134	1275	1417
d	f_1	0.224	0.449	0.673	0.897	1.122	1.346	1.570	1.794	2.019	2.243
	f_2	122	245	367	489	612	734	856	979	1101	1224
e	f_1	0.262	0.523	0.785	1.047	1.308	1.570	1.832	2.093	2.355	2.617
	f_2	104	208	313	417	521	625	730	834	938	1042
f	f_1	0.298	0.595	0.893	1.191	1.488	1.786	2.084	2.381	2.679	2.977
	f_2	87	173	260	346	433	520	606	693	779	866
g	f_1	0.331	0.662	0.994	1.325	1.656	1.987	2.318	2.649	2.981	3.312
	f_2	69	138	206	275	344	413	482	550	619	688
h	f_1	0.375	0.749	1.124	1.499	1.873	2.248	2.623	2.997	3.372	3.747
	f_2	50	99	149	199	248	298	348	398	447	497
i	f_1	0.435	0.870	1.305	1.739	2.174	2.609	3.044	3.479	3.914	4.349
	f_2	27	55	82	110	137	164	192	219	247	274

1. ($\pm$0%): The energy consumptions drawn from track networks were not changed during simulation.
2. ($\pm$15%): The energy consumptions drawn from track networks were either decreased or increased by a random value up to 15%.
3. ($+$15%): The energy consumptions drawn from track networks were always decreased by a random value up to 15%.
4. ($-$15%): The energy consumptions drawn from track networks were always increased by a random value up to 15%.

The results are shown in Fig. 5.43 (for a detailed description of the simulation parameters and the executed scenarios cf. [119]) When regarding the percentage of failed plan execution during the simulation runs for different scenarios, adjusting the two parameters threshold values and number of alternative plans reduces the number of failed plans significantly.

A detailed explanation of behavior planning for mechatronic systems can be found in [119, 125]. In particular, [125] gives a deeper understanding of the probabilistic plan structure used in the analysis of the just-in-case planning. The basic methods were originally published in the dissertation by Klöpper (2009) [118], which may also be a good starting point for further information.

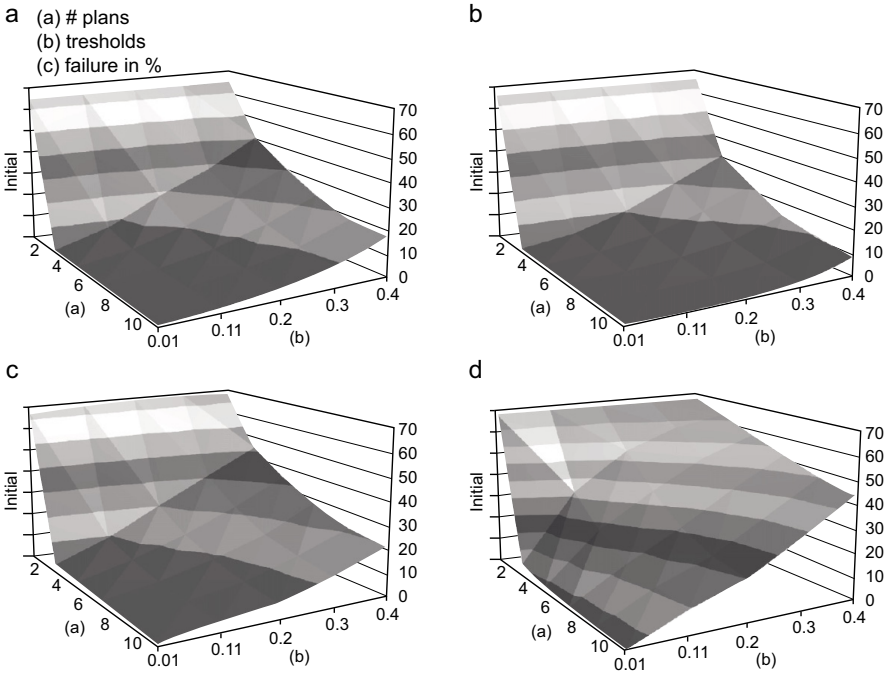

Fig. 5.43 Percentage of failed execution depending on threshold probability and number of available alternative plans; (a) Return to Standardplan (±0%); (b) No Return to Standardplan (±0%); (c) Return to Standardplan (±15%); (d) Return to Standardplan (+15%) (source: [119])

5.3.11 FIPA Conform Cross-Domain Communication

Philip Hartmann

Another advantage of self-optimizing systems is given by the possibility for intelligent communication of individual subsystems. The FIPA specifications (cf. [106]) provide a suitable way to implement cross-domain communication for autonomous mechatronic systems. To enable a more sophisticated approach the further considerations will include a requirement scenario for the RailCab system. A production facility is pursuing a just-in-time procurement strategy (JIT). To achieve this strategy, the transportation of goods is done by the RailCab system. For this purpose the production facility has access to a data base of RailCabs, which are able to deliver goods in the given time. Because of the RailCab's ability to work in a team, there is the possibility to take a closer look at complex voting scenarios to determine a suitable RailCab for specific orders. Both the production facility and the RailCab system are modeled as multi-agent systems with two different domains represented by ontologies (cf. [101]). The main goal of this section is to show a principal

feasibility for the implementation of a FIPA based communication across the given domain boundary. First, an exemplary overview of the ontologies, that are available to the production facility (domain 1) and the RailCab system (domain 2) is provided. In this approach an interface ontology forms the basis of the cross-domain communication. In addition the communication procedure of the agent interaction, as described by the FIPA standard, is outlined in an FIPA conform auction between the production facility and RailCabs within the JIT radius. The JIT radius forms a set of suppliers. They have in common, that they are able to deliver the required goods within the given time. Therefore, the goal of the auction is to identify the lowest priced RailCab. During the auction RailCabs may occur as a team. Negotiation and voting procedures enables each RailCab to submit an optimal offer for the team.

5.3.11.1 T-Box Design

The following two ontologies are designed to demonstrate in which way the T-Box design of underlying ontologies can be done. It is important to point out, that there is a so called *Open World Assumption* given. This implies the need to explicitly rule out unwanted facts within the ontology design. Another aspect of central importance for the successful deployment of ontologies is their integration. Unfortunately, this is not trivial in general. Figures 5.44 and 5.45 show the conceptual approach to be proportionate to the problem, based on [19] and [12]. Figure 5.44 shows that a distinction is done within the facility (domain 1) between the following levels:

- The *Foundational Ontology* includes the abstract concepts of time, space, object, event, etc. As well as the concepts of major priority for this context, as there are *transporter*, *product* and *package*. It is desired that only one *Foundational Ontology* exists within this design, which serves as a starting point for modelling the *production-side* and the *RailCab-Interface-Ontology*. In this manner, the other ontologies can be seen as specializations and by thus allow integration. The *Foundational Ontology* forms a key specification that allows more specialized ontologies to model redundant concepts.
- The *RailCab-Interface-Ontology* provides the communication interface to the RailCab agents.
- The *domain1-ontology* represents the entire ontology structure, that is available for domain 1.

Figure 5.45 provides an overview of the integration approach for the mentioned ontologies in more detail, with respect to RailCab agent system. It should be noted that both, the already known *Foundational Ontology* and the *RailCab Interfaceontology* can be integrated into the *domain2-Ontology* in the same way. The ontology of the RailCab is similar to the one of the production side in a way, that both share the concept of the *Foundational-Ontology* as well as the terms of the *RailCab interface-ontology*.

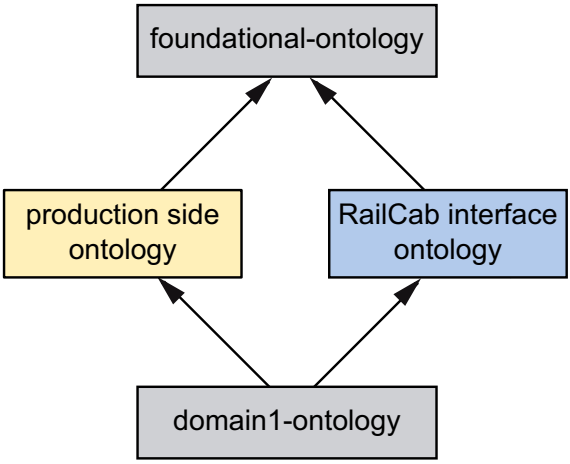

Fig. 5.44 Three levels of ontology generalization regarding the production facility (domain 1)

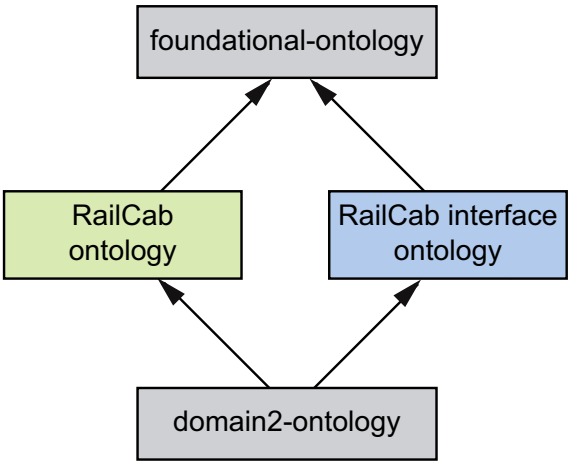

Fig. 5.45 Three levels of the ontology generalization regarding the RailCab system (domain 2)

5.3.11.2 Communication Flow

In this section, a communication protocol is presented that allows the communication between the production site (domain 1) and the RailCab system (domain 2). The shown communication process meets the specifications by FIPA standard. In this, the production side will request RailCabs to make an offer regarding to the transportation of a specified delivery. The contacted RailCabs may be associated with a supplier's fleet, with teams trying to optimize their company's profit. This is realized by distributing received auctions towards their team members as part of a negotiation. In order to vote for the RailCab which offers the best conditions for the group to take over the job. The figure 5.46 illustrates the underlying connection graph of a single RailCab team. Highlighted are the agents *RailCab A* and *C*, and *productionside* because they are particularly interesting for the further consideration of the communication sequence.

It is necessary to find an efficient voting algorithm which allows the RailCabs to optimize their teams benefit within the given time. The problem is equivalent to the principle *Leader Election* problem where agents are differentiated on the basis of a utility function. Since this may not be clear, however, ambiguity of the function is not relevant for maximizing the supplier's earnings.

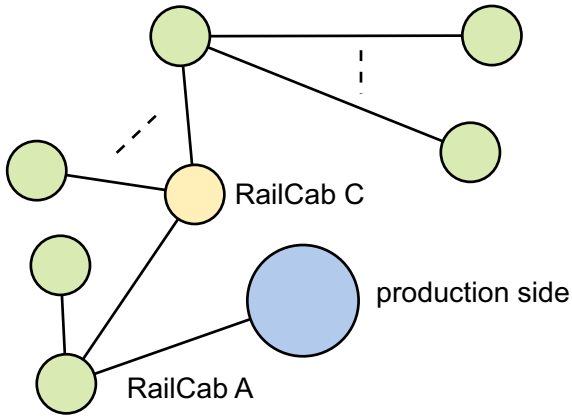

Fig. 5.46 Arbitrary connection graph of Railcab units and the production site

It must also be assumed that not every member of the team can exactly name all other team members that are relevant for the problem. Since individual data might not be actual, or communication might not be successful within the given deadline by the production side (cf. Fig. 5.46). The voting problem in principle is a *Leader Election* in a spanning tree with asynchronous communication. It is therefore useful to take advantage of the *FloodMax* approach here. Unfortunately, the algorithm is generally in arbitrary graphs with asynchronous communication is very difficult to use (cf. [139]). Therefore the algorithm will be used in an optimized form regarding to the problem.

The voting procedure can be divided into several sub-routines. Figure 5.47 illustrates the reaction of the RailCab receiving a `call-for-proposal` message. It sends an `inform` message to all known team members containing the following:

- The original `cfp` message, that was send from the production site, this contains all the terms of the offer and a deadline until proposals in the form of `proposal` messages have to be done.
- The value that was determined by the utility function, with respect to the auction. This should be the benefit of the team as the transmitter can achieve if it would accept the job.

Each receiver of such an `inform` message has to check weather the utility function may result in an higher value, with respect to his individual parameters. In case of a higher value the receiver knows that a better result for it's team can be achieved by taking the job, rather than the team member, that has send the `inform` message.

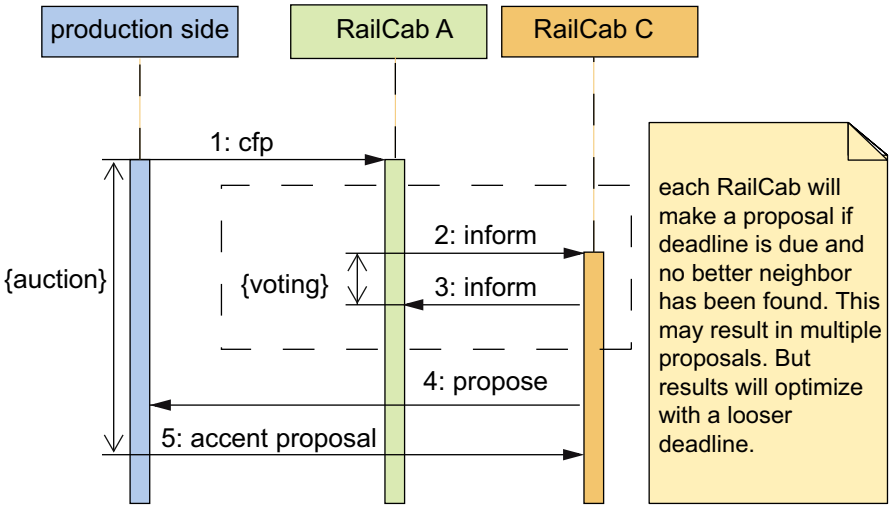

Fig. 5.47 Sequence diagram communication flow

If the deadline, given by the production side within the initial `cfp` message, is about to expire, each RailCab is in a difficult situation. The potential team benefit can only be optimal if and only if the production side gets proposals from team members, which can grant a maximal profit. Because the production side itself does not differ between the optimal and suboptimal team members, because there is no data nor interest about it, it may choose randomly among the best bids. In order to maximize the potential team profit, it would be best for those suboptimal team members to ignore the auction, by this they higher the probabilities of each optimal team member to gain the job. Figure 5.47 shows an example for the flow of communication, the presentation is limited for reasons of clarity to only three RailCabs of the same team. It can be seen how the auction is initiated and in which way the negotiation of RailCab voting takes place.

5.3.12 Preparing Solution Pattern "Hybrid Planning"

Roman Dumitrescu and Harald Anacker

To enable self-optimization in mechatronic systems, planning methods are of high importance. However, classic planning methods consider state transitions as a black box, so only the state before and after the transition will be accounted for the self-optimization process. In mechatronic systems the continuous run of the processes taking place within the system should not be neglected in order to avoid deviations during the execution of a plan. As a consequence, mechatronic processes have to be described in a continuous way, but also needs to be planned. The **solution pattern** "Hybrid Planning" is based on the detailed method for the behavior planning in non-deterministic environment which was explained before. Core of the solution

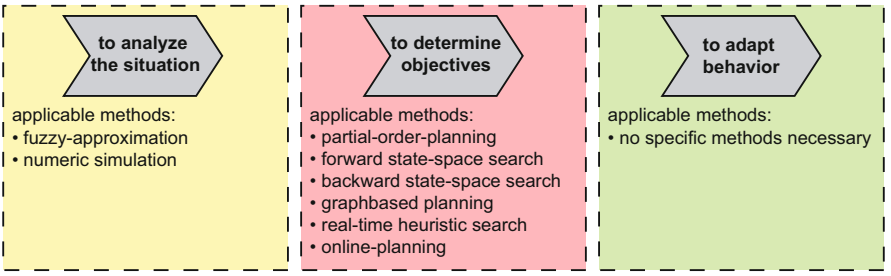

Fig. 5.48 Possible methods fort he implementation of the solution pattern "Hybrid Planning"

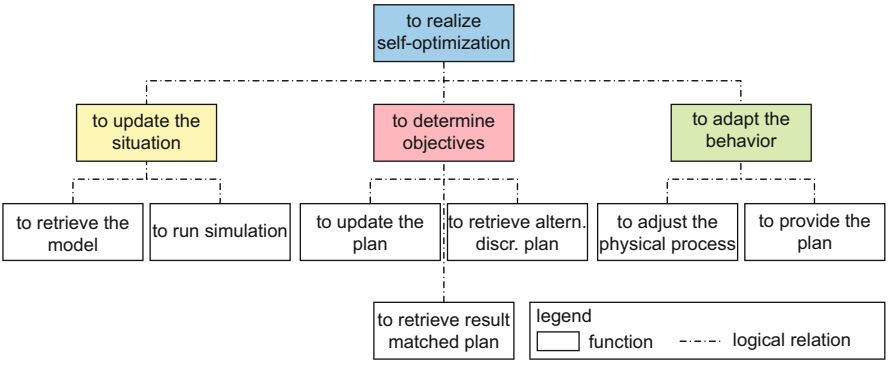

Fig. 5.49 Partial model behavior–activity of the solution pattern "Hybrid Planning"

pattern is the combination of classical planning algorithms with methods for the approximation of continuous behavior. Regularly the approximation is realized by a simulation model and an update of the existing plan. The solution pattern could be realized by different combination of methods that are illustrated in Fig. 5.48. The different methods are allocated to the different phases of the self-optimization process.

The main planner is subdivided in two different planners. A discreet planner generates the offline plan before the system starts running. Depending on the different usage conditions additional planners are necessary, for example to cooperate with additional (sub)systems. Figure 5.49 shows the partial model functions of the solution pattern "Hybrid Planning". A discrete planning method "determines the objectives" to generate plans or partial plans, whereas the continuous parts of the planning focuses on "to update the situation" for the evaluation of the planning steps. Merging the results of the continuous planning into the discrete planning results in "adapting the behavior" whether by "providing the plan" or by "adjusting the physical process".

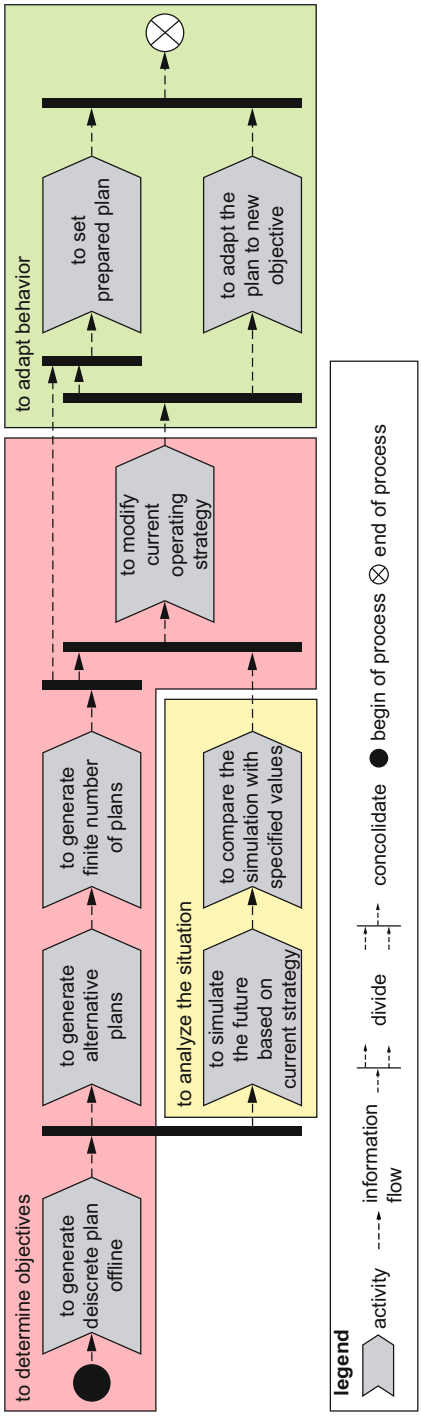

Fig. 5.50 Partial model active structure of the solution pattern "Hybrid Planning"

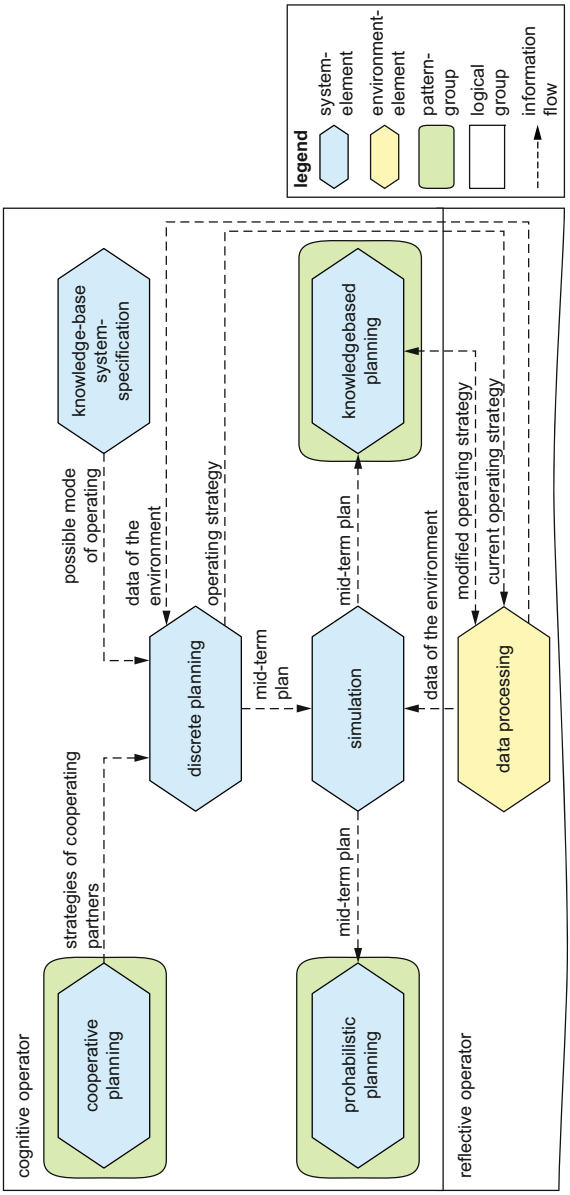

Fig. 5.51 Partial model functions of the solution pattern "Hybrid Planning"

The procedure of the hybrid planning takes place in several steps (cf. Fig. 5.50). As opposed to the three actions of the self-optimizing process, the hybrid planning has at the beginning four main activities, which can be classified into the three steps of self-optimization anyhow. Even before the "Analysis of the situation" takes place, there is an initial offline planning which determines the initial objectives (red element in Fig. 5.50). Then, the plan actions are analyzed in simulation by comparing

the plan with the current situation and the current plan gets modified or rescheduled, not only by calculating alternative plans, but also potentially by using the data from the simulation steps. Eventually the currently active plan gets executed.

For the implementation of the functions of the solution pattern "Hybrid Planning" the following essential system elements and arrangement of them were identified (cf. Fig. 5.50).

The system elements "offline planning" and "predictive planning" are realizing the function "to retrieve alternative discrete plan", which is illustrated in Fig. 5.51. The "online planning" holds the function "to retrieve result matched plan". The system element "approximation of continuous behaviors" carries out the functions "to run simulation", "to retrieve the model" and "to run simulation".

5.4 Dynamic Reconfiguration

Sebastian Korf and Mario Porrmann

When principles of self-optimization refer to the topology and structure of **microelectronic systems**, a reconfiguration of the system architecture or of the dedicated system components is required. In this context, reconfigurability means the possibility to change the functionality or interconnection of hardware modules in microelectronic systems before and during operation. We distinguish between fine-grained (FPGA-based) and coarse-grained (processor-based) reconfigurable architectures. These architectures assign two different hardware technologies for the process step "Selection of Hardware Technology" in the design and development of electronic engineering in Sect. 3.3.4 on page 88. In Sect. 5.4.1, fine-grained FPGA-based dynamically reconfigurable systems are introduced which facilitate System on Programmable Chip (SoPC) designs with a complexity of several million logic gates, several hundred kBytes of internal SRAM memory, and embedded processor cores. Section 5.4.2 will detail our work on embedded processor cores that can adapt their internal structure at run-time. In Sect. 5.4.3, two modeling approaches for reconfigurable architectures are described, which are used to determine the appropriate model for the process step "Modeling of Information Processing Dynamic Reconfigurable Hardware" in the design and development of electronic engineering. The modeling approaches are used in Sect. 5.4.4 for the design of a dynamically reconfigurable system. The design methods are used within the process steps "Modeling of Information Processing Dynamically Reconfigurable Hardware" to "Synthesis of Dynamically Reconfigurable Hardware". Section 5.4.5 concludes with concrete applications for fine-grained and coarse-grained architectures.

5.4.1 Fine-Grained Reconfigurable Architectures

FPGA-based reconfigurable systems try to fill the gap between flexible, programmable microprocessors and application-specific hardware with respect to cost, energy-efficiency, and performance. Partially and dynamically reconfigurable

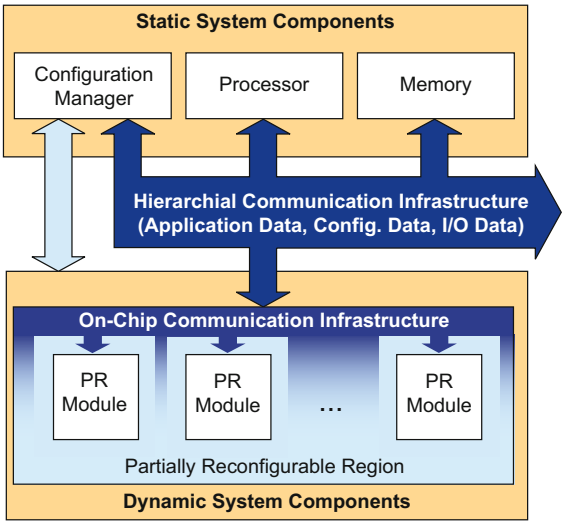

Fig. 5.52 Architecture of a dynamically reconfigurable system

systems add an additional level of flexibility since the functions and interconnectivity of their hardware resources can be changed during run-time. In this way, the architecture can be flexibly adapted to changing environmental conditions. The traditionally static partitioning into hardware and software can be replaced by a dynamic partitioning at run-time. Therefore, dynamically reconfigurable hardware is a promising technology for information processing in self-optimizing systems. Nevertheless, these methods are rarely used in real-world applications due to a lack of sophisticated design tools that support partial reconfiguration. Therefore, new design methods and new hardware platforms have been developed, which enable an efficient utilization of dynamically reconfigurable systems.

Figure 5.52 shows the system architecture that is used for the implementation of FPGA-based dynamically reconfigurable hardware. The FPGA resources are divided into a static and a partially reconfigurable region (PR region), connected by a hierarchical communication infrastructure. The static region typically comprises of one or more processors, embedded memory, and a configuration manager that manages the available resources, configuration files, and the reconfiguration process. The dynamic system components are represented by partial reconfiguration modules (PR modules) and the placement of a PR module is done by configuring a predefined area in a PR region of the FPGA with the corresponding configuration data. PR modules can be loaded into or erased from the system during run-time. Communication between PR modules as well as with the static components is realized by a flexible on-chip communication infrastructure. Using state-of-the-art FPGAs enables the realization of complete systems on one chip, since these devices provide all the logic resources that are required.

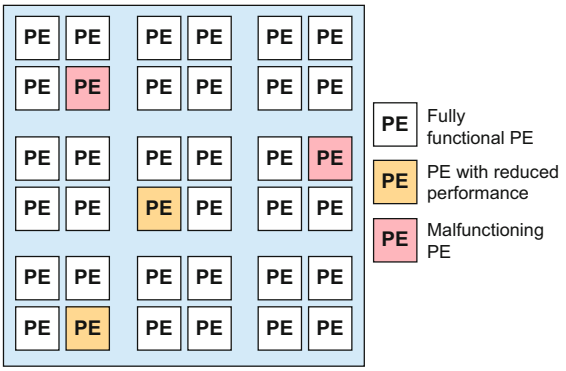

Fig. 5.53 Multiprocessor with processing elements in different conditions

5.4.2 *Coarse-Grained Reconfigurable Architectures*

The high flexibility of fine-grained reconfigurable systems, like FPGAs, comes at the cost of high overhead in terms of chip area, timing delays, and power. An alternative to dynamically reconfigurable FPGA-based systems are **Multi Processor System on Chip** (MPSoC) architectures, which are also able to cope with today's requirements on short time-to-market due to manageable design complexity, high energy efficiency in spite of high performance, and high reliability [210]. Here, we target on-chip multiprocessors composed of hundreds of simple embedded processors, connected by a network on-chip (NoC) [107]. In these architectures, the inherent redundancy can be utilized to increase reliability and system lifetime [165].

As illustrated in Fig. 5.53, it is expected that future on-chip multiprocessors will comprise a growing number of processing elements. Some of them will probably be malfunctioning or provide only reduced performance, e.g. due to semiconductor parameter variations. Unfortunately, more and more of these system faults occur dynamically during operation. The goal of our approach for future self-optimizing MPSoCs is to provide the user with the maximum performance of energy efficiency that can be achieved in the actual system state by utilizing as many hardware building blocks of the architecture as possible. Therefore, we integrate methods for dynamic reconfiguration into the architecture, which enable reconfiguration of the interconnection between the building blocks of the processors at run-time. Details about these methods are described in Sect. 5.4.5.3.

5.4.3 *Modelling*

The realization of dynamically reconfigurable systems requires a complex design flow that cannot be established based on commercially available tools. Therefore, two modelling approaches will be introduced. In Sect. 5.4.3.1 the PALMERA model abstracts the design on different layers. The DMC model described in Sect. 5.4.3.2 further introduces analysis methods and concepts.

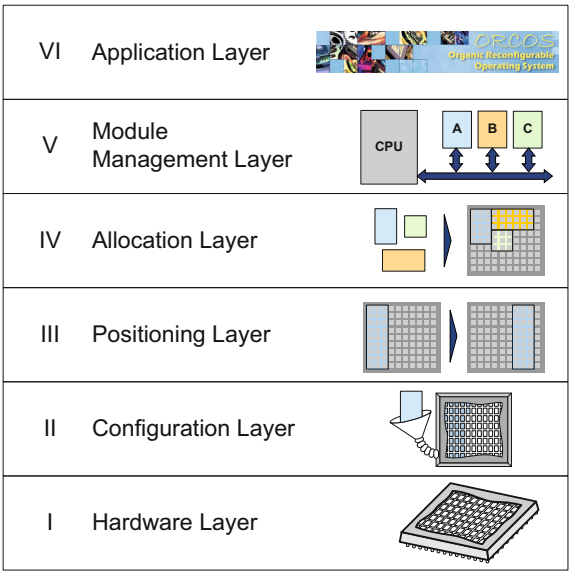

Fig. 5.54 PALMERA – Paderborn Layer Model for Embedded Reconfigurable Architectures

5.4.3.1 PALMERA (Paderborn Layer Model for Embedded Reconfigurable Architectures)

In order to realize dynamically reconfigurable systems, we propose a layer-based approach to dynamic reconfiguration in [116]. This model systematically abstracts the underlying reconfigurable hardware to the application level by means of six specified layers and well defined interfaces between these layers, as depicted in Fig. 5.54. The main objective is to reduce the error-proneness of the system design while increasing the reusability of existing system components. Additionally, it can be used for the comparison and consolidation of various approaches to dynamic reconfiguration that have been proposed in literature. Each layer offers services to the next higher layer and makes requests of the next lower layer. As for other known layer models in computer science and engineering, the interfaces between layers are standardized to enable an easy and separate exchange of single layers without modifying the whole system.

The first layer in PALMERA is the **Hardware Layer**, representing the underlying reconfigurable hardware. As such it is defined after choosing an FPGA architecture for the system. The interface to its adjacent layer is the configuration port of the chosen FPGA. This makes the interface between the Hardware Layer and the Configuration Layer the only non-specified interface in our model. It is the task of the Configuration Layer to adapt to this interface.

The purpose of the **Configuration Layer** is to abstract from the underlying hardware and its configuration port and to give a standardized interface to the Positioning Layer. For Xilinx FPGAs, the configuration ports are typically either the internal configuration access port (ICAP) or an external configuration port such as the SelectMAP interface. It should support write and read-back of partial bitstreams as

well as a complete configuration with a complete bitstream. Due to the streaming-based configuration interfaces of common FPGAs, the Configuration Layer can efficiently be realized in hardware. To shorten configuration time and to avoid storing the bitstreams in the Configuration Layer before configuring, the interface to the Positioning Layer should offer a streaming-based data input for incoming bitstreams as well as a data output for storing the information, which was read back from the FPGA.

The **Positioning Layer** adapts the position information of a given bitstream to a desired location on the FPGA. This can significantly reduce the number of bitstreams that have to be stored for each module since all equivalent (homogenous) areas on the FPGA can be configured with the same bitstream in this case. This also applies to existing heterogeneous architectures if the placement is chosen appropriately. The Positioning Layer thus performs a bitstream manipulation that can be done in software (e.g. with PARBIT [103]), or in hardware (e.g. with REPLICA [112]). However, a hardware implementation of the Positioning Layer is preferred, since it can be realized using only a few resources, without increasing the configuration time significantly. The Positioning Layer has a separate interface to the memory holding the partial bitstreams. It is the uppermost layer that deals with bitstreams as physical representations of the modules. The three upper layers treat the modules as abstract units. Hence, the interface to the Allocation Layer consists only of control flow signals. The services offered to the Allocation Layer are loading and reading configuration data to/from a given area of the FPGA. In addition, combined reading and writing should be offered, in order to shift active modules as needed for a defragmentation of the FPGA.

The **Allocation Layer** manages all available reconfigurable hardware resources on the FPGA and assigns appropriate positions to incoming modules. Therefore, the Allocation Layer holds an abstract image of the resources which can be allocated and deallocated during run-time. In addition, a list of all currently loaded modules is stored in this layer. It holds information about the modules' names, positions, status (active, inactive, etc.), module type and a unique ID. This ID is used to identify a module within the upper two layers. The mapping of a module to an area on the FPGA is done according to a given placement strategy, such as first fit, best fit, or even more sophisticated strategies for heterogeneous FPGAs, as proposed, e.g. in [121]. When needed, the possibility to defragment the FPGA area can also be implemented in the Allocation Layer. A defragmentation can be accomplished automatically (e.g. when a certain degree of fragmentation is reached) or it can be done on-demand. The Allocation Layer offers the service to place a module of a given type on the FPGA or to delete a module with a given ID. The latter is realized by loading an empty bitstream to the FPGA (as required for some fine-grained placement approaches) or by simply deallocating the used resources.

The **Module Management Layer** completely abstracts from the reconfigurable hardware. Its main service offered to the Application Layer is to provide access to a module of a requested type. For this reason it holds a list of all currently loaded modules. With this list a set of different strategies can be implemented, e.g. a caching of unused modules. For this strategy, modules are set to inactive after being released

from an application. In case an application needs a module of which an inactive instance exists, a time consuming configuration can be avoided by just reactivating the concerned module. Inactive modules get deleted from the FPGA as soon as the Allocation Layer runs out of free resources. In this case the Module Management Layer chooses a module to be deleted. This can be done according to different strategies such as longest-unused-module-first or module priorities.

The **Application Layer** represents any task using the dynamically reconfigurable hardware. This could be either software running on a (embedded) processor, such as an operating system, or other static hardware modules. In the last-mentioned case it is possible that multiple applications use the dynamically reconfigurable hardware modules simultaneously.

Depending on the architecture of the system, all layers can be implemented either in hardware or in software. On the Application Layer e.g. a small reflex operator can be realized as a pure hardware solution or as a complex software solution running on a CPU with an RTOS such as *ORCOS*. PALMERA has been included in an extension of the OS Monta-Vista-Linux, where the bottom layers (up to the Positioning Layer) are implemented in hardware and the upper layers are software implementations running on a PowerPC processor on a Xilinx FPGA [174].

5.4.3.2 DMC Model for Dynamically Reconfigurable Systems

The DMC (Design, Module, and Component) model [124] is used as a basis for the analysis of the methods and concepts for dynamic reconfiguration. The model divides the placement of a hardware module into three levels of abstraction: Design, Module, and Component. It defines the relations between these levels and is restricted to the fundamental measures that are required for the realization of dynamic reconfiguration. Therefore, methods for placement and scheduling in dynamically reconfigurable systems can be formally described using the DMC model. In the DMC model, reconfigurable architectures are modeled as reconfigurable cells, which are arranged in a matrix structure and interconnected by a communication infrastructure. Fine-grained architectures like FPGAs can be modeled as well as coarse grained and heterogeneous architectures. A design in the DMC model represents an abstract specification of the hardware design, e.g. based on a hardware description language or a schematic. The term module refers to a specific implementation of a design, e.g. generated by a hardware synthesis. Finally, the component represents an instance of a module. Several instances of the same module may be placed in parallel at different positions on the reconfigurable hardware. For the analysis of architectures and methods based on the DMC model, we have developed the simulation framework SARA (Simulation Framework for Analyzing Reconfigurable Architectures). SARA is specifically designed for FPGA-based architectures. The simulation flow of SARA is split into three phases. In phase one, a Virtual Synthesis tool creates the modules to be downloaded to the FPGA from a given set of module descriptions. These descriptions include information about the required FPGA resources as well as minimum module dimensions. According to one or more given synthesis strategies, module implementations with various aspect

ratios are generated for each module description. In phase two, an RTR-manager (run-time reconfiguration manager) executes the given benchmark and places the required modules in a predefined order onto a virtual FPGA. The simulation analysis is done in phase three by a dedicated analysis tool integrated in SARA. In the context of self-optimizing systems, SARA is specifically used for the analysis of new placement and defragmentation strategies [120, 122, 123].

5.4.4 Design Methods for Dynamic Reconfigurable Systems

Based on the abstract modelling of PALMERA and the DMC model, the Integrated Design Flow for Reconfigurable Architectures (INDRA) has been developed that guides the designer through the different implementation steps to create a concrete dynamic reconfigurable system architecture. This design flow is described in Sect. 5.4.4.1. Section 5.4.4.2 introduces algorithms for the flexible placement of dynamically reconfigurable (hardware) modules. A design method for a Hardware-in-the-Loop (HiL) implementation is shown in Sect. 5.4.4.3.

5.4.4.1 INDRA (Integrated Design Flow for Reconfigurable Architectures)

The design-flow of a partially reconfigurable system is different from the standard design-flow of reconfigurable systems, which only allows the reconfiguration of the whole FPGA. To efficiently handle these deviations from the standard flow, the Integrated Design Flow for Reconfigurable Architectures (INDRA) has been developed (cf. Fig. 5.55). INDRA integrates all tools that are required to design dynamically reconfigurable systems based on Xilinx FPGAs [88]. It combines commercial state-of-the-art tools and tools that have been adapted or especially designed for this framework. INDRA supports a flexible one-dimensional or two-dimensional module placement.

First, the given application is partitioned into static and dynamic system components. The area that is used for the static components is also referred to as the base region. The partitioning depends on the properties of the selected device, such as the reconfiguration granularity (length of a so-called configuration frame), and on the selected placement approach. The architecture of Xilinx Virtex-4 to Virtex-7 devices allows a two-dimensional placement of partial reconfiguration modules at a granularity of a configuration frame. The description of the static and dynamic system components as well as their interconnections between each other on the top level are specified in a hardware description language (HDL). The synthesis of the base region and of the PR Modules is performed based on the system partitioning. Depending on the size of the modules, which is obtained from synthesis estimation, and on the inherent heterogeneity of the FPGA, INDRA determines the steps required for the synthesis of the PR Modules. The floorplanning of the system (the mapping of each component to a position on the FPGA) is done by SARA, which implements the DMC model for the dynamically reconfigurable system.

In addition to the partitioning and floorplaning of the FPGA, the concept of partial reconfiguration requires a suitable communication infrastructure for

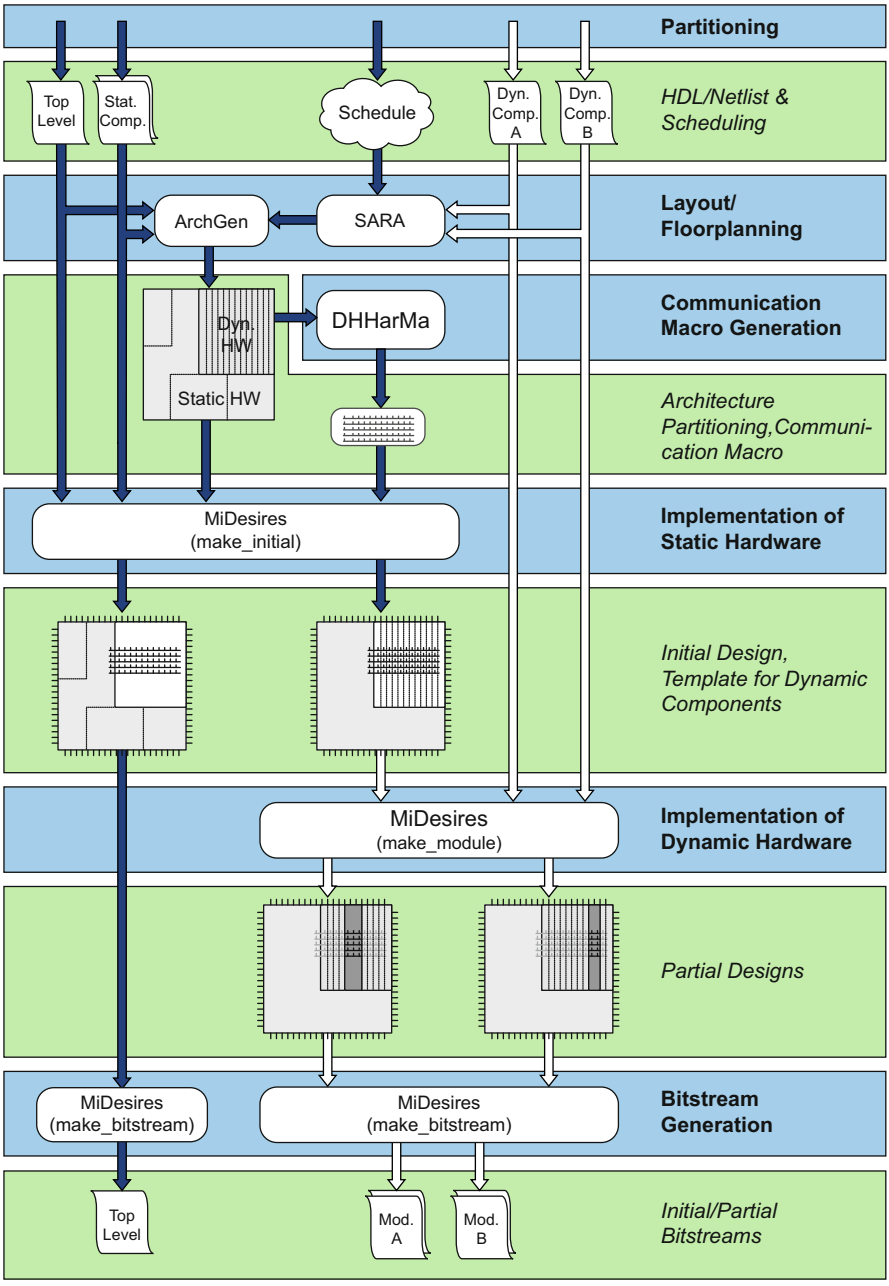

Fig. 5.55 INDRA – Integrated Design Flow for Reconfigurable Architectures

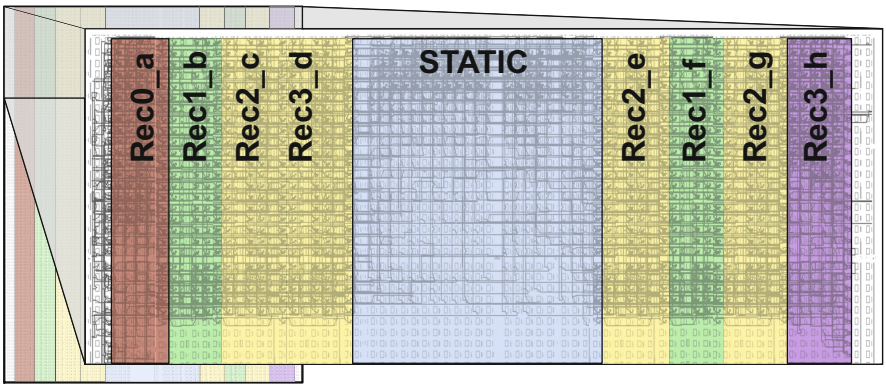

Fig. 5.56 Example of a homogeneous hard macro for a communication infrastructure with 9 regions and 4 different types of regions

interconnecting the PR modules and the base region. The communication infrastructure should not introduce any further heterogeneity in the system to maintain the flexibility of placement by preserving the number of feasible positions of the PR modules. Homogeneity implies that the individually reconfigurable tiles (a tile is the atomic partially reconfigurable unit, cf. 5.4.4.2) are connected by the same routing resources. Thus, modules cannot only be placed at one dedicated position, but at any position with sufficient free contiguous resources [87]. Current commercially available FPGA place and route tools lack an option for generating this type of homogeneous designs. The Design Flow for Homogeneous Hard Macros (DHHarMa) [126] targets the automatic generation of homogeneous and regular designs starting from a high-level description, such as VHDL or Verilog. Using DHHarMA, complex communication infrastructures for dynamically reconfigurable systems can be generated based on an abstract high-level description. In [126], examples are presented, using 32 Bit data, 32 Bit addresses, 4 Byte-enable signals, and 4 Bit auxiliary lines. Additionally, dedicated signals are connected to each region for strobe, master request, master grant, region enable, and region reset. The communication infrastructure also supports bursts (transmission of multiple data packets at a time) using an embedded 8 Bit burst counter (cf. Fig. 5.56).

Figure 5.57 shows an example partitioning of a Virtex-4 FX100 FPGA. In the Virtex-4 architecture, the area of a PR region should be multiples of a configuration frame. In the example implementation, we vertically divided the FPGA, so that the resources located left of the center column are dedicated to static system components, and the resources located right of the center column are considered for the tiled PR region.

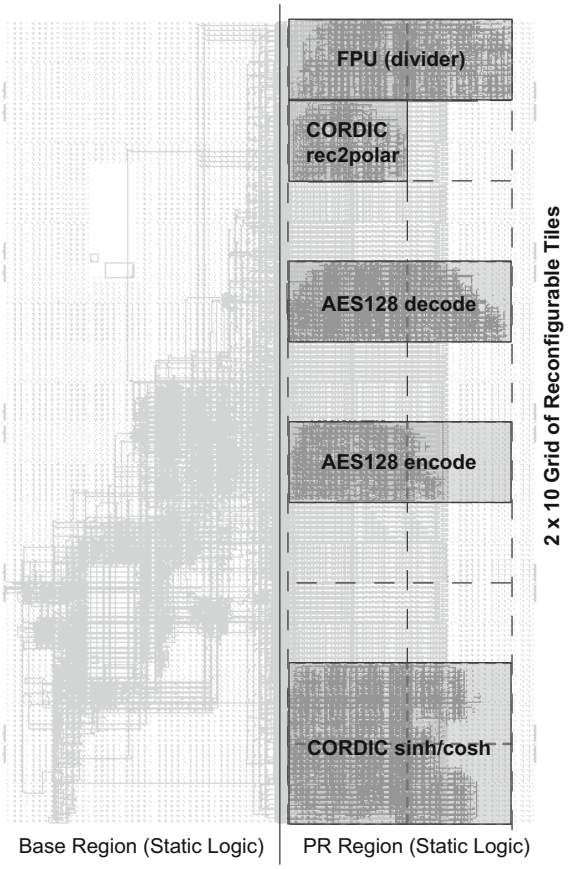

Fig. 5.57 Example for the partitioning of a Xilinx Virtex-4 FX100 FPGA

5.4.4.2 Placement Algorithms for Flexible Dynamically Reconfigurable Systems

Nowadays, most realizations of dynamically reconfigurable systems use simple approaches that are based on fixed module slots. The placement flexibility of these implementations is different from the flexibility assumed and analyzed in the theoretical research work. In [123] we present ways to help close this gap by showing how today's heterogeneous FPGAs can be used for dynamic reconfiguration with free module placement, varying module sizes, and multiple instances of modules.

In a tiled partially reconfigurable system as described in [87] the partially reconfigurable region is subdivided into reconfigurable *tiles*. Tiled partitioning allows for the placement of multiple PR modules with various sizes in a PR region. A reconfigurable tile can be considered as an atomic unit of partial reconfiguration. A PR region may contain several different types of tiles offering different amounts of available resources. The tile sizes may vary according to the different resource types within each tile.

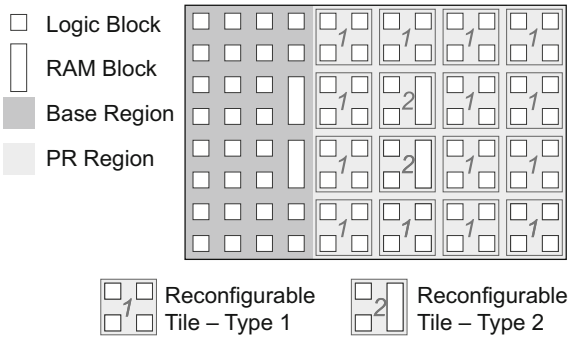

Fig. 5.58 Example of a partitioning scheme using a PR region with reconfigurable tiles

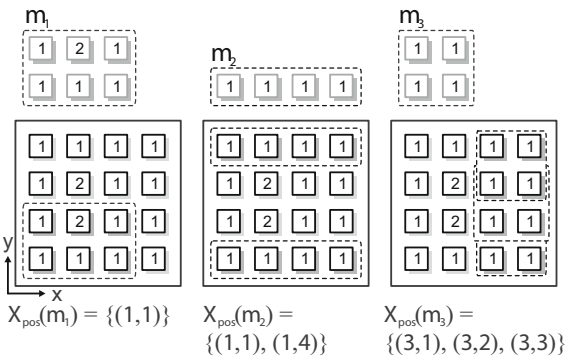

Fig. 5.59 Example of a set of PR modules and their feasible positions

Figure 5.58 shows an example with a base region and a PR region, which is partitioned into an area of 4×4 reconfigurable tiles. The PR region in the example is heterogeneous, since two different types of tiles are used. At run-time, an instance of a PR module is mapped to one or several contiguously aligned tiles. This is done by partially reconfiguring the selected tiles using the equivalent configuration data (partial bitstream) of the PR module. A PR module can occupy any size from a single tile to all tiles of the PR region. Figure 5.59 shows the PR region of Fig. 5.58 and an example of a set of PR modules with the corresponding feasible positions. The values in each tile indicate the type of the tile.

With respect to run-time placement, the PR modules vary according to their resource requirements, their shape, and their feasible positions. Each feasible position of a PR module can have a different degree of overlap with the feasible positions of the other PR modules in the system. The degree of overlap has an impact on the placeability of the PR module. Those feasible positions that overlap with many other feasible positions are likely to be blocked by a previously placed instance of another PR module. Thus a reasonable online placement policy is to always select the free position with the least degree of overlap as discussed in [121]. Besides maintaining a large number of free positions at run-time, it is also possible to optimize the placeability of PR modules at design-time. This is done by minimizing the degree of overlap of the feasible positions of the given PR modules. At design-time, the set

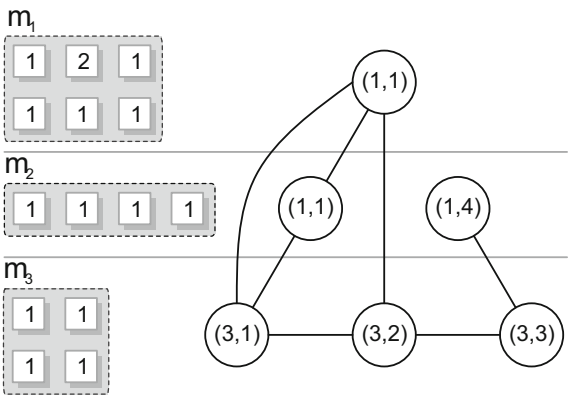

Fig. 5.60 Example of an overlap graph

of feasible positions of a PR module is defined by the shape and position of the synthesis region. The optimization of the placeability is done by selecting the synthesis regions of the PR modules that allow the best possible placement at run-time.

In order to optimize the placeability of the PR modules, a metric is required, which quantifies the degree of overlap of the feasible positions. The overlap graph $G = (V,E)$ is an undirected graph, where V are the nodes and E the edges between the nodes, that enables visualizing these resource dependencies. It shows which of the feasible positions of the PR modules overlap with each other. The graph can be used with arbitrarily shaped PR modules. For simplicity we will focus on rectangular PR modules. A vertex $v = (m,x,y) \in V$ represents a feasible position $(x,y) \in X_{pos}(m)$ of the PR module $m \in M$. The set of all vertices is defined as

$$V = \bigcup_{m \in M} \{(m,x,y) \mid (x,y) \in X_{pos}(m)\}. \tag{5.22}$$

Hence, the number of vertices is the same as the sum of feasible positions of all PR modules. For a vertex $v_1 = (m_1,x_1,y_1) \in V$ and a vertex $v_2 = (m_2,x_2,y_2) \in V$ an edge (v_1,v_2) is created, if $v_1 \neq v_2$ and the area of PR module m_1 at position (x_1,y_1) overlaps with the area of PR module m_2 at position (x_2,y_2). Figure 5.60 shows the overlap graph for the PR modules of the example in Fig. 5.59.

With the overlap graph, we can evaluate the degree of overlap for each feasible position of the PR modules. For this purpose we introduce the *position weight*. Using the overlap graph, the computation of the position weights is done in two steps. First, the *probability weights*

$$w_p(v) = p_{alloc}(m)/|X_{pos}(m)| \tag{5.23}$$

are computed for each vertex $v = (m,x,y) \in V$, where $p_{alloc}(m)$ denotes the probability of an allocation of the PR module m. The probability weight $w_p(v)$ indicates the probability of a feasible position to be chosen, if all tiles in the PR region are available and a random placement is applied.

Secondly, the position weight $w_{pos}(v)$ of a feasible position is computed by summing the probability weights of the adjacent vertices. The set of adjacent vertices V_{adj} is defined as

$$V_{adj}(v) = \{v_{adj} \mid (v, v_{adj}) \in E\}, \tag{5.24}$$

and the resulting position weight is calculated by

$$w_{pos}(v) = w_p(v) + \sum_{v_{adj} \in V_{adj}(v)} w_p(v_{adj}). \tag{5.25}$$

The position weights reflect the degree of overlap. For example, the placement of an instance of m_2 at position $(1,4)$ only blocks the position $(3,3)$ of m_3, while the placement of an instance of m_2 at position $(1,1)$ blocks the positions $(1,1)$ of m_1 and $(3,1)$ of m_3. Therefore, the position weight $5/18$ of position $(1,4)$ of m_2 is lower than the one from position $(1,1)$.

Apart from the design-time aspects, the position weight can also be used for the placement of PR modules at run-time. The placement is done by selecting the available position with the least position weight. This ensures maintaining a large number of available positions for future placements.

A metric to evaluate the degree of overlap of all feasible positions is to generate a weighted sum of the position weights of all feasible positions. As the probability weight $w_p(v)$ reflects the probability of a feasible position to be selected when randomly placing a module, the overlap weight of all PR modules is defined as follows:

$$w_{ovr}(V) = \frac{1}{|V|} \sum_{v \in V} w_{pos}(v) \cdot w_p(v) \tag{5.26}$$

The weighted mean of the position weights is divided by the total number of feasible positions $|V|$ to balance the degree of overlap and the number of feasible positions. The synthesis regions of the given PR modules can be selected in such a way as to minimize $w_{ovr}(V)$. A small $w_{ovr}(V)$ indicates that the overlaps of feasible positions of the PR modules are small. Minimizing the overlap weight aims at maximizing the number of available positions after placement of a PR module at run-time. Thus the overlap weight is a metric for the placeability of all PR modules.

5.4.4.3 Hardware-in-the-Loop

Hardware-in-the-Loop (HiL) simulations are applied in many areas of embedded systems design, but originate from control design, this is still the main area of interest. In [162], three concepts and tools are presented which allow to interface a simulation of the controller's environment (plant) to an actual implementation of the controller on an FPGA. While this usually requires a model of the plant which can be calculated in real-time, we slow the implemented controller down by exploiting special features of digital hardware. In fact, the simulation environment running the plant model gets in charge of the clock of the hardware design. With this technique we can integrate nearly any FPGA based DUT (Design Under Test) into a simulation environment like MATLAB (Simulink), CamelView [148], or ModelSim. This

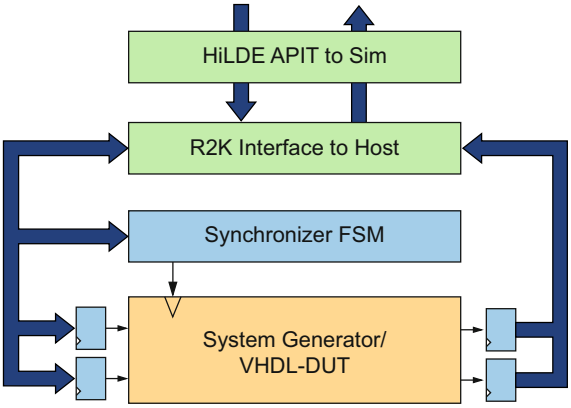

Fig. 5.61 Non-real-time hardware interface for a Hardware-in-the-Loop system

offline HiL tool flow, called HiLDE (Hardware-in-the-Loop Design Environment) allows for a functional verification of the implemented controller in real hardware, while former test benches from pure software simulations can be reused. Additionally, HiLDE can speed up simulations by several orders of magnitude, depending on the number of in- and outputs and the complexity of the user design. As soon as the DUT is embedded in its target environment, HiLDE cannot be used for testing anymore, as real-time processing is required then. For this, we developed HiLDEGART, (HiLDE for Generic Active Real Time Testing), a tool to visualize and parameterize an active controller in its real environment. Both branches of our tool flow use vMAGIC, an API for the generation and manipulation of VHDL code, to generate the required hardware interfaces as well as configuration data. In the following paragraph, the developed tools HiLDE, HiLDEGART, and vMAGIC are discussed in more detail.

HiLDE: The basic idea of our framework is the automatic integration of a DUT into a standardized hardware interface (cf. Fig. 5.61), which enables communication between a simulator and the DUT. This interface consists of a clock controller (Synchronizer) and a set of registers at the inputs and outputs of the DUT. The Synchronizer allows clock cycle accurate control over the DUTs clock by a software environment like Simulink. The input and output registers are used to transfer data between the DUT and the simulation. During a simulation, the three steps 1) write inputs, 2) do n-clock cycles, and 3) read outputs are repeated in a loop controlled by the simulator. The data transfer from the simulation to DUT is done with the Rapid Prototyping Platfrom RAPTOR (e.g. the R2K, cf. Sect. 5.4.5.1). In general, the clock speed in HiLDE simulations will be much slower than the desired clock Speed of the DUT (non-real-time) because the computation of a simulation step of the test-bench or plant-model in the simulator typically takes a lot of time. However, the overall simulation can become faster by several orders of magnitude, if a DUT is moved from the simulation towards hardware.

HiLDEGART: After a design has been successfully tested in the HiLDE environment, it can be integrated into its target environment, where it works in real-time.

Still, it is desirable to monitor the controller's IOs as well as its internal states for further testing under real-life conditions. To meet these requirements, HiLDEGART generates hardware interfaces capable of recording signals in real-time without additional external hardware such as logic analyzers. The idea is basically the same as for HiLDE: a hardware wrapper is generated which adds memories to the signals in question and connects those to a bus interface. This enables a GUI to access and display the recorded values. As the output data rates can be much higher than the available communication bandwidth between hardware and software, resampling units and FIFOs are used instead of registers. In addition to that, registers are connected to inputs which parameterize the DUT (such as constants of a controller), so that users can change those values from within the HiLDEGART GUI. Furthermore, the GUI offers advanced features like a triggering unit, which casts events based on boolean operations on the IO signals. This facilitates for example, to increase the resampling rate once a signal reaches a critical level.

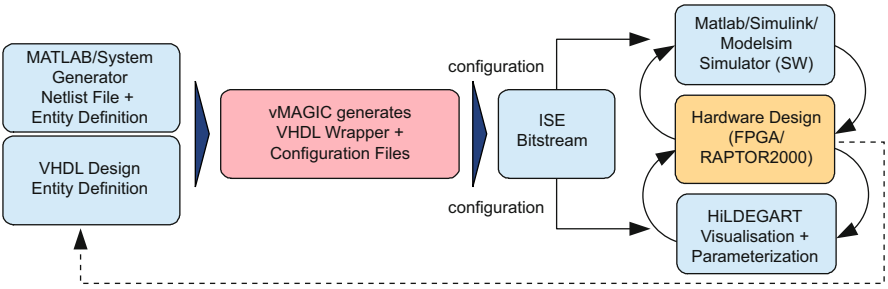

Fig. 5.62 Design Flow for HiLDE and HiLDEGART using vMAGIC

vMAGIC: The interfaces described in the previous sections are DUT specific and have to be adapted to each new design, which is a tedious and error prone task. As the basic structure stays the same between all implementations, an API was developed, which enables users to write scripts that automatically generate interfaces like these, or automate any other recurring task based on VHDL code as depicted in Fig. 5.62. In [163], the implementation details of vMAGIC are presented.

5.4.5 Platforms and Applications

Two main platforms that support dynamic reconfiguration are used in this Section: RAPTOR and BeBot. In the process of developing microelectronic systems, a fast and reliable methodology for the realization of new architectural concepts is of vital importance. Prototypical implementations help to convert new ideas into products quickly and efficiently. Furthermore, they allow for the parallel development of hardware and software for a given application, thus shortening time to market. FPGA-based hardware emulation can be used for functional verification of new MPSoC architectures as well as for HW/SW co-verification and for design-space

exploration. The rapid prototyping systems of the RAPTOR family [164] provide the user with a complete hardware and software infrastructure for ASIC and MPSoC prototyping (cf. Sect. 5.4.5.3). A distinct feature of the RAPTOR systems is that the platform can be easily scaled from the emulation of small embedded systems to the emulation of large MPSoCs with hundreds of processors. Along with rapid prototyping, the system can be used to accelerate computationally intensive applications and to perform partial dynamic reconfiguration of Xilinx FPGAs, as presented in Sect. 5.4.5.1.

The BeBot miniature robot, which will be discussed in detail in Sect. 2.2, integrates an embedded processor and a dynamically reconfigurable FPGA (Xilinx Spartan-3). An example for a vision processing application utilizing this architecture will be discussed in Sect. 5.4.5.2.

5.4.5.1 Dynamic Reconfiguration of FPGAs on the Rapid Prototyping Platfrom RAPTOR

The RAPTOR systems follow a modular approach, consisting of a base system and up to six daughterboards. The base system comprises the communication and management infrastructure, used by the daughterboards, which realize the required application-specific functionalities. Because of the modular design, the user can easily integrate new FPGA technologies or communication facilities by means of additional daughterboards. The RAPTOR base system can be integrated into a host PC or run as a stand-alone system. The optional host system can be used to ease monitoring and debugging. For communication with the host system, the RAPTOR-X64 base system integrates a PCI-X and a USB-2.0 interface. The board can be operated outside the normal PCI environment by utilizing the USB-2.0 interface. It is also possible to integrate a PCI-Express-based host-connection by replacing the RAPTOR-X64 by the RAPTOR-XPress baseboard.

The Local Bus and the Broadcast Bus, which are provided with the RAPTOR base system, offer powerful communication infrastructures and guarantee a high-speed communication with the host system and between individual modules. Additionally, direct links between neighboring modules can be used to exchange data with high bandwidth and low latency. Furthermore, all FPGA modules provide additional high-speed serial links for communication between the modules. Reconfiguration (including dynamic reconfiguration) is performed with the maximum possible bandwidth that the FPGAs support. The RAPTOR systems provide a direct migration path from FPGA-based prototypes to ASIC realizations by simply replacing the FPGA based daughterboards with daughterboards that integrate the developed ASICs. Daughterboards integrating different MPSoCs (discussed in detail in Sect. 5.4.5.3) have been realized and can be integrated together with additional dynamically reconfigurable FPGA modules. In this way, information processing systems can be realized that combine the advantages of dynamically reconfigurable FPGAs and MPSoCs.

Figure 5.63 gives an overview of an architecture that has been developed for the realization of self-optimizing drive controllers for a permanent magnet servo

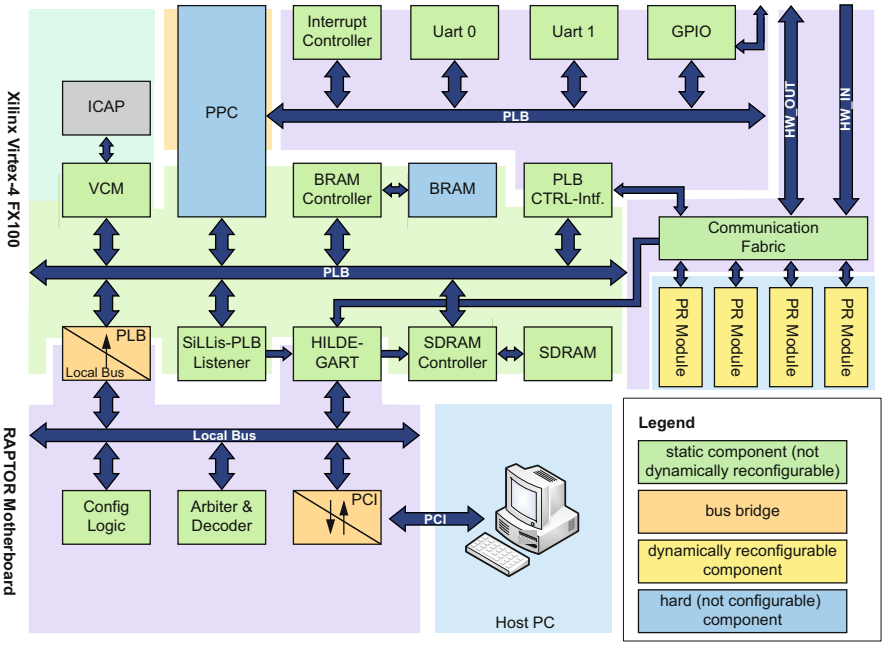

Fig. 5.63 System architecture for the implementation of a self-optimizing drive controller based on the RAPTOR prototyping system

motor [159]. The implementation is based on the methods for dynamic reconfiguration that have been previously described and is realized on a Xilinx Virtex-4 FX100 FPGA. The architecture is composed of an embedded PowerPC processor connected to dynamically reconfigurable resources (*PR Module*). A processor local bus (PLB) enables communication to the local bus of the RAPTOR system, and from there to the host PC. The dynamically reconfigurable PR modules are used to implement controllers or signal conditioning blocks, since these elements are exchanged according to the current state of the plant and the current objective of the system. The reconfiguration is performed by the Virtex Configuration Manager (VCM) [88].

A program running on the PowerPC initiates the reconfiguration based on a continuous evaluation of the control quality and realization effort, indicating the memory space from the external SDRAM where the partial bitstream ought to be copied. The partial bitstream contains only the needed configuration for one PR module. When a reconfiguration is requested, the VCM initiates DMA transfers from the SDRAM controller, loads the requested partial bitstream to the target PR module by accessing the Internal Configuration Access Port (ICAP), and sends an interrupt to the PowerPC when done. The reconfiguration process lasts about 4.38ms, which represents several control cycles. To overcome this, an initialization routing is used to calculate the initial states of the new-loaded controller. A supervising program, running in the PowerPC, is in charge of monitoring system activity and triggering the dynamic reconfiguration. For the verification of the implemented control

algorithms and for testing the correct behavior of the system during dynamic reconfiguration between different controller implementations, we have used the HiL environments HiLDE and HiLDEGART as described in Sect. 5.4.4.3.

5.4.5.2 Dynamic Reconfiguration of FPGAs on the Miniature Robot BeBot

The hybrid processing architecture of the BeBot miniature robot consists of a main processor complemented by an FPGA device that offers on-demand parallel processing with the major advantage that the FPGA can be dynamically reconfigured during runtime to optimally utilize the hardware resources and the energy budget. In contrast to other reported approaches on dynamic hardware reconfiguration, for example [23, 38, 145], we focus on a concept that automatically and dynamically allocates hardware resources depending on the current status of the robot, the required tasks, and the context of operation [149]. Processes can be executed in software on the processor or as modules on the FPGA using partial dynamic reconfiguration. The reconfiguration process is managed by the robot's operating system. The access to the hardware is, from the application point of view, transparent.

Utilizing dynamically reconfigurable hardware for image processing instead of a pure software solution enables real-time image processing and significantly reduces the required computing power of the CPU. Instead, the CPU can be used, e.g. for sensor fusion tasks, behavior generation, and communication within the wireless network. Depending on the current context, hardware configurations can be automatically loaded into the FPGA device, that is, one or more hardware modules are able to process images in parallel. If a specific processing task has been finished, a new hardware configuration can be loaded to optimize the resource utilization on-the-fly.

An initial configuration of the FPGA is loaded after booting the robot. Typically, the local flash memory of the robot is used to store the configuration data. But it is also possible to load the configuration via one of the available wireless communication links. This feature is very useful in multi-robot applications in order to share available processing resources of the robot team by wirelessly transmitting FPGA configuration data to robots that are able to offer computing resources to other robots. This type of resource sharing between wirelessly connected robots, requires additional operating system services, as discussed, e.g. in [80]. In the context of image processing, this can significantly reduce computation time and increase the throughput of images. Reconfiguration of the complete FPGA, requires loading a bitstream file of 728 kByte. On the BeBot, a complete FPGA reconfiguration is performed in 25 ms, corresponding to a reconfiguration rate of 30 MByte/s. If slow communication interfaces are used to transfer the reconfiguration data to the robot, the configuration files can be cached in the internal SDRAM.

Utilizing the INDRA design flow (cf. Sect. 5.4.4.1), partial dynamic reconfiguration of the FPGA can be used to reduce the reconfiguration time. Furthermore, with this concept, it is possible to keep parts of the application and the application data inside the FPGA, essentially reducing communication time. Here, the FPGA is divided into two parts: a static region utilizing 20% of the FPGA resources (slices)

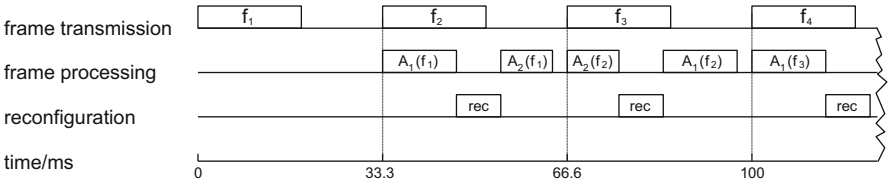

Fig. 5.64 Dynamic reconfiguration on a frame-by-frame basis

and a dynamically reconfigurable region, comprising 80% of the available FPGA slices. During dynamic reconfiguration, the base region remains unchanged while the partially reconfigurable region is completely reconfigured. Here, partial reconfiguration can be performed in 20 ms if the bitstream is available locally on the BeBot.

Depending on the trigger for dynamic reconfiguration, a differentiation can be made between time-driven reconfiguration and event-driven reconfiguration. In time-driven reconfiguration, the time and order in which PR modules are loaded is known at design-time and do not change at run-time. The reconfiguration controller can be a simple state machine that triggers the reconfiguration at the predefined time intervals. On the one hand, the time between two reconfigurations can be orders of magnitude higher than the reconfiguration time, if complex applications are executed in turn. An example would be changing between two video processing algorithms every 10 seconds. On the other hand, fast partial reconfiguration enables hardware changes at high frequency: in video processing, time-driven reconfiguration can be triggered on a frame-by-frame basis. Figure 5.64 gives an example, where two applications (A_1 and A_2) are processing consecutive data frames ($f_1, f_2, ...$). It has to be assured that the sum of reconfiguration time and application execution times are lower than 33 ms for the 30 frame per second on BeBot, i.e. 13 ms are available for application execution for the used partitioning, which requires 20 ms for reconfiguration. In the example, the applications are decoupled from the data transmission from the camera since all executions are performed at the previous frame. If this is not possible, a more complex scheduling is required, and application execution typically starts in parallel to data transmission to increase performance.

In the event-driven scheme, the reconfiguration time and the order of the PR modules are not known at design-time. A trigger for dynamic reconfiguration can occur at any time. Changes of the ambient light could, e.g. be used to reconfigure between different video processing algorithms. While the tool flow and the hardware infrastructure are identical for event-driven and time-driven reconfiguration, the implementation of the reconfiguration controller varies. For time-driven reconfiguration, the reconfiguration controller can be realized by a simple timer. In event-driven reconfiguration various internal and external parameters may have to be taken into account to decide when and which PR module to load. On the BeBot the trigger for reconfiguration is set by a software implementation on the internal processor. In both schemes, time-driven and event-driven reconfiguration, the calculation times

of this software part of the reconfiguration controller are negligible compared to the FPGA reconfiguration time.

Two different hardware modules have been developed in the context of vision processing. The first provides optical flow motion detection and the second supports color recognition [4, 35, 52]. The optical flow calculation is used to detect walls or obstacles in the operational area and to dynamically construct a map of the environment. The color detection algorithm enhances options to identify objects like marked landmarks or other robots, in order to improve navigation and to map buildings. With the dynamic approach, both algorithms can be processed on the robot platform under real-time constraints achieving a good utilization of the processing devices.

To evaluate the performance of the hardware implementation of these two algrotihms, the BeBot miniature robot prototype has been evaluated in a test room with artificial lighting and a convenient environment. A frame size of 160x120 was chosen, requiring 38,400 Bytes to be transferred from the camera to the SDRAM. The frame rate is fixed to 30 frames per second by the camera used. Therefore, the bandwidth required to transfer the image data into memory is less than 1.2 Mbyte/s. Since the SDRAM can be accessed with more than 80 MByte/s and the FPGA implementation achieves about 46 MByte/s, sufficient bandwidth is available in the system to transfer data between the system components. Performance is mainly limited by the processing time on the FPGA. Wherever possible, communication and calculation are performed in parallel, i.e. calculation starts directly after receiving the first data from the camera, or one frame is processed while the next frame is loaded.

The optical flow does not need any particular parameter updates when the environment is changing, except for the number of columns and the speed threshold. These parameters do not affect the performance nor the area usage of the FPGA-based implementation. A single module implementation of the optical flow requires 1338 slices (18% of the resources available for the partially reconfigurable module) and one frame is processed in 0.9 ms. Processing time includes the time for reading the image data from the SDRAM and for writing the results to the processor.

In contrast, the required FPGA resources and the computation time for the color recognition module can vary significantly depending on the number of maximum recognizable blocks and on the number of colors. In the following paragraph, an analysis of the different configurations on BeBot will be presented, focusing on area consumption and computation time. For the evaluation of the block recognition module, various configurations have been tested. Table 5.4 shows the configurations chosen for a laboratory test. The resource requirements of the PR modules are given in the column *Used slices*. The *Utilization* represents the percentage of slices in the PR modules that are utilized by the implementation. Configurations that could not be realized on the Spartan-3 FPGA because of the resource limitations of the PR-modules are marked with an *X*.

In Table 5.4 the execution time and the amount of frames per second are calculated starting from the write command for the first pixel of the frame to the final

Table 5.4 Configuration test parameters for color recognition on BeBot

N colors	N blocks	Used slices	Utilization	Execution time [μs]	Frame/sec
1	1	1250	11%	783.83	1275
1	2	1363	12%	784.13	1275
1	4	1898	16%	785.20	1273
1	8	2547	22%	810.60	1233
1	16	3867	33%	5281.64	189
1	32	6794	58%	20450.22	48
2	1	2351	20%	783.83	1275
2	2	2912	25%	784.13	1275
2	4	3672	31%	785.20	1273
2	8	5061	43%	810.60	1233
2	16	7816	67%	5281.64	189
2	32	X	X	X	X
4	1	5174	44%	783.83	1275
4	2	6165	53%	784.13	1275
4	4	7434	64%	785.20	1273
4	8	9846	84%	810.60	1233
4	16	X	X	X	X
4	32	X	X	X	X

interrupt provided by the module. At that time the results of the computation are already stored in the output FIFOs.

Reconfiguring between optical flow and color recognition requires 20 ms. The camera sends data with 30 frames per second, which results in 33 ms for calculation. Since the optical flow is calculated in 0.9 ms, 12.1 ms are available for color recognition if dynamic reconfiguration on a frame-by-frame basis is performed (cf. Fig. 5.64). Hence, optical flow and color recognition for up to 16 blocks can be performed virtually in parallel without frame-loss by dynamically reconfiguring between two frames.

The proposed hardware implementations of the vision algorithms on the BeBot miniature robot platform show that real-time image processing is possible even on platforms with limited processing capabilities. The use of FPGA-based hardware releases the processor from these very computational intensive tasks. Additionally, dynamic reconfiguration can be used to switch between different applications or to modify the elaboration parameters at run-time.

5.4.5.3 Dynamic Reconfiguration of Multi Processor System on Chip

In addition to fine-grained FPGA-based architectures, coarse-grained architectures are evaluated. In this context, we focus on on-chip multiprocessors that integrate mechanisms for dynamic reconfiguration with minimum area overhead. In general, our MPSoC system comprises of a generic and hierarchical architecture, so

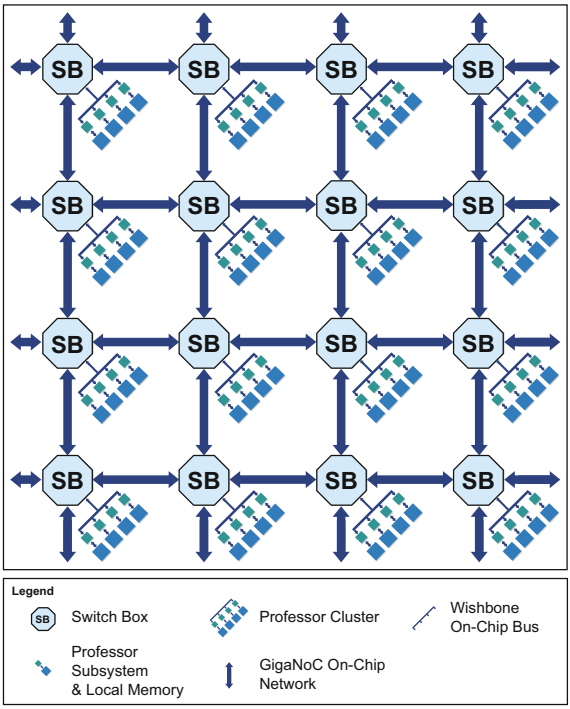

Fig. 5.65 MPSoC architecture comprising the GigaNoC communication infrastructure

that the system can be configured for different application scenarios at design-time. Figure 5.65 depicts the MPSoC architecture proposed in [150], consisting of the *SoC level*, the *cluster level*, and the *processor level*. At the SoC level, a variable number of cluster components is connected via a network-on-chip communication infrastructure. Due to the homogeneous structure, the MPSoC system can be scaled to meet the performance requirements of various application domains. While the NoC provides the communication backbone for propagating data, at cluster level, this data is processed by a reconfigurable multiprocessor system. Via an on-chip Wishbone bus, the processor elements of each cluster can communicate locally and can access shared memory. A single processor element represents the lowest level of hierarchy of our MPSoC architecture.

GigaNoC is our hierarchical and scalable NoC communication infrastructure, which is especially suitable for multiprocessor SoCs [109, 170]. The GigaNoC architecture is depicted in Fig. 5.65. The switch boxes (*SB*) represent the core components of the NoC and act like high-performance routing nodes that propagate the data through the on-chip network. GigaNoC comprises of packet-switching [107] and each packet is divided into smaller fragments, called flits. In order to support arbitrary network topologies with different connectivity, the number of communication ports for each switch box can be configured during design-time. For the mesh topology, depicted in Fig. 5.65, every switch box has four external and one internal communication port, which connects the processor cluster to the NoC.

While the number of communication channels per switch box is chosen at design-time, the NoC topology and the routing strategies inside the switch boxes can be adapted at run-time. Several possible routing schemes are integrated into the hardware description of the switch box IP-core. A pre-selection can be made at design-time; at run-time the user or the operating system can easily switch between the integrated routing schemes by using special command flits. These command flits are also used to disable single malfunctioning embedded processors or complete processor clusters. In this case, the routing is automatically adapted to changes in the architecture.

QuadroCore Multiprocessor Cluster

Due to the generic implementation of the internal communication port, arbitrary processor cores and processor clusters can be attached to the NoC. Figure 5.65 depicts an example configuration, where clusters of four N-Core processors are attached to each switch box. N-Core is a 32-bit RISC microprocessor, which was developed in our group as a softmacro that can be easily adapted to the needs of specific areas of application [150]. N-Core has a common load/store architecture with a three-stage pipeline, which delivers reasonable performance for embedded systems.

The cluster organization based on the N-Core processor elements represents a typical MIMD multiprocessor cluster. In order to optimize flexibility and fault tolerance, a fast reconfiguration mechanism with low overhead has been added to the processor cluster, resulting in the run-time reconfigurable multiprocessor cluster *QuadroCore*. Without altering the instruction set architecture of the processors, run-time reconfigurability has been introduced by adding intra-processor interconnects to adapt the architecture in terms of synchronization, communication, and the degree of parallelism [105]. Figure 5.66 depicts the base architecture of QuadroCore and two typical configurations. Each of the four processors has its own local register file and instruction and data memory. Exchange of register contents between the four processors is achieved via a shared register file. Large amount of data sharing is possible via external shared memory, accessible by a shared bus.

The decision of altering the existing structure is driven by a special reconfiguration instruction, that has been added to the instruction set of the N-Core processors. This mechanism enables a quick, single-cycle run-time reconfiguration, i.e. very low overhead in terms of time required to reconfigure the resource connectivity. The reconfiguration instructions can be embedded into the normal program code by the programmer or by an optimizing compiler; no additional memory is required to store the configuration data, like in FPGAs.

In the proposed implementation, reconfiguration requires an alteration in the interconnection of the various building blocks inside and between the processors. Currently, capabilities for reconfiguration have been added between the decode & execute stages, between the execute & register read/write stages, and between the processors and the shared memory. In QuadroCore, the reconfigurable interconnect is realized by means of additional multiplexers that have been integrated into the architecture. The reconfiguration instruction provides the configuration information

to determine the functionality of the reconfigurable interconnects between the intermediate stages of the instruction pipeline, i.e. the control signals of the added multiplexers.

As mentioned, the QuadroCore cluster can be dynamically reconfigured with respect to three main features: synchronization, communication, and parallelism, which are briefly described in the following paragraphs.

Synchronization: Depending on the amount and frequency of inter-processor data exchange, the processors in the cluster can operate synchronously at instruction level or asynchronously. The cluster can be adapted according to the application characteristics during run-time, since both a fine-grained synchronization scheme (for instruction-level parallelism) and a coarse-grained independent operation (for task-level parallelism) are supported. The run-time change in synchronization is achieved by introducing a synchronization instruction between parts of the application where a change in application characteristics is determined during compilation. The synchronous mode ensures a lock-step operation while the asynchronous mode initiates a barrier synchronization for every inter-processor data exchange.

Communication: The communication between the processing elements is mainly categorized in terms of frequency of data exchange and amount of data exchange. To suit applications where exchange of data such as register contents is frequent, a shared register file provides a quick data-exchange mechanism. For large amount of data, the shared memory is accessible via arbitration over a common bus. The shared register file has a round-trip time (write and read) of 4 clock cycles, whereas the shared memory has a variable access time between 5 to 12 clock cycles for each access. Furthermore, register sharing among processors is possible on account of the reconfigurable interconnect introduced between the ALUs and the register files as depicted in the right configuration in Fig. 5.66. For applications with high register pressure, registers from the neighboring processors can be utilized (supported by the compiler).

Parallelism: The choice of data-parallel or task-parallel behaviour steers architectural characteristics. The Multiple Instruction, Multiple Data (MIMD) mode, allows asynchronous operations on independent data and instruction streams. The Single Instruction, Multiple Data (SIMD) mode, as illustrated in the middle configuration in Fig. 5.66, co-ordinates all the four data-paths with a single instruction stream, thus saving energy via reduced memory interactions.

As mentioned above, the reconfiguration can be easily controlled by the user by adding simple reconfiguration instructions into the C code. A smarter way of introducing reconfiguration is to integrate the decision process into the compiler. As detailed in [105, 168], the choice of the best mode of execution can be made using standard program analysis techniques. For every basic block, our compiler, called COBRA (Compiler-Driven Dynamic reconfiguration of Architectural Variants), determines the best possible mode during compilation and a reconfiguration is inserted between the modes. The reconfiguration overhead is kept as low as possible since the reconfiguration between modes requires only a single clock cycle [167, 169].

CoreVA VLIW Processor

As an alternative to the QuadroCore architecture, we have developed a VLIW (Very Long Instruction Word) processor, especially suited for signal processing applications, called CoreVA [110]. Resource efficiency together with high flexibility were the main design goals for the processor implementation. The CoreVA architecture is a modular soft-core design, which can be configured at design-time with respect to the number of functional units (e.g. ALUs, Multiply-Acummulate (MAC) units, division step units), the width of the data paths, and the structure of forwarding circuits. In the default configuration, the CoreVA architecture represents a 4-issue VLIW architecture, implemented as a Harvard Architecture with separated instruction and data memory and six pipeline stages.

The operations follow a two- and three-address format and are all executed in one clock cycle. Most instructions have a latency of one clock cycle, except branch, MAC and load operations, which have a latency of two clock cycles. In SIMD mode, two 16-bit words can be processed in each functional unit, which leads to an eight-fold parallelism. As a first prototype, the CoreVA architecture has been fabricated in a 65nm STMicroelectronics technology. The CoreVA system (including level-1 cache and several dedicated hardware extensions) operates at a clock frequency of up to 285 MHz with a power consumption of about 100mW. The chip area is about 2.7sqmm including 32 kByte level-1 cache for instruction and data.

With respect to adaptability of the architecture, two mechanisms have been integrated to enhance the efficiency of the architecture: dynamic bypass reconfiguration and dynamic voltage and frequency scaling using a specially designed subthreshold standard cell library. The bypass reconfiguration exploits the fact that many paths of the integrated bypass are rarely used for certain applications. Therefore, the user can change between application-specific bypass configurations at run-time. Depending on the actual implementation, this leads to a reduction of the critical path by 26%, which can be used to dynamically increase the clock frequency or to decrease power consumption.

The next generation of CoreVA processor has been realized utilizing a specially developed ultra-low power standard cell library. The new processor, CoreVA ULP, is capable of dynamically adapting its operating parameters according to application requirements and environmental conditions at run-time [138]. During times of low processor load, power dissipation is substantially reduced by operating the processor in subthreshold mode. A chip containing two CoreVA ULP processors was fabricated in an STMicroelectronics 65 nm CMOS technology and has been successfully tested. At 1.2V, the average energy dissipation of a single-slot processor core is 110.22pJ (at a clock frequency of 100 MHz). The minimum energy point of 9.94pJ occurs at 325mV, i.e. energy savings of 11.1 % can be achieved during subthreshold operation. The average clock frequency at this point is 133kHz.

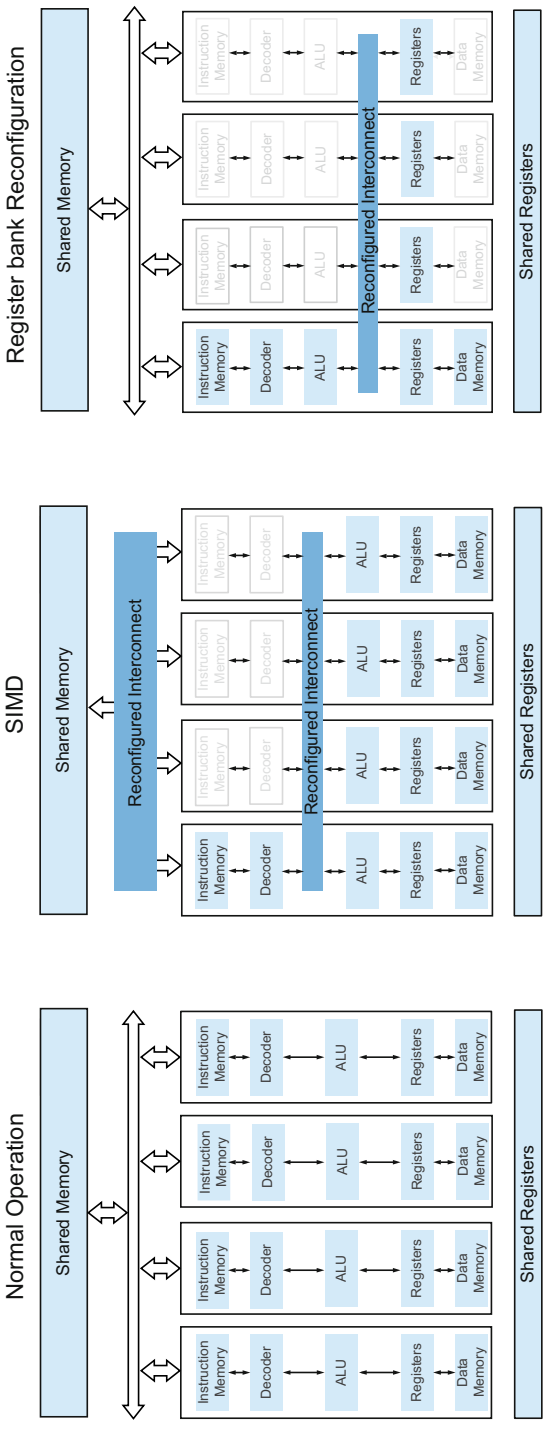

Fig. 5.66 QuadroCore enables switching between different modes of operation at run-time

5.5 System Software

Stefan Groesbrink, Simon Oberthür, and Katharina Stahl

System software encompasses all software approaches that interconnect the application software layer and the hardware layer. This includes operating systems as well as other middleware approaches, e.g. virtualization.

In the context of self-optimizing mechatronic systems, the system software provides an execution platform for self-optimizing applications that run on online-reconfigurable hardware. The ability to cope with the dynamically changing requirements from either the software or the hardware and the capability to adapt to these changes at run-time is a prerequisite for system software being applied on self-optimizing mechatronic systems. Hence, the system software must implement self-optimizing methods by itself. Self-optimization in system software is thereby not restricted to reacting to dynamical changes. The system software may also implement methods that can be applied to self-optimize the performance of the execution platform concerning e.g. resource utilization.

The functionality the system software offers is strongly coupled to the requirements related to it. Referring to the overall design and development of the self-optimizing mechatronic system, the general requirements on the self-optimizing system software are defined by the principle solution (cf. Sect. 3.3.3). One example for a general requirement is the optimization of resource allocation by exploiting unused (however reserved) resources. This requirement is solved by means of the flexible resource management which is presented in Sect. 5.5.2. Another example for a general requirement on the system software is to support run-time dependability. The problem of run-time dependability is addressed by two different operating system service approaches, one is testing the application state by using an online model checking while the other one is inspired by artificial immune systems that tries to identify system behavior anomalies. However, dependability of self-optimizing mechatronic systems is it's own subject. These operating system services have been introduced in detail in [69, D.o.S.O.M.S. Sect. 3.2] .

General requirements express general properties of the system software, that is implemented in the form of an OS kernel module, an OS service, or a separate middleware layer.

However, being the interconnecting execution platform, there is a strong interdependency between the system software and the application software layer as well as between the system software and the reconfigurable hardware. Specific requirements arise from this interconnection in terms of provided interfaces or services that are required by the applications or on the other hand in providing abstractions of hardware implementation in order to encapsulate a change in the hardware configuration. Both, the general requirements on the system coming from the principle solution and the specific requirements arising from the software and hardware have to be satisfied by the system software so that an adequate platform for self-optimizing mechatronic systems can be ensured. We assume the self-optimizing system software to be composed of reusable components (cf. Sect. 3.3.3) that are

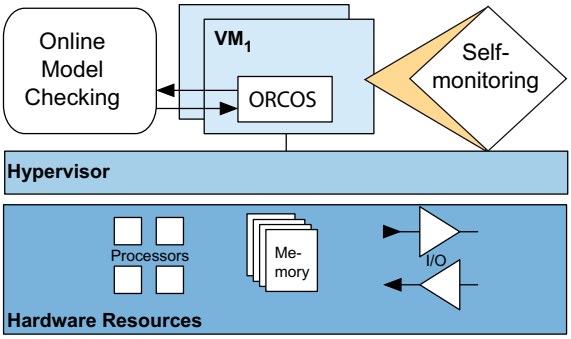

Fig. 5.67 Self-optimizing system software layers

activated with respect to the present system requirements. Each system component addresses a specific system function or system property.

We separate the self-optimizing system software into two different layers: one containing the self-optimizing real-time operating system ORCOS and the lower level layer containing the virtualization platform including the hypervisor named Proteus. This section presents the self-optimizing real-time operating system ORCOS and the self-optimizing virtualization platform. Fig. 5.67 shows an overview of the system software architecture. In addition, figure Fig. 5.67 also illustrates how the operating system ORCOS integrates the methods for *Online Model Checking* and for *Self-Monitoring* which addresses the system's dependability.

Considering the **real-time operating system** (RTOS) of a self-optimizing mechatronic system, it has to be able to cope with dynamically changing system behavior and hence dynamical changes on the requirements of the platform. We distinguish between external and internal requirement changes. External requirement changes are those originating from outside the system software. That means changes in the requirements either from the software or the hardware layer. For example, due to a software reconfiguration, an OS service will be required that has not been provided by the operating system before. Another scenario might be that the implementation strategy of an OS service must be exchanged based on the new software configuration. The biggest challenge to cope within such a system is that not all system parameters are determined during design-time and therefore have to be identified during run-time. Resulting from this, the OS has to provide an interface that allows to signify the changes in requirements. Furthermore, to be able to satisfy these changes, the operating system must also provide alternatives in implementation. And last but not least, the operating system needs structures to enable the run-time activation or online exchange of components.

Internal requirements come from the operating system itself. As being a self-optimizing operating system, it is equipped with desired objectives and optimization criteria. Usually, the operating system's objective is to optimally manage the application's task and the resources, e.g. in terms of resource consumption, memory management, scheduling strategies, etc. To achieve those requirements, the operating system requires internal structures to monitor and analyze the performance and to verify whether the optimization objectives are fulfilled. Obviously, all

self-optimization efforts of the operating system must be performed under considerations of real-time constraints.

The **real-time operating system ORCOS** [59] was first designed to be fully customizable during design-time. However, for the purpose of online self-optimization, we needed an operating system that is flexible and can be extended during run-time. Therefore, we adjusted the architecture of ORCOS to enable information processing required to enable self-optimization. We will describe the resulting OS architecture in Sect. 5.5.1. As a basis for run-time reconfiguration, we use the concept of our Profile Framework, described in Sect. 5.5.1.1, that allows alternative implementations for applications task as well as OS components.

Modern self-optimizing mechatronic systems have highly dynamic resource consumption. One of the main objectives of a real-time operating system is to optimize resource management. Common real-time systems and middleware software are fixed and not optimal for such scenarios. A problem with dynamic real-time applications using common real-time system software is that applications allocate resources up to their maximum requirements. On the one hand, this allocation behavior guarantees that the applications have all resources being required during execution. On the other hand, the maximum resources are often required only in the worst case and are mostly unused. An approach for optimizing the dynamic resource consumption is presented by applying flexible resource management. We developed a *Flexible Resource Manager* (FRM) (cf. Sect. 5.5.2) that allows to optimize resource consumption autonomously. It uses the Profile Framework as the basis for alternative resource requirements of applications. The FRM optimizes resource consumption in such a way that it allows temporal usage of resources that are reserved by other applications for worst-case. However, this overallocation of resources is conducted in a safe manner as the FRM guarantees a reconfiguration of the system without violating worst-case deadlines.

An additional implementation technique for self-optimization – especially in terms of resource utilization – is our **virtualization platform**. Like any kind of virtualization our approach provides strict separation of hardware resources by means of a hypervisor. A two-level scheduling (hypervisor and RTOS) has been designed in such a way that real-time aspects are strictly taken into consideration. The FRM approach has been extended in such a way that now a two-level FRM (integrated in hypervisor and RTOS) is implemented and dynamic reconfiguration may even happen across virtual machines (cf. Sect. 5.5.4). Some further interesting aspects concerning enhancing the system dependability by virtualization are illustrated in more details in [69, D.o.S.O.M.S. Sect. 3.2] .

5.5.1 Architecture for Self-optimizing Operating Systems

As the first step, we developed a real-time operating system named **ORCOS** (**Organic Re-Configurable Operating System**, see [59]) that allows a fine-grained configuration of the basic (functional) OS components according to the given requirements. According to the definition of self-optimization (cf. Sect. 1.2), the

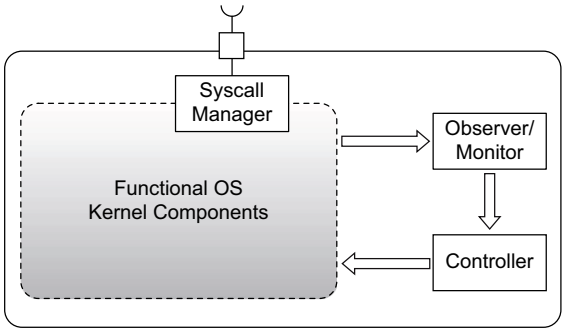

Fig. 5.68 Architecture for a self-optimizing operating system

workflow of a self-optimizing system includes the following steps: Analysis of the current situation, determination of objectives and adaptation of the system behavior. Hence, the self-optimizing real-time operating system must provide mechanisms and structures to enable the implementation of a self-optimization workflow.

The architecture of the RTOS is required to be extended in order to support self-optimization and adaptation to the changing requirements of the self-optimizing software and hardware. We adjusted the OS architecture based upon the Observer-Controller Architecture that was first instantiated by the Organic Computing Initiative [179]. An Observer and a Controller component extend the OS and build up the basis of self-optimization and enable monitoring, self-reflection and reconfiguration in the real-time operating system. The resulting architecture of the operating system ORCOS is presented in Fig. 5.68. These new components are integrated into ORCOS as configurable kernel components but separated from the functional OS kernel modules. The Observer is responsible for monitoring and data collection, and analyzing and evaluating system behavior based on the defined system policies and objectives. Self-optimization in the operating system is triggered by a reconfiguration that is initiated by the Controller as a reaction on the evaluation procedure results. The ORCOS Profile Framework builds up the basis for reconfiguration in the operating system, as it offers alternative implementations defined within a profile from which the Controller can select.

5.5.1.1 Reconfiguration Framework

The central component for the run-time reconfiguration of the operating system is build up by the **Controller**. It is responsible for initializing a reconfiguration on the basis of the Profile Framework [156] provided by ORCOS.

Originally, it has been developed in the context of the Flexible Resource Manager (FRM) [154] to self-optimize resource consumption in resource restricted real-time systems as it allows for alternative implementations of an application task in terms of resource requirements.

The Profile Framework follows the following principle: at each point of time exactly one profile of a task is active (cf. Fig. 5.69). A configuration c of the system

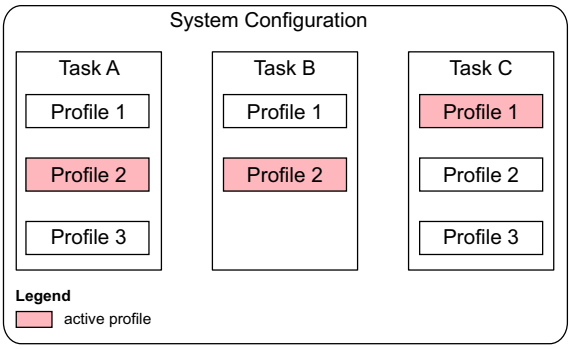

Fig. 5.69 Profile Framework: Example for a system configuration

is defined as configuration $c = (p_1, p_2, \ldots, p_n)$, with n being the number of running tasks τ and $p_1 \in P_1, p_2 \in P_2, \ldots, p_n \in P_n$ and P_i being the profile set of task τ_i. Each task must define at least one profile to be executed. For any task there may be available multiple profiles, i.e. versions with different parameters concerning nonfunctional properties. Selecting a specific profile is completed due to dynamic decisions at run-time.

We enlarge the concept of a profile to be applied to any reconfigurable OS component. This encompasses all system entities:

- application tasks
- OS kernel components and services
- components of the self-optimizing framework (e.g. Observer consisting of a reconfigurable Monitor and Analyzer described before)

Profiles may differ concerning their resource demands, which resources are applied (e.g. a specific communication resource), the implemented algorithm (e.g. in terms of accuracy of the algorithm or the strategy), execution times, deadlines etc. A prerequisite for identifying a component to be a reconfigurable component is the existence of alternatives, which in fact means the definition of at least two different profiles. Applying profiles to all OS components and the applications running on the system, all system parts become (re)-configurable online. In this case we can really speak about fully realizing a self-optimization operating system.

For reconfiguration the Controller contains policies, restrictions and thresholds for decision making. Of course, a decision for reconfiguration must be checked against the system characteristics and the real-time requirements of the application tasks. A reconfiguration must not harm the system service delivery and guarantee the compliance with the task's real-time deadline. If all the conditions are met, the Controller performs a reconfiguration of the system at run-time by simply switching between the profiles.

Observer

The Observer component is responsible for (1) identifying and selecting the appropriate information and (2) analyzing them in order to make conclusions on future

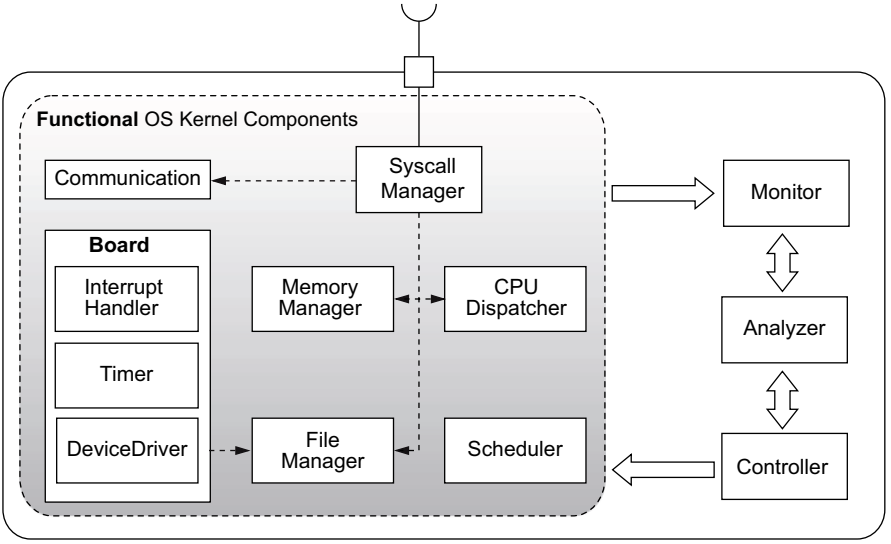

Fig. 5.70 ORCOS Architecture integrating Monitor, Analyzer and Controller.

system states. For our purpose, we subdivide the observer into two separate entities: a Monitor and an Analyzer (cf. Fig. 5.70). Although, these two entities are strongly coupled (the monitor only collects the data that is required by the analyzing method) they exhibit self-contained tasks which can be executed in a timely decoupled manner. Due to the ability of online reconfiguration we introduce an additional interconnection between the components: The strategy of the analyzer is reconfigurable at run-time and can be exchanged by the controller. The data collected by the monitor depends on the applied analyzing algorithm so that an exchange of the analyzing strategy in turn has effect on the data aggregation of the monitor.

The monitor collects the behavioral data and aggregates the data for the analyzer while the analyzer evaluates the data and passes its evaluation results to the Controller. In order to collect data about the application behavior, we can monitor the SyscallManager interface as it is the only interface through which a task can interact with the kernel. Integrated into the SyscallManager, the Monitor is able to intercept and record all the system calls from the tasks.

The idea is to enable the use of a broad range of analysis algorithms with the same monitor. Hence, the monitor is designed to be *independent* of any analysis algorithm that evaluates behavior profiles. However, different analysis algorithms use different parameters to evaluate task behavior. Some of them use system calls and system call arguments, while others use the return address stack or the program counter [58].

The monitor defines different *monitor modes* to make it reconfigurable at runtime in order to modify which parameters are monitored in accordance to the analyzing algorithm and control monitor memory usage. The monitor is designed to selectively monitor specific parameters like the system call id, system call arguments,

return address stack, task resources etc. These parameters can be reconfigured at run-time by changing the monitor's mode. By using this filtering mechanism, the monitor minimizes the overhead associated with monitoring data since it only aggregates the data that is required to build up the behavior knowledge base for the associated anomaly detection algorithm. To set up the monitor modes, the monitor offers an interface through a *monitor API*.

The monitor extracts the system call information parameters along with further OS state information whenever a system call happens. As the monitor intercepts the system call execution in order to record the data, it introduces additional run-time overhead into the system call handling.

Furthermore, real-time systems have limited memory and, hence, real-time applications have severe restrictions on the amount of memory they can use. The monitor has similar restrictions. System call information is collected at thread basis. The amount of memory used is also governed by the frequency of system calls invocations and the amount of monitored parameters. This mechanism enables the controller to re-configure the monitor at run-time in order to prevent memory overflows and safeguard overall memory usage.

Analyzer

The ORCOS Analyzer provides a framework for anomaly detection algorithms that, based on the run-time reconfigurability of the system, may be exchanged at run time. Algorithms implemented to analyze system behavior must comply with the API provided by the monitor and be applicable for analyzing self-x behavior.

The workflow of each analyzing algorithm is:

1. generate a behavior representation according to the requirements of the algorithm from the behavioral data base provided by the monitor,
2. evaluate and match the actual behavior against the normal behavior profile provided knowledge base,
3. inform the controller in case of deviations and detected anomalies.

Strictly separated from the monitor which in turn is directly attached to the system call handler, the analyzer is scheduled individually. The monitor must record the data whenever a system call is invoked while the analyzer is triggered when a sufficient amount of behavioral data is available (determined by the anomaly detection strategy). An anomaly detection method that can be integrated into the analyzer is introduced in [69, D.o.S.O.M.S. Sect. 3.2] .

5.5.2 Self-optimized Flexible Resource Management

The Flexible Resource Manager (FRM) [155] permits an over-allocation of resources under hard real-time constraints. The technique allows to minimize the internal waste of resources by putting temporarily unused resources, which are only reserved for the worst-case, at other applications' disposal. Additionally, an adaptive self-optimizing system or middleware software can be built using this technique.

To use the Flexible Resource Manager, applications have to stick to a specific resource allocation paradigm and can specify multiple modes of operation – so called profiles – to allocate additional resources if other applications temporarily do not need them. The resource allocation paradigm comprises:

1. The application has to specify *a priori* the minimum and maximum limits per *resource usage*. The application can not acquire less or more resources than specified in the current active profile, which the FRM activates. If the application wants to do so, then it has to specify a new profile with appropriate limits. The activation of the new profile underlies an *acceptance test* of the FRM.
2. The FRM is in charge of the assignment of applications into their profiles. If a reconfiguration between profiles is enforced by the FRM, application-specific transition functions are activated. This allows for an application-specific change between different operation modes with different resource requirements.
3. The FRM also registers the actual resource consumption of the active profile of an application, which must be within the specified limits. The FRM guarantees to the applications that they can allocate the resources up to the specified limit in the active profiles. In case of a resource conflict – when the system is over-allocated – the FRM solves the conflict by forcing applications into other profiles so that every resource request can be fulfilled. The FRM ensures that no deadlines of hard real-time tasks are violated. This is done by only allowing an over-allocation of a resource if a plan for solving every possible conflict exists and this plan can be scheduled under hard real-time constrains. Figure 5.71 illustrates this approach.
4. Resources are distinguished which can be reassigned within an negligible reallocation time and resources which have to be configured in the background by the system software. Resources which are reconfigurable in background need more time to be reassigned between different applications. All resource demands of background reconfigurable resources – also within the specified limits of the actual profile – require an announcement to the operating system. Between the announcement and the assignment a delay is assumed. The profile specifies a *maximum* delay per background reconfigurable resource. Note that this delay is a worst-case value.

The ability to schedule and the deadlock-freeness of the FRM approach have been formally proven.

To enable engineers to easily use the FRM and the profile model, the approach is integrated into the high-level design process for self-optimizing mechatronic systems. A semi-automatic code generation was presented [33], which allows for a generation of profiles out of hybrid real-time state charts. Hybrid state charts combine continuous models and discrete real-time state charts (e.g. for the reconfiguration model). The application programmer only has to specify a minimum of additional information to generate profiles. For simulation purposes the FRM was not only implemented on top of the operating system DREAMS [51] , but also integrated into MATLAB/SIMULINK. This enabled a simulation of an application using the FRM in which the continuous part, including the controller and the plant, of the application is encapsulated [157].

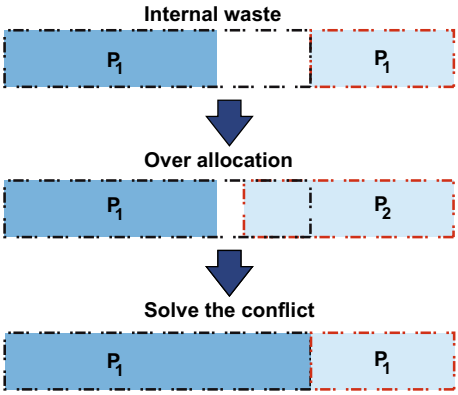

Fig. 5.71 Over allocation

5.5.3 Self-optimization in the Operating System

If self-optimizing applications change their behavior and their resource requests dynamically during run-time, even the underlying real-time operating system (RTOS) should reconfigure its QoS by means of the currently provided services. For example, a specific protocol stack should only be present in the RTOS, when applications request this protocol for their communication. I.e., a reconfigurable/customizable RTOS includes only those services that are currently required by its applications. Hence, services of the RTOS must be loaded or removed on demand. Thus, the RTOS also releases valuable resources that can be used by applications.

As self-optimizing applications are – in the context of this book – embedded mechatronic systems, they run under hard or soft real-time constraints. Thus, the reconfiguration of RTOS components is critical. The RTOS always has to assure a timely and functional correct behavior and has to support the required services. Hence, the reconfiguration underlies the same deadlines as the normal operation of the applications. To handle exactly this problem, the FRM can be applied to the RTOS as well. The FRM model executes the reconfiguration under real-time constraints. The acceptance test inside the FRM assures that the reconfiguration does not violate real-time constraints.

The main idea is to release resources of system services by deactivating/activating basic versions or activating development alternatives (e.g. an implementation on the FPGA instead on the CPU) of these services. These different states of the services are modeled as different profiles for each service. Then, these RTOS components will be handled by the FRM as normal application profiles. Thus, no change of the FRM model is required.

5.5.3.1 Reconfiguration Model

To build an online configurable RTOS, components which can be re-configured (activated, deactivated, etc.) have to be identified during run-time. For this purpose the offline configurator TEReCS is reused (**T**ools for **E**mbedded **Re**al-Time

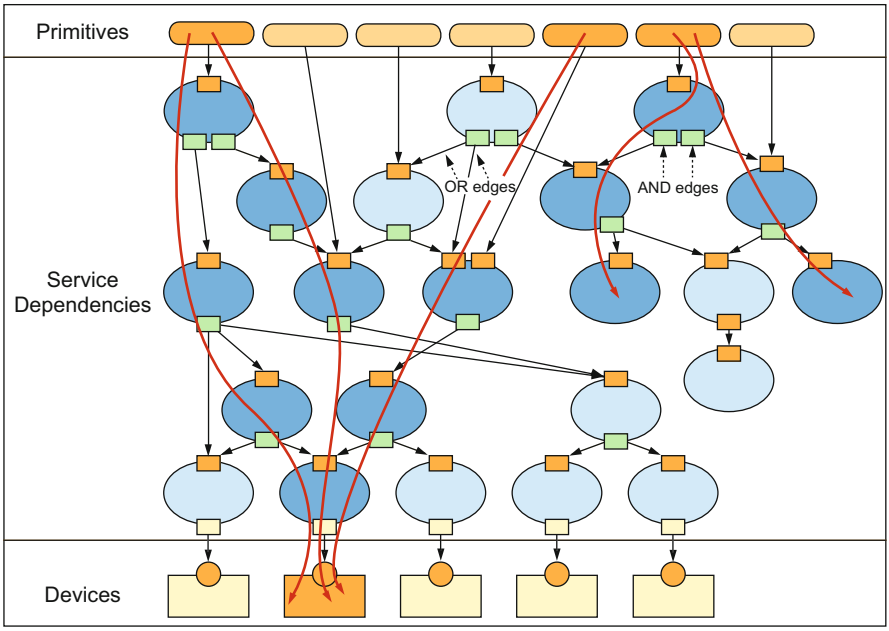

Fig. 5.72 TEReCS's design space description from system primitives via services down to hardware devices (from [29]).

Communication **S**ystems) [28, 29]. In the TEReCS approach the complete and valid design space of the customizable operating system is specified in a knowledge base by a so-called AND/OR service dependency graph [36]. This domain knowledge contains options, costs, and constraints and defines an over-specification by containing alternative options.

The complete valid design space of the configurable operating system is specified by an AND/OR graph:

- Nodes represent *services* of the operating system and are the smallest atomic items, which are subject of the configuration,
- Mandatory dependencies between services are specified by the AND edges,
- Optional or alternative dependencies between services are specified by the OR edges,
- Services and their dependencies have costs and can be prioritized,
- *Constraints* (preferences, prohibitions, enforcements under specific conditions) for the alternatives can be specified,
- Root nodes of the graph are interpreted as *system primitives*/*system calls* of the operating system.

The algorithm works, e. g. for communication primitives, as follows: A path can be found through the complete graph from the sending primitive down to the sending device, considering the routing and then up to the receiving primitive. The services

that are visited on this path have to be installed on the appropriate nodes of the service platform (see more color saturated nodes in Fig. 5.72). Thereby, the path should create minimal costs by the use of the services.

Such paths will be searched for all primitives that are used in the requirement specification. Because only a subset of all primitives is normally used, especially the particular selection is responsible for the instantiated services and its parameterization. The primitives can be considered as the strings of a puppet. Depending on which strings are pulled, the "configuration" of the puppet will change accordingly. The service dependencies can be compared to the joints of the puppet. Therefore, the algorithm is named *"Puppet Configuration"*.

The online configuration makes use of pre-defined solutions that have been configured offline. Thus, it is up to the online configuration phase to identify the use cases, for which the solutions have been created and to activate them. The identification is simple, because it depends on the system primitives, which are used by applications and other clusters. Those pre-defined solutions have to be instantiated so that all required primitives are implemented for the concrete situation during run-time. If primitives are unused an alternative cluster can be activated during run-time, which does not implement the unused primitives.

The same system primitives that have been used to create a pre-defined solution are leading to the selection of that solution component in the coarse-grained design space level. This condition must be assured during the specification of the abstract design space for the pre-defined solutions. This problem must also be solved by the system expert offline. This procedure is allowed, as TEReCS' main philosophy obliges the encapsulation of all expert knowledge in the design space descriptions.

Example

Figures 5.73 and 5.74 sketch an example for two pre-defined cluster options (B+C). The primitives *Scheduling* and *Communication* in Fig. 5.73 are used by the equally named clusters from Fig. 5.74. The option B is generated from A if the primitive *Scheduling* is not used. The option C is generated alternatively. In Fig. 5.74, the pre-generated solutions B+C are included as the *OS Hierarchy Option I* and *II*.

Except the cluster *Scheduling* all other clusters can use both options alternately. Only the cluster *Scheduling* requires explicitly the solution of the *OS Hierarchy II*, which supports multiple threads. If the primitive *CreateThread* will be used, then the cluster *Scheduling* is requested. Thus, the request of the primitive *CreateThread* from an application requests the cluster *Scheduling* to be instantiated. Moreover, as the cluster *Scheduling* requires the internal primitive *Scheduling*, the cluster *OS Hierarchy II* also has to be instantiated instead of the cluster *OS Hierarchy I*.

The alternatives of the operating system services are modeled as different profiles. Using the previous example, *OS Hierarchy I* and *OS Hierarchy II* are mapped into two profiles of the system service *OS Hierarchy*, which is from the point of view of the FRM handled as a normal application.

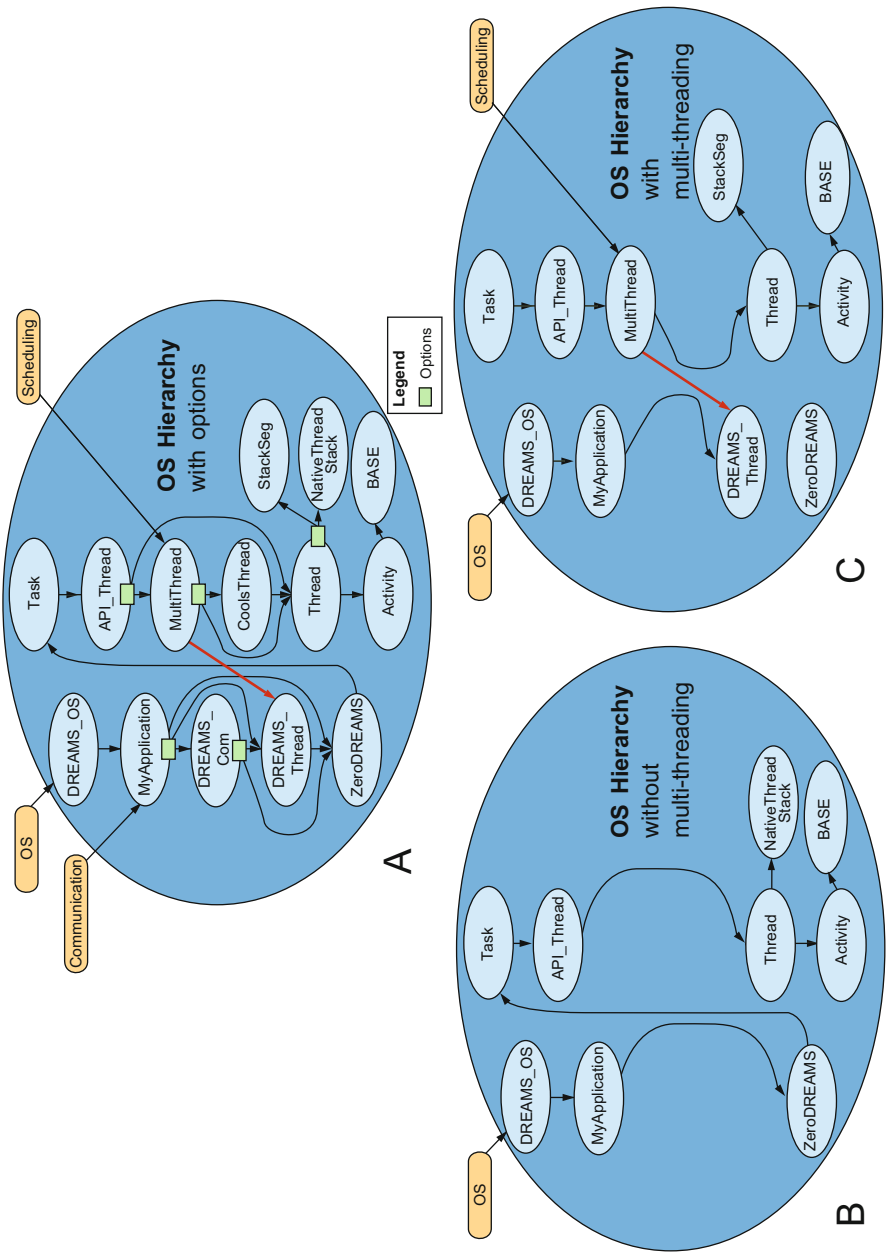

Fig. 5.73 Zoom to the fine-grained level of to the OS cluster with its optional components (A) and two pre-defined configuration examples (B+C).

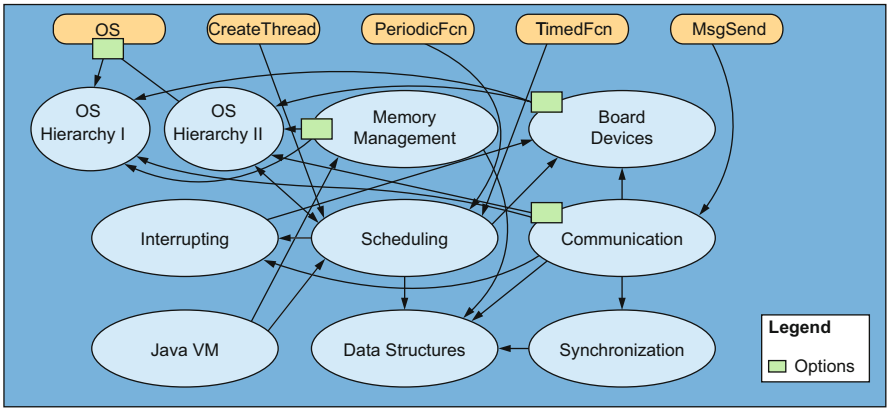

Fig. 5.74 OS design space at 2^{nd} level with integrated options for pre-defined solutions of clusters.

5.5.3.2 Online Reconfiguration

For each primitive a new resource is introduced. When an application or other RTOS service wants to use a system primitive, it requests the corresponding resource. Initially, each service holds each corresponding resource of the primitive it provides. When an application or other system service arrives or wants to use a primitive, it has to request the corresponding resource, which must be in the range of the specified resource boundary of its actual profile. As a reaction, the FRM activates a corresponding profile of the service, where the service does not block the primitive by occupying the corresponding resource but implements the primitive by activating an alternative pre-defined solution. The service implements in the enter and leave functions of the profiles the switch between the pre-defined clusters. These reconfiguration functions represent the Online-TEReCS module as a whole entity. In a profile the meaning for the system services of holding a primitive provides the reverse meaning of an application: a service holding a primitive means that the primitive is not required and does not need to be implemented. On the other hand, when an application holds a primitive, the service has to provide the primitive's code.

Clusters representing the system services are reconfigured during run-time employing the FRM approach to mediate the reconfiguration. The alternatives are modeled as different profiles. Using the previous example again, *OS Hierarchy I* and *OS Hierarchy II* are mapped into two profiles of the system service *OS Hierarchy*, which is from the point of view of the FRM handled as a normal application.

As sketched in Fig. 5.75 and before, the reconfiguration of the RTOS cluster components is completely managed by the Online-TEReCS module in the enter/leave functions of the corresponding profiles. The reconfiguration options are modeled as optional profiles being offered by Online-TEReCS, which are activated and deactivated by the FRM. Each profile defines which primitives are used by a system service profile and which are not used – as the primitives are modeled as resources.

Thus, the FRM does not need to distinguish between RTOS components and normal applications. The FRM mediates the system primitives (resources) between all the applications and the RTOS. Thus, it handles the competition between the applications and the reconfiguration options of the RTOS. The system primitives represent the dependencies between the services. The services which are locked do not provide system primitives in corresponding profiles by allocating the system primitive itself. Thus, the FRM manages and assures that all dependencies are considered during run-time, otherwise the concrete allocation of a system primitive corresponding recourse would by surmount the maximum available number and lead to unfeasible system configuration. Real-time constraints are respected by modeling the reconfiguration time of the RTOS in the switching conditions respective to the minimum dwell time of the profiles and the acceptance test in the FRM.

Applications must define all real-time constraints regarding their future resource allocations. Additionally, an application can only allocate resources in the range of the profile, which is currently active. With this information the FRM guarantees by means of the acceptance test, that all resource allocations can be timely performed. Deactivating a system service, by activating a profile in which this service is not configured into the system, and an application, which is currently not using the service but specifying a possible future use through the defined profile parameters, creates an over-allocation state. If the application wants to use the service this leads into a reconfiguration. The acceptance test assures that a system service is only deactivated if the reconfiguration to reactivate the service can be executed "in time" to provide the resources when needed.

The creation of pre-defined solutions for clusters is done automatically. For each combination of possible requests or dismissals of *system primitives* and *internal primitives* a configuration is generated. For the optimization and reduction of the design space of the operating system, a system expert might restrict the combinations of parallelly instantiated system primitives to only those ones that make sense which cover other solutions and – with high probability – are not used simultaneously.

A repository stores all pre-defined solutions of the clusters. A cache will temporarily store the code and the description of optional configurations for clusters, in order to speed up the loading of required cluster implementations. The cache can retrieve other configuration's implementations from background storage (hard disk) or from the network (cf. Fig. 5.75).

The FRM tries to optimize the system according to the current resource requirements of the components (system services and applications) and the quality information of the profiles. To do this, the FRM requests the application and system services to change their current profiles. This results in a reconfiguration of the RTOS and an optimization of the resource usage between the applications and the operating system.

The FRM approach includes the definition of quality values per profile. Thus, the FRM can not only reason about the optimality of application profiles, but it can additionally reason about the optimality of the RTOS configuration.

By the integration of TEReCS and the FRM into a RTOS a self-optimizing real-time operating system (SO-RTOS) is derived. Such an OS adapts itself with the help of the FRM to the needs of the current applications executed on top of it. Using this technique services can be deactivated and freed resource can be put at the applications' disposal. The real-time capability of the FRM ensures that only such services are deactivated, which can be reactivated under hard-real time constraints, if required by application tasks.

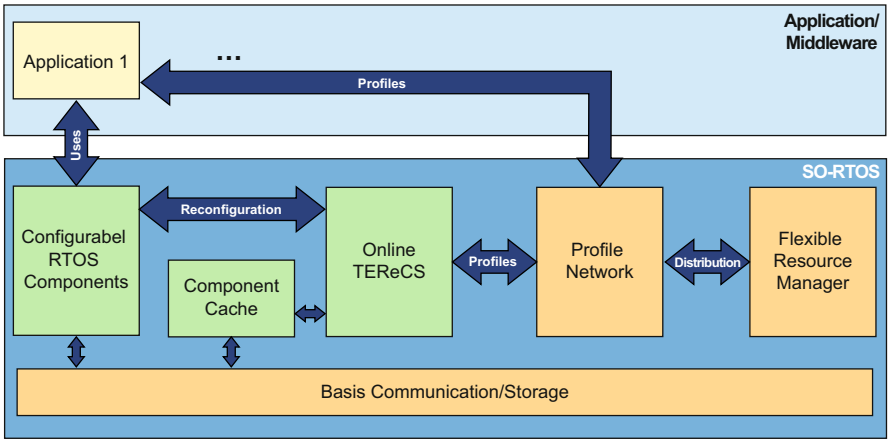

Fig. 5.75 Integration of TEReCS and the FRM framework into the RTOS

5.5.4 Hierarchical Flexible Resource Manager

The Flexible Resource Manager concept was adapted to system virtualization [195]. Integrated systems with multiple software systems executed on a single hardware unit provide often a more resource-efficient implementation compared to multiple separated hardware systems. System virtualization realizes this integration of multiple systems with maintained separation, and therefore is well suited for safety-critical mechatronic systems. The hypervisor allows the sharing of the underlying hardware among multiple operating systems, each executed in an isolated virtual machine. We developed a real-time capable hypervisor for embedded systems, which is characterized by multicore support and possibility to host both paravirtualized and fully virtualized guests [14, 77].

The Hierarchical Flexible Resource Manager consists of FRM components on two levels, *Guest-FRMs* and *Hypervisor-FRM*, as depicted in Fig. 5.76. In a partitioned manner, the FRMs on both levels take resource management decisions. The Guest-FRM is part of the operating system and switches between task profiles in order to assign resources to tasks, as previously described in detail. The Hypervisor-FRM is part of the hypervisor and switches between virtual machine profiles in order to assign resources to virtual machines. Communication takes place in both directions. The Guest-FRMs inform the Hypervisor-FRM about the dynamic resource

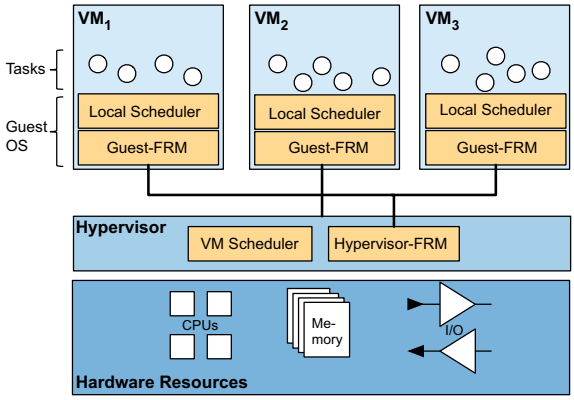

Fig. 5.76 General Architecture: Hierarchical Flexible Resource Manager [195]

requirements and current resource utilization. The Hypervisor-FRM's resource allocation among the virtual machines is based on this information. The Hypervisor-FRM informs the Guest-FRMs about the assigned resources in order to allow the Guest-FRM to manage its resource share. The cooperation of the hypervisor's virtual machine scheduler and Guest-FRM guarantees that each guest system becomes (1) active in time, (2) for a sufficient duration and (3) equipped with the necessary resources, in order to allow the guest to execute its applications in compliance with their timing requirements.

The implementation of the Hierarchical Flexible Resource Manager requires paravirtualization, since the Guest-FRMs as part of the guest operating systems have to pass information to the Hypervisor-FRM as part of the hypervisor. According to paravirtualization [16], the guest operating systems are aware of being executed in a virtualized manner on top of a hypervisor and not on top of the bare hardware. The guest operating systems are modified and explicitly ported to the interface of the hypervisor. By consequence, they are able to communicate with the hypervisor. The requirement to modify the guest operating system is outweighed by the advantages gained in terms of flexibility of an explicit communication and cooperation of hypervisor and operating systems.

In contrast to static virtualization techniques where the resource shares are assigned a priori to the virtual machines, our approach allows for a dynamic resource allocation even across virtual machine borders. The cooperation of Hypervisor-FRM and Guest-FRMs is based on a hierarchical mode change protocol. We refer to profiles and transitions between them. A non-empty set of *task profiles* is assigned to each task, as introduced before. The Guest-FRM is in charge of switching between these profiles at run-time. The task profile P_{r_j} of task τ_j is defined as:

- resource allocation minimums and maximums:
  $\forall$ *resources* R_k *with limit* $\hat{R}_k : 0 \leq \phi_{j,k}^{min} \leq \phi_{j,k}^{max} \leq \hat{R}_k$
- profile quality $Q(\tau_j) \in [0,1]$
- subset of the set of task profiles to which the Guest-FRM can switch from P_{r_j}

In addition to task profiles, there are *VM profiles*, which specify the minimal and maximal resource limits for a virtual machine. VM profiles unite the active profiles of the tasks of a virtual machine. In case of a task profile transition, the VM profile is updated and communicated to the Hypervisor-FRM and used for the resource assignment among the virtual machines.

A VM profile P_{VM_i} is defined as follows:

- resource allocation minimums and maximums:
  $\forall\, resources\ R_k : \forall\, tasks\ j\ of\ VM_i :\ \Phi_{i,k}^{min} = \sum_j \phi_{j,k}^{min},\quad \Phi_{i,k}^{max} = \sum_j \phi_{j,k}^{max}$
- profile quality $Q(VM_i) \in \mathbb{N} : Q(VM_i) = \sum_j Q(\tau_j)$
- subset of the set of VM profiles to which the Hypervisor-FRM can switch from P_{VM_i}

The set of active profiles is called *configuration*. The possibility to switch between profiles on both the task level and on virtual machine level enables a dynamic resource assignment across virtual machine borders. A Guest-FRM can shift resources by task profile switches from one task to another; and similarly, due to the cooperation of the FRMs on the two levels, resources can be reallocated from task τ_i of VM_1 to task τ_k of VM_2. The Hypervisor-FRM activates a VM profile with a lower resource allocation maximum for VM_1 and according to this, the Guest-FRM of VM_1 activates a task profile with a lower resource allocation for τ_i. This allows the Hypervisor-FRM to activate a VM profile with a higher resource allocation maximum for VM_2 and the Guest-level FRM of VM_2 to activate a task profile with a higher resource allocation for τ_k.

The hierarchical FRM assigns fractions of resources at run-time to other tasks whenever a task does not use the complete amount of resources as needed in the worst case. If at a later point in time, the resource lending task needs more resources than remaining, a *resource conflict* occurs and has to be solved under real-time constraints. There are two kinds of resource conflicts, caused by two kinds of dynamic resource reallocation. The Guest-FRMs can reallocate resources among their tasks and the Hypervisor-FRM can reallocate resources among virtual machines. In both cases, an acceptance test precedes and a resource reallocation is accepted if and only if:

- $\forall\, Resources\ R_k\ ,\ \forall\, tasks\ 1..n : \sum_{i=1}^{n} \phi_{i,k}^{max} \leq \hat{R}_k$
- the FRM identifies a feasible *reconfiguration*

A reconfiguration is a sequence of profile switches that activate a configuration, which fulfills the worst-case requirements of all tasks. If such a reconfiguration plan includes VM profile switches, it is called *global reconfiguration*. The Hypervisor-FRM and at least two Guest-FRMs have to perform configuration switches. In contrast, a *local reconfiguration* only includes task profile switches and is accomplished by a single Guest-FRM. A reconfiguration plan can only be accepted, if the schedulability analysis attested that the time required to execute the reconfiguration does not lead to a deadline miss. The reconfiguration plans for conflict resolution are stored in *conflict resolution tables*. An entry is created after a reconfiguration was accepted and lists the required profile switches to reach a state that guarantees all

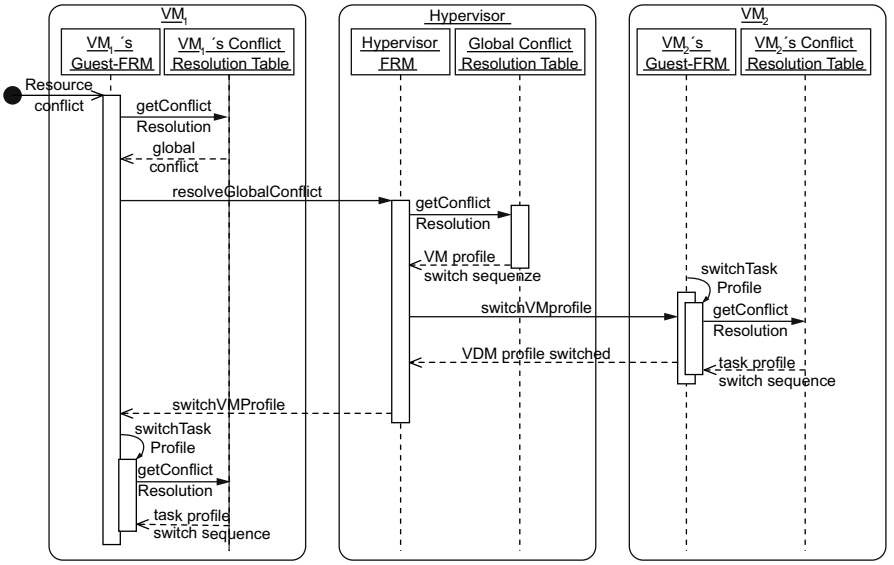

Fig. 5.77 Conflict Resolution: Global Reconfiguration Sequence [195]

deadlines. If a conflict is always solved by reconfiguration to the initial state, there is at most one entry per task profile. A larger table with the possibility to reconfigure to multiple optimization levels is more promising, but requires additional memory.

In the following, an example depicts the conflict resolution process. It is assumed that task *A*, executed in virtual machine VM_1, has a specific worst-case requirement of a resource and consequently, this resource share was assigned. Since the actual resource usage of task *A* was significantly below the reserved amount, the Guest-FRM switched to another profile and made a fraction of the assigned resources available to task *B* of the same VM. In case of a resource conflict, i.e. task *A* requires a larger resource share than remaining, the Guest-FRM resolves the conflict by switching to task profiles with a resource distribution that fulfills the timing requirements of task *A*. The sequence of profile switches that have to be performed to obtain this state was stored in VM_1's local conflict resolution table when the acceptance test for the resource reallocation was passed.

It is possible that the Guest-FRM can not resolve the resource conflict, since a global reconfiguration is required to achieve this. This is the case, if it was caused by a resource reallocation to another guest system. A share of the resource reserved for virtual machine VM_1 could have been assigned to virtual machine VM_2 by the Hypervisor-FRM, and further assigned by VM_2's Guest-FRM to task *C*. The Hypervisor-FRM informed VM_1's Guest-FRM about this resource reallocation and the Guest-FRM noted this in the local conflict resolution table. The conflict resolution is depicted in the UML sequence diagram of Fig. 5.77. In case

of a resource conflict of task A, the Guest-FRM of VM_1 informs the Hypervisor-FRM, which prompts the Guest-FRM of VM_2 to release the supplemental resources. The Guest-FRM of VM_2 switches the profile of task C to one of lower resource utilization. The Hypervisor-FRM can accordingly switch the profile of both VM_1 and VM_2 and inform the Guest-FRM of VM_1 to ultimately activate the conflict resolving profile switch for task A.

In order to guarantee real-time requirements, hypervisors for embedded systems typically assign virtual machines statically to processors. This static approach is inappropriate for the varying resource requirements of self-optimizing mechatronic systems. Virtualization's architectural abstraction and the encapsulation of virtual machines support migration, i.e. the relocation of a virtual machine from one processing element and connected memory to another one at run-time. Prerequisite is a multiprocessor architecture with an instance of the hypervisor running on each processor. Multiple processing elements operate on their own dedicated memory, but are connected by input/output devices.

By the application of emulation, virtual machine migration is even possible for heterogeneous multiprocessor platforms, which are characterized by processors with differing instruction set architectures. Emulation executes program binaries that were compiled for a different architecture. This translation between instruction set architectures realizes cross-platform software portability. We developed a real-time capable emulation approach [115] and a real-time migration for heterogeneous multiprocessor architectures, which analyzes at run-time whether a virtual machine with real-time constraints can be performed without risking a deadline miss [82]. It selects an appropriate target for the migration and controls the migration process. This migration manager was integrated into our hypervisor Proteus [83].

For the migration of virtual machines, a coarse-grained dynamic reassignment of resources can be realized in comparison to the mode switches of the Hierarchical Flexible Resource Manager. In particular it is useful for open systems, in which the addition of applications or subsystems at run-time is possible. System virtualization isolates arriving potentially faulty or malicious software from existing critical applications. The acceptance of an application or even an entire subsystem typically changes the load balancing significantly and it might actually be necessary to perform migration to be able to accept an arriving subsystem.

System virtualization and its ability to reuse subsystems is a powerful technique to meet the functionality and reliability requirements (see [69, D.o.S.O.M.S. Sect. 3.2.8]) of increasingly complex systems and has potential to support the migration to multiprocessor platforms. Targeting this architecture, the Hierarchical Flexible Resource Manager provides a resource management for the dynamically varying resource requirements of integrated adaptive systems. The two-level solution beyond virtual machine borders has the potential to increase the resource utilization significantly compared to static approaches.

5.6 Virtual Prototyping

Jörg Stöcklein, Wolfgang Müller, Tao Xie, and Rafael Radkowski

Virtual prototyping is a technique, which applies Virtual Reality-based product development for the engineering of mechanical and mechatronic systems. Virtual prototyping is based on the modeling, design, and analysis of Virtual Prototypes (VPs), i. e. computer-internal models, which are executed and analyzed in a Virtual Environment (VE). VPs are typically developed prior to physical prototypes (or mockups), which are mainly profitable for relatively small subsystems, e. g. the Hybrid Energy Storage System (HES) (cf. Sect. 2.1.5) or Active Guidance Module (cf. Sect. 2.1.3). Compared to physical prototypes, the development of VPs is less expensive and time-consuming, and VPs provide a significantly higher flexibility for change requests and variant management. Moreover, due to the virtualization of the prototype and the environment, Virtual Prototypes faciliate the early evaluation of the final product. All experiments can be conducted under controlled conditions of a well structured Virtual Test Bench (VTB) and for instance can easily be repeated for regression testing.

Today, with the outcome of sufficiently fast and affordable computing platforms and devices, **Virtual Reality** (VR) based Virtual Prototyping is widely accepted in several engineering disciplines. Examples are the design review of vehicles [114, 192] and plant engineering [212]. Meanwhile, **Augmented Reality** (AR) based technologies also are frequently considered for engineering related applications such as evaluation of automotive prototypes and the preparation of experiments [117, 209].

We apply Virtual Prototyping for the development of complex self-optimizing mechatronic systems with inherent intelligence, which react autonomously and are flexible to changing environmental conditions. This applies at each level of the hierarchical structure that makes up a complex mechatronic system: e.g. Mechatronic Function Modules (MFM) such as an intelligent suspension strut, Autonomous Mechatronic Systems (AMS) such as a vehicle, and Networked Mechatronic Systems (NMS) such as a vehicle convoy. Due to their complex structure and highly dynamically evolving behavior, self-optimizing mechanical systems impose huge challenges during their entire product development process, starting from the initial specification to composition, analysis, testing, and final operation [68].

As such, for the development of self-optimizing mechatronic systems, the adequate combination of VTBs, VEs, and simulation-based VPs can be beneficial over classical development platforms as they are highly flexible and customizable for the individual, dynamic needs and requirements of such systems and support different views and seamless integration of multiple integrated models. Nevertheless, the manual configuration of virtual platforms for mechatronic systems is cumbersome and error-prone as it is conducted on an individual base and requires the integration of different domains like electrical and mechanical engineering.

We introduce a novel Virtual Prototyping platform dedicated to the development of self-optimizing mechatronic systems. The platform seamlessly combines

VR- and AR-based user interaction and control with model-based execution of the different integrated VPs, which are controlled by domain-specific simulators or integrated Hardware-in-the-Loop components [17, 172]. For true design automation, we investigated automatic linking of the different models to VEs and advanced VTB technologies for the controlled and repeatable execution of experiments.

The remainder of this section is organized as follows. In the next subsection we will introduce the principles of VPs and VEs 5.6.1. Next we will describe our approach to automatic VE configuration. Section 5.6.2 presents the agent-based automatic model linking. Section 5.6.3 gives an agent-based solution to link the models to different visualizations. Finally, Sect. 5.6.4 outlines the basic concepts of our self-optimizing VTB, which is based on the principles of mutation analysis. All subsections present application examples based on the reconfigurable miniature robot BeBot [99], which serves as one of our main development platforms.

5.6.1 Virtual Prototypes and Virtual Environments

Rafael Radkowski

A **Virtual Prototype** (VP) is defined as a computer-internal representation of a real prototype of a product [128]. Figure 5.78 outlines the basic concept of a VP, which is based on the notion of a digital mock-up (DMU) with the definition of the product shape and structure. A DMU is typically based on two models: 3D CAD models and the logical product structure. A VP extends a DMU by further domain-specific aspects like the kinematics, dynamics, force or information processing. Each of these aspects is defined by a different domain-specific view. As such, VPs help engineers to exercise, analyze, and evaluate the interaction of the system and its subcomponents. That way, VPs facilitate an easy comprehension of the product behavior long before a first physical mock-up is built.

VPs are executed and analyzed in a **Virtual Environment** (VE). A VE is a VR/AR-based synthetic environment, which provides a visual, haptic, auditive, and interactive experimentation environment for the VP [86].

In our approach, we developed a methodology and technologies for simulation-based VEs for the advanced interactive analysis of self-optimizing mechatronic systems. Our VE also comprises a Virtual Test Bench (VTB) for the structured and controlled execution of experiments and tests, respectively. As such, we have advanced the idea of the classical VE [86] towards a simulation-based VE like in the Extensible Modeling and Simulation Framework (XMSF) [32] or in the High Level Architecture (HLA) [39]. That means, our approach is not limited to visualize VP models, rather than also integrates behavioral simulation models and physical hardware (Hardware-in-the-Loop) for real-time user interaction with VPs for realistic product development, analysis, and testing. For this, we have already introduced a common VE infrastructure for semi-automatic multi-domain integration of the VP [17, 172].

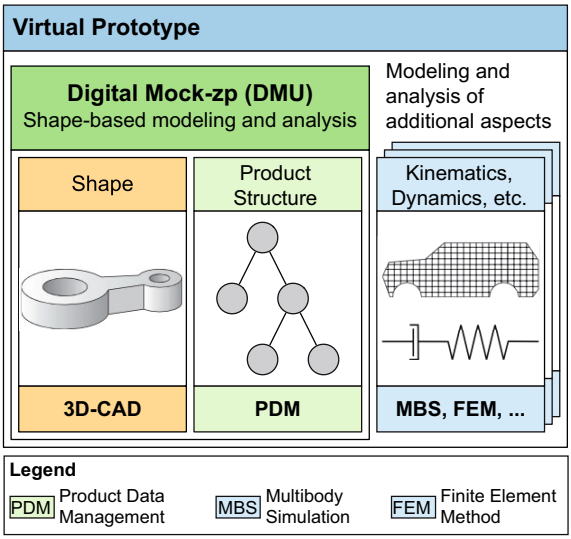

Fig. 5.78 Schematic representation of the term Virtual Prototype (VP)

Figure 5.79 presents an overview of the principle components of our VE and their interaction by the example of our BeBot robot [99]. The VE on top is composed of interacting Virtual Prototypes (VPs) and a **Virtual Test Bench** (VTB). The latter covers the objects of the test environment including the behavior for interacting with the VPs, i.e. stimuli, as well as a test strategy given by a verification plan for the controlled execution of experiments and tests, respectively. We can see that each VP has different aspects: shape, structure, and behavior such as kinematics, dynamics behavior, which can be given by an executable component, such as Hardware-in-the-Loop, or a domain-specific simulation model, such as a MATLAB/Simulink model, as illustrated at the bottom of Fig. 5.79.

Typically, a VE is created manually like the maritime combat simulation in [85] or the virtual factory in [186]. That means, either interactions between the different VPs and the VE are implemented manually or by means of predefined data structure with a fixed set of variables for each VP. As of today, with the increasing intelligence of mechatronic systems, the number of considered system components and their interaction significantly increases. The increasing complexity of data structures and their interaction both make the manual integration of VPs to a simulation-based real-time VE infrastructure highly time-consuming and error-prone. Therefore, we have developed an agent-based approach where software agents [108] identify the physical and non-physical interaction between single VPs automatically and link them to a VE. An agent compares two function structures given as standard models of the product development process and identifies similarities for automatic model linking.

Before we will introduce our approach for VTB automation, the next two sections outline our concepts for automatic model linking and their linking to the visual representation.

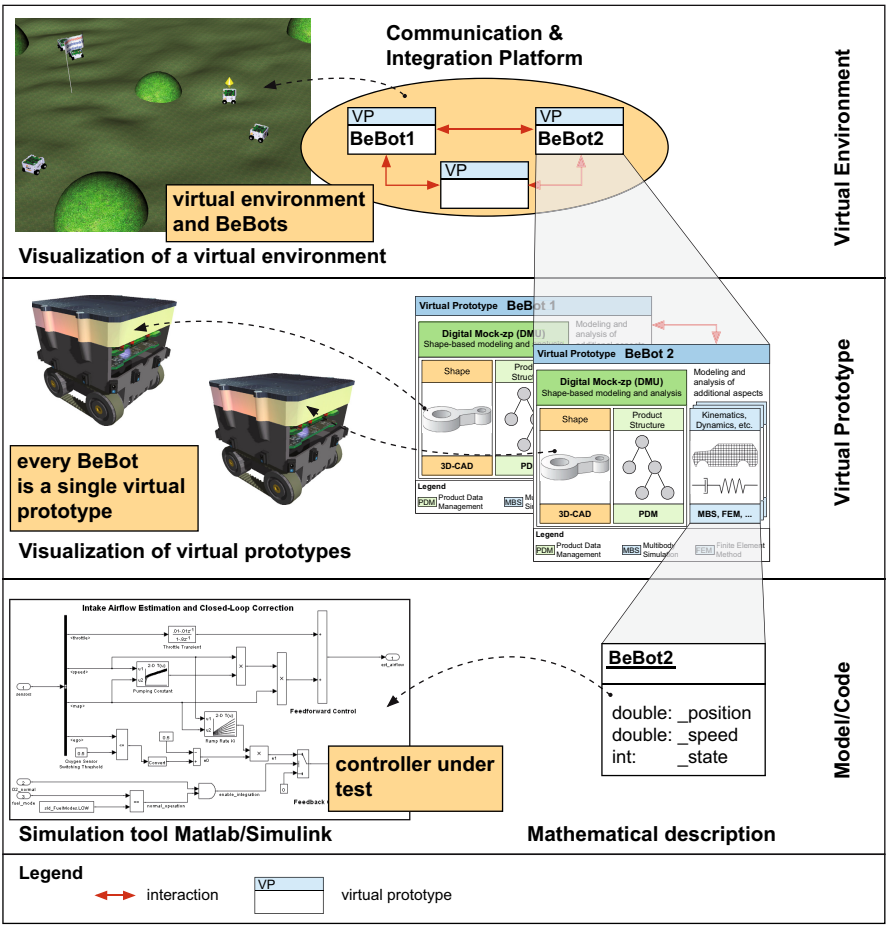

Fig. 5.79 Schematic overview of the composition of a virtual environment

5.6.2 Automatic Model Linking

Rafael Radkowski

Our automatic model linking is based on Semantic Web technologies. The Semantic Web (SW) facilitates machines to capture the content of web pages and other similar documents [22]. Thus, machines can automatically link information from different sources. The **Resource Description Framework** (RDF) plays a decisive role for the SW. RDF is a description language, which is used to annotate the content of a web page; it is the syntax for meta-data of a web page. The underlying model is based on a directed graph. The nodes of the graph denote resources, while the edges denote properties. The idea of RDF is to describe complex facts by a network of simple RDF statements. A RDF statement consists of a subject, a predicate, and an

object. The predicate is the most important part of the semantic. It is defined as a W3C (World Wide Web Consortium)-standardized predicate for the description of business cards. The SW can only function successfully if all participants have the same understanding of these predicates and interprets them in the same manner.

A reasoning system is necessary to identify relations between two RDF-annotated web pages. RDF represents the database only. For that purpose, query languages are used to query the necessary information. Queries need to be transformed to a form where reasoning is possible by processing production rules.

Some researchers have already used RDF and the related reasoning mechanism for the engineering of technical systems. For instance, Bludau and Welp (2012) [27] have developed a framework, which supports engineers during the development of mechatronic systems. Their framework searches for active principles and solution elements, which meet a given specification. Restrepo (2007) [177] uses SW techniques to search for design solutions for a given problem. He has developed a database, which contains different design solutions; Simulink RDF annotates every solution. A reasoning mechanism searches for solutions for the given design problem. Ding et al. (2009) use XML-based annotations to annotate CAD models with design constrains, goals, relationships, and bounds [50]. They mainly annotate geometric, topological, and kinematic properties of a given design. Their approach can be utilized to find an optimal design solution during the product development process. The authors use XML as a notation basis, but their notation is similar to a RDF notation. Ding et al. (2009) developed an XML-based product representation that also allows an annotation of geometrical properties [49]. For further information, Li et al. (2009) present a classification of different annotation approaches [135]. They all demonstrate the importance of software agents and annotation techniques in engineering design, on which our approach is based.

The main principle of our approach for agent-based automatic model linking is outlined in Fig. 5.80. On the bottom left of the figure, we can see the example of two Virtual Prototypes (VP), which are linked to a Virtual Environment (VE). A software agent represents each VP.

In this example, each VP includes two models: a 3D model and a behavioral model. Both models are shown at the top of the figure with the 3D model on the left and the behavioral model on the right. The latter is illustrated by a MATLAB/Simulink screenshot. The application contains and processes a model that simulates the behavior. Both models are annotated. Therefore, an RDF-notation is utilized; the annotations describe the purpose of the models. Normally, more than two aspect models (3D model and behavior) and one VP are used.

The main task of the software agent is to combine both aspect models (3D model and behavior) to one VP and to integrate them into the VP-template, which is provided by the VE. As shown in Fig. 5.80, this requires five steps.

The first step is an initialization by a user (1). Normally, the user specifies one model (3D model or behavior model) as the origin. The objective of the agent is to identify the other models and to integrate them into the template of the VP. Therefore, the agent searches for every available model. A service directory of the agent platform references them. The annotation of every available model is read (2). The

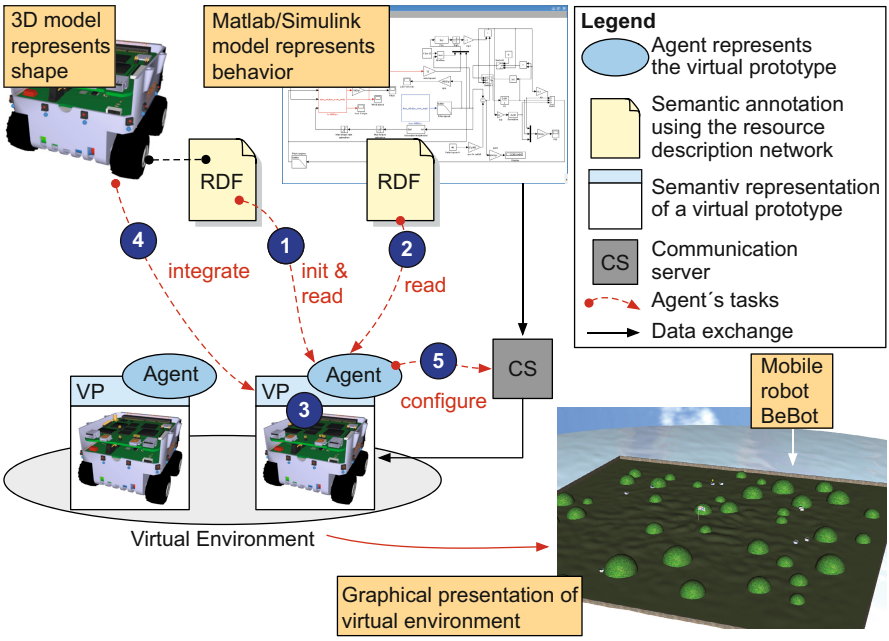

Fig. 5.80 Automatic model linking overview

agent compares the RDF model of the 3D model with the RDF model of the behavior model (3). A set of production rules is used for this task. If two models pass the production rules, the agent assumes them to be similar.

Next, the 3D model is integrated into the VE by loading the model and including it into an internal data model (4). However, the behavioral model cannot be included by simply importing it. Since the processing of the behavioral model is very resource consuming, it is executed on a separate computer system. As only simulation results are required for an analysis of the VP, only the results are transmitted by means of a communication server (CS) to the main host. The CS manages the communication between the simulation software and the VP/VE. The agent configures the CS and establishes the communication between the behavior model and the 3D model (5).

5.6.2.1 Semantic Annotations with the Resource Description Framework

After automatic linking, the VP models are enhanced by semantic annotations by means of the RDF (Resource Description Framework) language [22]. RDF provides a syntax for web page meta data, where the underlying model is based on a directed graph. The nodes of the graph refer to resources, the edges to properties. The idea of RDF is to describe complex facts by a network of simple RDF statements. We apply RDF as an annotation language to describe the context of each VP model. The challenge of the annotation is to identify the relevant elements of a specific model, which are required to conduct the automatic integration of the model. At this

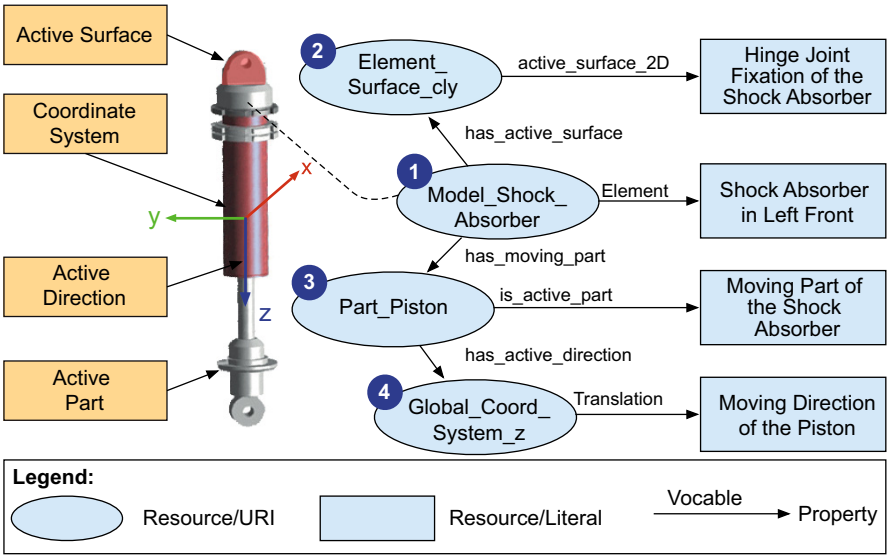

Fig. 5.81 Example of the semantic annotation of a 3D model

step, we presume the availability of VP models of the following aspects: shape (3D model), behavior, functions, and activations.

For semantic annotation, we will outline the main concepts by the example of a shock absorber as given in Fig. 5.81. Four items of a 3D model are annotated: the entire part, the active surfaces, the subparts, and the active directions:

Entire part *(1)*: The resource is linked to the variable, which represents the model, normally a file. In this example, the name of the model is *Model_Shock_ Absorber*. The variable is annotated by the predicate *element*. To describe the *element*, a literal is used. In this example it is 'Shock Absorber in Left Front '.

Active surface *(2)*: The active surfaces of a component are the surfaces, which fulfill the functions of this component [158]. The resource is linked to the variable in the data structure of the 3D model, which represents the active surface. In the example, the name of the variable is *Element_Surface_cly*. *active_surface_2D* is the predicate, which defines the item as an active surface.

Active part *(3)*: This type of part moves to cause an effect of the entire model. The resource refers the variable, which describes the main part in the data structure of a 3D model; in this case it is the entire piston. The word *has_moving_part* is used as RDF predicate to annotate to the subpart, which describes the part in the structure of the entire 3D model; the variable's name is *Part_Piston*. Furthermore, to describe this active part, the predicate *is_active_part* is used. It facilitates the annotation with a literal. In this case the literal is: 'Moving Part of the Shock Absorber'.

Active direction *(4)*: The fourth annotation type is the active direction. According to Pahl and Beitz (2007) [158], the active direction describes the direction,

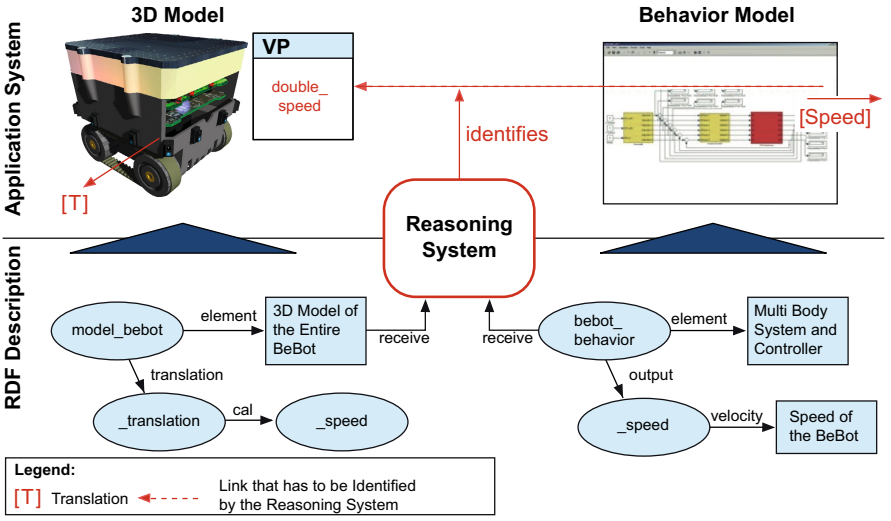

Fig. 5.82 Schematic overview of the reasoning using an example

into which a function of a component effects. In Fig. 5.81, the piston of the shock absorber is the active part, which active direction should be described. The variable *Global_Coord_System_z* describes the coordinate system; it describes the direction of moving. It is attached to the resource *part_piston* by the predicate *has_active_direction*. In addition, the resource *Global_Coord_System_z* needs to be annotated with a human understandable literal. In the example, the literal is 'Moving direction of the Piston'. It is attached to the resource by the predicate translation.

However, though we outlined our concepts by just one example it should be sufficient to give an impression to show how the semantic annotation with RDF is applied, and which elements of a 3D model are necessary in order to describe the purpose and functionality of a 3D model in a natural way (literals). In total, we have defined 36 RDF keywords to describe active surfaces and directions as well as the parts and subparts of an assembly. Further details can be found in [171].

5.6.2.2 Software Agent Reasoning

Recall here that the software agent has two major tasks. First, it has to identify similar aspect models and, second, it has to establish the communication and the exchange of data between different software tools. The following paragraphs will outline the reasoning mechanism for establishing the communication infrastructure as sketched by the example in Fig. 5.82.

Figure 5.82 shows a 3D model of our miniature robot BeBot on the left side and a behavioral (MATLAB/Simulink) model of the robot on the right. We presume that both models are already annotated by RDF, where the example just shows a small

portion of it, the variable speed. As a BeBot can fulfill different tasks in a team, the variables of the behavior model need to be linked with the related variables of the 3D model and with other VPs. For that, a reasoning mechanism identifies variables that are related to each other. In general the software agent compares the RDF models, two at the same time, and converts the results of the comparison into a numerical value. This numerical value expresses the similarity of two models, respectively their variables. The comparison is based on production rules. Each production rule has the form:

$$IF\,(Condition\,C_1\,\&\,Condition\,B_1\,\&\,\ldots\,\&\,Condition\,C_n\,\&\,Condition\,B_m)$$
$$THEN\,A_1;\ldots;A_0$$

Conditions of type *C* are predicates of the 3D model, conditions of type *B* are predicates of the behavior model. By applying these production rules a set of corresponding predicates is identified. As corresponding predicates each pair of predicates is defined, which describes the same meaning of an item. For instance, condition *C* states *translation & cal* (calculated) and condition *B* states *output & velocity* are defined as corresponding predicates; they result in an output $A = 1$. Otherwise they result in an output $A = 0$. The result of this calculation is weighted by a weight value *g*:

$$A_0 = Ag + E$$

The value g indicates the importance of a production rule. The term *E* is an offset. It is calculated by comparing the literals of each pair of corresponding predicates. This is done by a statistical phrase analysis (see also [171]). A vector describes the results of every production rule:

$$R_{similar} = A_1, A_2, \ldots, A_0$$

That vector is a rating scale for the quality of the similarity of a certain task. After the vector is determined, the agent ranges all results $R_{similar,i}$, where the index *i* refers to a certain production rule of two compared models. A statistical method is used for this comparison, the so-called squared ranking. This method calculates a likelihood value $p(i)$ for each corresponding pair of predicates:

$$p(j) = \frac{1}{size} \cdot \left(E_{max} - (E_{max} - E_{min}) \cdot \frac{(R_{similar,j} - 1)^2}{size - 1} \right)$$

with two rating values E_{max} and E_{min}. These values express the estimated amount of minimal and maximal corresponding predicates, respectively the number of possible relations. The equation assigns a numerical value to each production rule and expresses the fulfilled rules by a numerical value. A high value indicates the similarity of the compared variables. The agent links all data, which value $p(j)$ exceed a threshold:

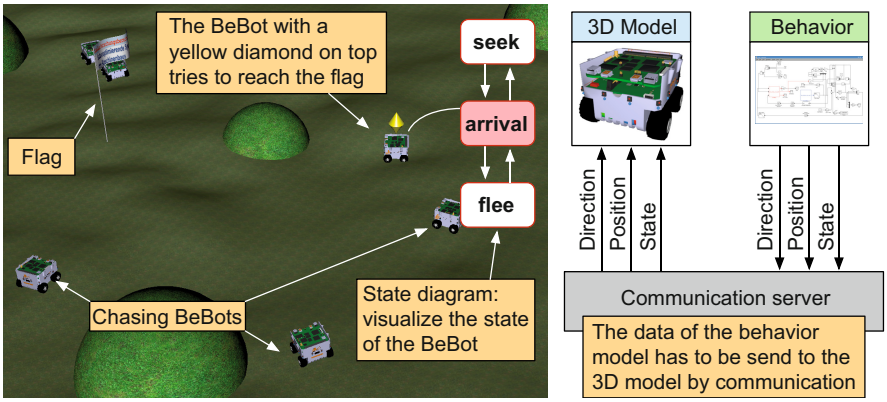

Fig. 5.83 Overview of the application (left), the architecture of the application (right)

$$p(i) > p_{threshold}$$

The value p(i) needs to cross a threshold $p_{threshold}$. At this time the threshold is determine empirically. After this decision, the agent establishes the communication between the behavior model and the 3D model. Further information about the communication infrastructure and the behavior of the agent inside the VE has been presented in [173].

5.6.2.3 Application Example

To proof our concepts of automatic model linking, a software prototype was developed and validated by the BeBot robot application example [99]. Figure 5.83 shows a screenshot with an overview of the VE on the left. In the environment, the Flag is located in the middle of the environment and spheres are placed as obstacles for the BeBots around it. The BeBot with the diamond on top tries to capture the flag. On the right, it shows the corresponding infrastructure of the VPs.

Each BeBot is represented by a 3D model and a behavioral model. Both models are annotated by RDF [171]. The annotation of the 3D model describes the input variables to set the position and direction of a robot as well as a state diagram to visualize its current state. The behavioral model provides the position and direction of each BeBot. Both applications (VE and behavior) need to be linked by the agent, the agent has to identify the variables and link them respectively.

In summary, the agent is able to realize the communication between both models/applications utilizing the RDF-based annotations of both models. The desired application can be realized, without any need for a user to describe the communication manually.

5.6.3 Visualization Agents

Rafael Radkowski

In engineering, software agents are utilized in many different application fields. Agents are mainly used to support the design process by making decisions, which are based on a large amount of data. Mendez et al. (2005) describe an agent-based software architecture for agents in virtual environments [142]. They introduce the concept of expert agents. Expert agents are software agents with an expert knowledge in a specific technical domain. Based on this knowledge, the expert agent is capable of finding a solution to solve a specific problem. The paper introduces a similar idea. However, their desired tasks are training tasks. Galea et al. (2009) present a framework for an intelligent design tool that assists a designer, while working on micro-scale components [67]. They do not label their framework as software agent, but they use a similar artificial intelligence technique to model the knowledge and the reasoning system. Multi-agent systems have also been used to support engineers in time-critical tasks [161]. An agent aggregates relevant information from other agents that represent different members of an engineering team. Thus, an engineer gets the right information at the right time. Baolu et al. (2009) propose the so-called Multi-Agent Cooperative Model (MACM) [15]. It is a product design system that facilitates easy access to similar data of different products. The system facilitates the product design and manages product data. With its aid the product design cycle will be shortened. Geiger et al. (1998) introduce the agent modeling language SAM (Solid Agents in Motion), a language to describe 3D models in virtual environments and their behavior [71]. In contrast to our work, SAM covers the complete visualization of animated processing of SAM-specific rules rather than links to arbitrary behavioral models.

In the following, we will describe the concept of visualization agents for linking visual representations to VP models. For this, we presume two different agents: one agent for the VP (VP-agent) and a second one for the visualization (Vis-agent). We also presume an agent platform, which is formed by a set of interacting agents, which finally form the VE. Each agent contains an internal data model. This data model describes the represented object like meta-data. Along the lines of the previously introduced linking of models, we apply RDF in combination with a reasoning mechanism to identify similarities of annotations between visualization and model agents. As a result, if models are identified as similar, we assume that the visualization is suitable to explain the data of a VP. In the following, an overview of the entire concept is presented. Then the necessary agent models are described and finally, the reasoning mechanism is introduced.

5.6.3.1 Concept

Figure 5.84 shows a schematic overview of the basic principles. On the left side, a box represents the VP of a mobile BeBot robot. The box on the right side indicates

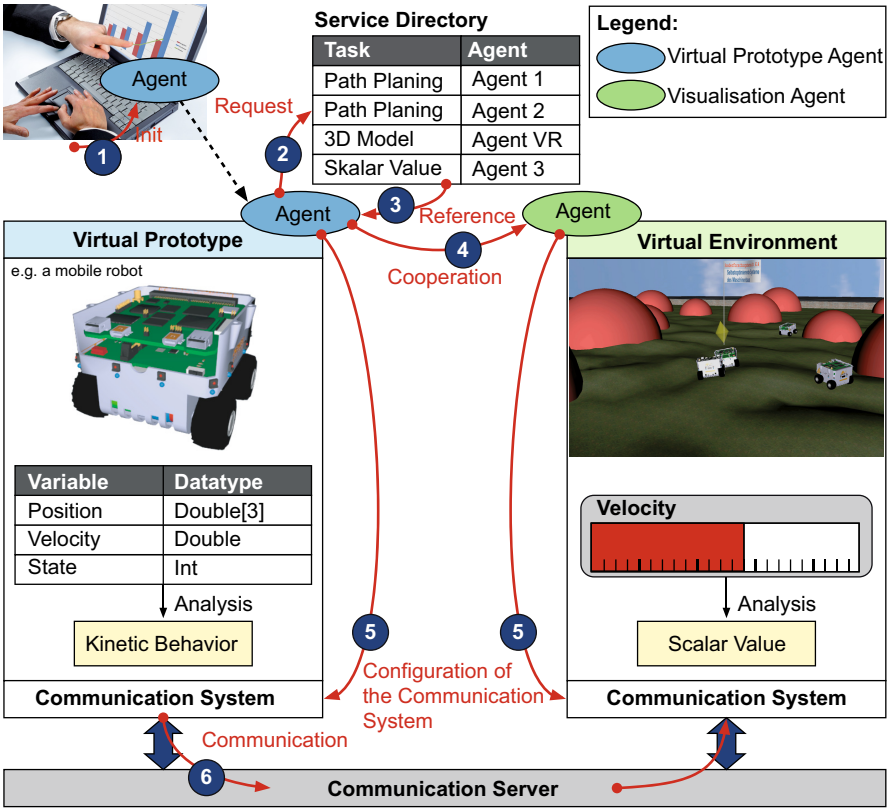

Fig. 5.84 Concept of the visualization agents

a VE with visualizations. Software agents are associated to the VP (VP-agent) and the visualization (Vis-agent).

The objective of both agents is to identify an appropriate visualization by communication and cooperation. This visualization should help a user accomplish a certain task. In the following, detailed steps are explained, which are necessary to identify a suitable Vis-agent for the visualization of a specific VP-agent along the six steps of Fig. 5.84.

In step *(1)*, a user needs to initialize the VP, the simulation, and the VE. At the beginning, the agent platform is initialized and the agents start to operate simultaneously. The user needs to specify the task, which he or she wants to carry out, e. g. to analyze the kinetic movement or to inspect the parts of the VP.

In step *(2)*, the VP starts to search for an appropriate visualization for the VP and its data. For this, the VP-agent contacts a service directory provided by the VE and queries for reachable Vis-agents. It contains a list of all reachable agents, sorted by a category of tasks. The VP-agents submit a desired category, which meets the kind of visualization the VP-agent searches for. Normally, more than one visualization

facilitates the visualization of the data of the VP. Thus, in step *(3)*, the VP-agent receives a list of potential Vis-agent candidates.

In step *(4)*, the VP-agent contacts each Vis-agent with the reference from the service directory. Thereby, it submits data about the functions of the VP and data about the task the user desired to apply to every Vis-agent. Each Vis-agent compares this data with two internal data models. These data models characterize the capabilities of a Vis-agent. A similarity-vector Evis is calculated. This vector and its numerical values represent the capability of the agent to visualize the queried task and data. This vector is returned as a result to the VP-agent. At the end of this step, the VP-agent has a set of similarity-vectors, one for each Vis-agent.

Next, the VP-agent compares the different similarity-vectors and by this, it compares the different Vis-agents. Therefore, a reasoning mechanism is used. After the VP-agent has decided for one Vis-agent *(4)*, they start to cooperate.

In step *(5)*, the visualization is realized. Therefore, the data of the VP needs to be submitted to the Vis-agent and its represented visualization. Figure 5.84 shows a simple example: the VP has a 'velocity' that needs to be visualized. The Vis-agent on the right side can visualize this by a bar chart. For that, the 'velocity' values need to be transmitted to the Vis-agent. To realize this data exchange, a communication server is used [173]. This communication server manages the data exchange between different connected programs. In the example, this is a program that simulates the VP and its behavior and a VE that hosts the visualization *(6)*. The task of both agents is to configure this communication server and by this, configure the data exchange. The VP-server informs the communication server about the attributes it wants to allocate. The Vis-agent informs the server, what data it requires. If the requested data is available, the data exchange starts until an agent stops its operation.

5.6.3.2 Data Models

Agents maintain three different RDF-based data models to represent their knowledge: a task model and a function model for the VP-agent, and a visualization model for the Vis-agent.

Task Model

A task is defined as *'the application of methods, techniques, and tools to add value to a set of inputs – such as material and information – to produce a work product that meets fitness for use standards established by formal or informal agreement'* [204]. A common technique to specify a task is a block diagram where each block represents a certain activity and the entire diagram represents the task (cf. Fig. 5.85). A string inside the block denotes the activity, e. g. 'Check the impulse response'. Incoming arrows represent input data (objects or information), which are processed during the activity. Information can be the velocity of a mobile robot, for instance, or an object of the computer-internal representation of the shape of the VP. In addition, an activity may also refer to a method and a tool. To concretize the task, boundary conditions can be specified. For instance, this can be the required amount of data.

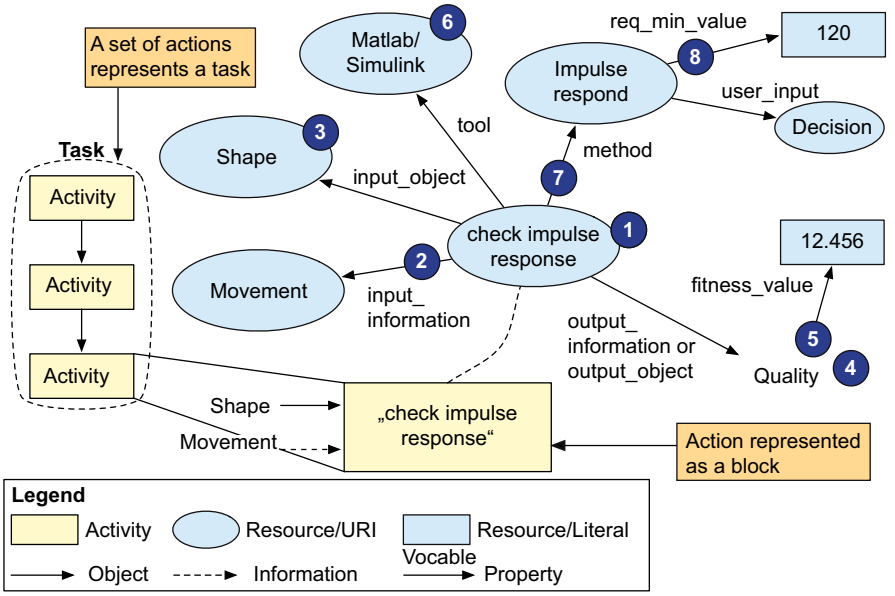

Fig. 5.85 RDF-Description of a task model

To describe this task model as computer-internal representation, an RDF-based notation has been developed, which covers the definition of a set of resources and properties describing a task. Figure 5.85 shows an extract of the resulting RDF-based notation for one action. The following resources and properties are used in that figure:

- **Activity** *(1)*: The activity itself is the main element. It is specified by a resource, which keeps a string of the action itself.
- **Input information** *(2)*: The activity has a property *input_information* to specify the incoming information. The property refers to a resource, movement in the shown example. This resource keeps a link to the computer-internal data of this information.
- **Input object** *(3)*: The activity uses a property *input_object* to refer to the incoming objects. The property points to an additional resource, which contains a link to the computer-internal representation of this object.

The properties *output_information* and *output_object (4)* are used to refer to the outgoing information and objects. The properties refer to resources, too. Every activity can use multiple input and output objects and information.

- **Fitness value** *(5)*: Every input and output object and information uses a property *fitness_value* to express a numerical value or a set of numerical values that quantifies the objects and information. It is an optional property. It refers to a literal that contains the numerical value.

- **Tool** *(6)*: The vocab tool labels a property of the activity to describe an additional software tool. This software tool is utilized to carry out the named activity. The property refers to a resource containing a link to this certain tool. This property is optional.
- **Method** *(7)*: Every activity needs one method, which is utilized to process the activity. The vocab method is used to express this property. It refers to an additional resource. At this time, only the resource keeps a name of the method. The methods are provided in a database. The user can only select a method.
- **Conditions** *(8)*: Every method can be concretized by additional conditions. Two conditions are used. The first one is a requirement value. It denotes a minimum amount of data that is necessary to process this method. This property is expressed by the keyword *req_min_value*. It refers to a literal containing a numerical value. The value quantifies the requirement. The second condition expresses whether a user input is necessary during this activity or not. For this, the keyword *user_input* is used. It labels a property, which refers to a resource. This resource contains a statement that expresses the type of user input. For instance a Boolean decision (yes/no). The conditions are optional properties.

Function Model

Figure 5.86 shows a schematic overview of the function model. It is defined to specify the functionality of a VP in respective to the product under development. Therefore, a function structure according to Pahl/Beitz (2007) is used [158]. For the graphical presentation of the function structure a block diagram is a common technique. Each block represents a function. A function is defined as 'Operation, activity, process, or action performed by a system element to achieve a specific objective within a prescribed set of performance limits'. According to Pahl/Beitz (2007) it is expressed by a substantive and a verb [158]. The substantive names the object that is processed by the function. The verb names the process or the activity the technical system carries out. To build up a function structure, the functions of a technical system are connected by the flow of material, energy, and information. The arrows in Fig. 5.86 show these flows. The entire function structure represents a model of the functionality of the technical system.

To use the function structure as a knowledge model for an agent, a formal computer internal representation has been developed. Therefore, we have developed a RDF notation, too. Figure 5.86 shows an extract of the developed RDF scheme in order to introduce the resources and properties and to demonstrate its application. The example explains how a function model can be built up and which properties are necessary to describe the functionality of a VP by RDF. The following notation is used:

- **Function** *(1)*: The function itself is expressed by a resource. The resource keeps a character string of the function. It is the main resource of every function and it is required.
- **Function term** *(2)*: To facilitate an automatic processing of the function term, the function uses a property *function_term*. This refers to an empty resource that

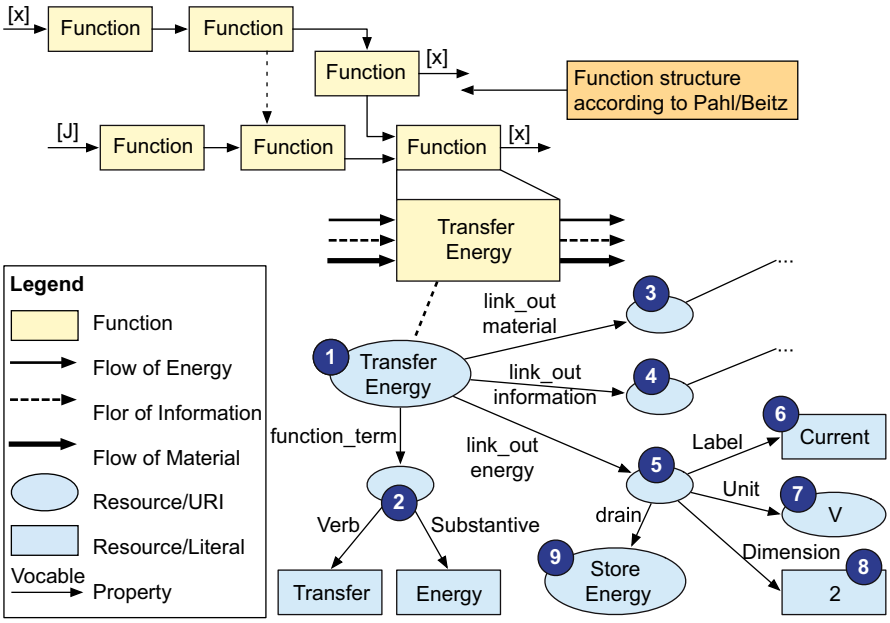

Fig. 5.86 Schematic presentation of the RDF notation of a function structure

points to the substantive and the verb of the function. The property substantive refers to a literal of the substantive. The property verb refers to the function verb literal.

- **Flow of energy, information, and material**: To model these three types of flow, the function uses the properties *link_x_material (3)*, *link_x_information (4)*, and *link_x_energy (5)*, where x is a wild-card for in or out. The property refers to an empty resource.
- **Attributes**: The flow of energy, information, and material need to be specified by three additional properties. These properties are the label, the unit of the technical dimension, and the dimension of the value. The property label *(6)* refers to a literal, it contains a character string that names the flow. The property unit *(7)* points to a resource. This resource keeps a value of a technical dimension; in the example the unit 'V' for voltage is shown. The last property depicts the dimension of the flow. A scalar, a vector, or an array can model the flow. For this, the property dimension *(8)* is used. It refers to a literal to characterize the dimension.
- **Source and drain** *(9)*: Every flow has a source and a drain. To specify them, the properties source and drain are applied. Both properties refer to a resource that contains a link to the related function.

Visualization Model

A visualization is defined as a technique to create images, diagrams, 3D models, and animations to communicate and to explain abstract data. For instance, it can be a bar

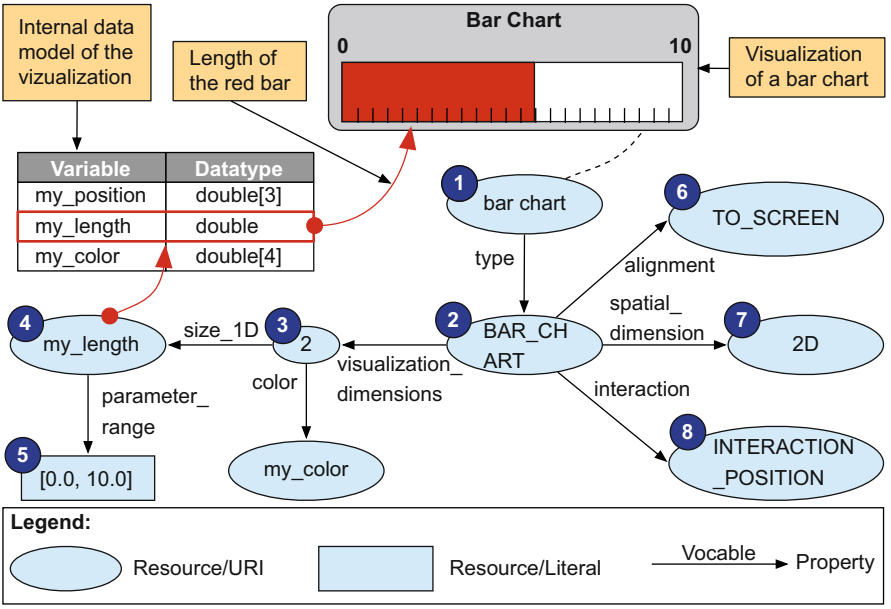

Fig. 5.87 Schematic representation of the visualization model

chart as a visual representation of a scalar value (cf. Fig. 5.87). In the context of the visualization agent, visualizations are diagrams and 3D models. Both of them are a part of the VE. A visualization is annotated by an RDF-notation, too. Figure 5.87 shows an overview of the used resources and properties and how they are applied. As an example, a bar chart is used. To define the RDF-notation, it was necessary to identify elements and attributes that specify a visualization and its capabilities. The following resources and properties are used:

- **Visualization** *(1)*: The visualization itself is modeled as a resource. The entry of this resource refers to the internal data model of the visualization. This key element is required.
- **Visualization type** *(2)*: To specify the type of visualization the related resource has a property *type*. This property refers to a resource that denotes the visualization by a keyword. In the example, the keyword BAR_CHART specifies a bar chart. Other keywords are SYMBOL, ICONS, NET, TREE, and some more. Each of them represents a certain type of visualization.
- **Dimensions** *(3)*: Every visualization has a set of visual variables. These visual variables are modified to express abstract data by a graphical representation. The property *visualization_dimensions* specifies the number of visual variables each visualization provides. It refers to a resource that contains the number of modifiable visual variables.
- **Visual variable** *(4)*: This property is used to specify the visual variables itself. To describe them, visual variables according to Bertin (1983) [24] are used. These

variables define the size of a visualization, the position, the orientation, the grey scale value, the color, the texture, and the shape. They are transferred to properties like *size_1D*, *size_2D*, *size_3D*, *position*, *orientation*, *color*, etc. For instance *size_1D* specify a visualization, which size can be modified in one dimension. In the example of the bar chart, it is the length of the bar. The property refers to a resource. This resource keeps a link to a variable of the visual variable, which represents its length inside the computer-internal data model. In the example shown, it refers to the double *my_length*.

- **Parameters** *(5)*: To concretize the visualization, the visual variable can be limited by a set of parameters. At this time, two parameters respectively properties are used: *range* and *threshold*. The property *range* specifies the boundaries of a dimension. For instance, the bar of the bar chart is limited by a minimum and a maximum value. In the example, it ranges from 0 to 10. The property *threshold* names a threshold, which is shown by the visual variable.
- **Alignment** *(6)*: The property *alignment* specifies the spatial alignment of the visualization. It refers to a resource that contains a keyword. Used alignments are HUD (head-up display), TO_SCREEN, TO_MODEL, and some more. For instance, TO_SCREEN means that the visualization is rotated into the viewing direction of the user automatically. Thus, the user sees the right face of the visualization every time.
- **Spatial Dimension** *(7)*: A visualization can be distinguished by its spatial dimension. This feature is specified by the term *spatial_dimension*. The property refers to an additional resource, it contains the dimension: 0D (Points), 1D (Lines), 2D (Surfaces), 3D (Volumes).
- **Interaction** *(8)*: The property *interaction* needs to be specified if input data from the user is necessary or possible, e.g. when a visualization should be moved on screen or the range of a bar needs to be adapted interactively. The property refers to a resource that contains the keyword INTERACTION_x, where x is a wild card for RANGE, POSITION, and some more. For instance, INTERACTION_RANGE means that the user can modify the boundaries of a visual attribute.

This data is sufficient to specify a visualization with a set of annotations. Its computer-internal representation has been integrated into an agent model to specify the visualization.

Reasoning Mechanism

The reasoning mechanism identifies the Vis-agent, which associated visualization is adequate to visualize the data of the VP or the VP itself. In general, the reasoning mechanism compares the models and converts the results to a numerical value. This numerical value expresses the capability of a Vis-agent to visualize the data of a VP.

We apply three steps to identify a proper visualization. The first step is processed by the Vis-agent. The second and the third step are processed by the VP-agent. At the beginning, we presume that the VP-agent has submitted its models to the Vis-agent.

In the first step, production rules are used to determine the similarity between different models. A Vis-agent keeps a set of production rules to evaluate the request. Each production has the form

$$IF\,(Condition\,C_1\,\&\,Condition\,B_1\,\&\,\ldots\,\&\,Condition\,C_n\,\&\,Condition\,B_m)$$
$$THEN\,A_1;\ldots;A_0$$

Conditions of type C_n are related to the function model and the task model of the VP-agent. Conditions of type B_m are related to the visualization model and task model of the Vis-agent. Each visualization agent contains a set of production rules. These rules compare the referred models and determine, whether the Vis-agent fulfills the requirements of the VP-agent. If the capabilities meet the requirements, action A_0 is processed. Each action is an equation of the form

$$A_0 = Ag + E$$

with the term $a = 1$ if the production rule is passed and $a = 0$ if the production rule fails. The value g is a weight that indicates how important the production rule is. The term E is an offset; it represents the experience of the agent and describes, how useful this action was during previous uses. The value A_0 represents the result. The results of every production rule are combined in one vector:

$$E_{Vis} = A_1, A_2, \ldots, A_0$$

This vector is a rating scale for the quality of the visualization in a certain task. Every Vis-agent calculates this vector and returns it to the VP-agent.

In the second step the VP-agent compares all results $E_{Vis,i}$, where the index i refers to a certain Vis-agent. A statistical method is used for this comparison, the so-called linear ranking. This method calculates a likelihood value $p(i)$ for each visualization:

$$p(i) = \sum_{j=0}^{m} E_{max} - (E_{max} - E_{min}) \cdot \frac{(E_{Vis,j} - 1)^2}{size - 1}$$

with rating values E_{max} and E_{min}, which determine the estimated amount of minimal and maximal fulfilled production rules. During the development of a VP-agent, it needs to be estimated how many production rules need to be fulfilled in order to identify a suitable visualization. This estimation needs to be evaluated by the developer of a individual visualization. The equation assigns a numerical value to each production rule and expresses the fulfilled rules by a numerical value. A high value indicates that the Vis-agent is adequate to visualize the VP and the generated data of the VP.

In the third step, the VP-agent decides, which visualization agent is applied: the VP-agent takes the Vis-agent with the highest value $p(i)$. One constraint is the equation:

$$p(i) > p_{threshold}$$

The value $p(i)$ needs to cross a threshold $p_{threshold}$. At this time, the threshold is determined empirically.

The concepts of visualization agent have been implemented and proven by the following application example.

5.6.3.3 Application Example

To test the concept of visualization agents and the developed models, a software prototype has been developed and a BeBot robot [99] application example has been implemented.

The software prototype has four components. The first component, a VE, is based on OpenSceneGraph[15], an open source scene graph library for the development of 3D graphic applications. The second component is a simulation for mobile robots based on Open Steer [178], an open source software library, which covers a robot model and a set of functions like seek, evade, path following, and leader following. The third component is JADE (Java Agent DEvelopment Framework). JADE is a software framework that facilitates the implementation of multi-agent systems by means of a middleware that complies with the FIPA (Foundation for Intelligent Physical Agents) specifications, a standard specification for software agents. Furthermore, it provides a set of tools that supports the debugging and deployment phases of agents. The described agent behavior has been implemented using the JADE framework. The fourth component is a communication server. It realizes the exchange of data between the three components, mentioned before. The entire system works in real time. The technical details of the server are described in [173].

In addition to the four components, the software SchemaAgent from Altova[16] is used to annotate the models. It provides a graphical user interfaces to model the resources, properties, and the entire RDF graph. The RDF model is stored in an XML notation. Finally, the software library Jena is used to implement the RDF vocabulary for the annotation, the RDF queries, and the reasoning system.[17] Jena is a framework for building Semantic Web applications. It provides a programmatic environment for RDF and RDF-Schemas including a rule-based inference engine. The inference engine has been extended to realize the method, which is described in Sect. 5.6.3.2.

Based on that platform, we will outline the basic principles of visualization by means of the Capture the Flag (CtF) application example.

[15] `www.openscengraph.org`
[16] `http://www.altova.com`
[17] `http://jena.sourceforge.net/`

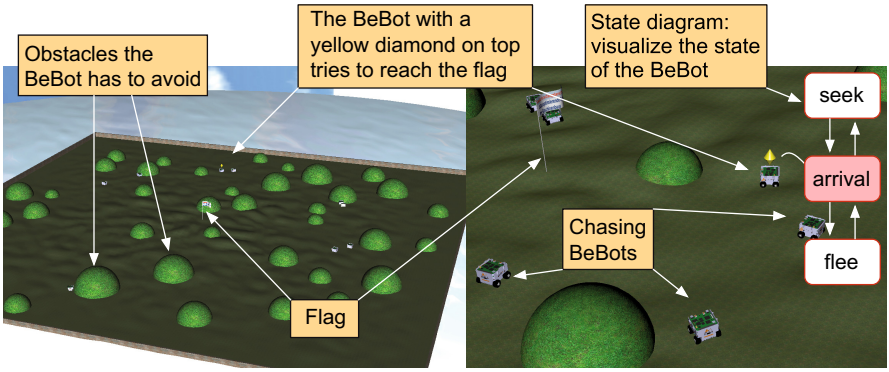

Fig. 5.88 Overview of the virtual environment (left), detail view of the test (right)

CtF is an example, which is originally based on a game where a hunter has to capture a flag, the other players chasing the hunter and try to prevent him from capturing the flag. In our case the players are the BeBots with one hunter and n chasers. The BeBots operate autonomously without any interactions from a user. Figure 5.88 shows two screenshots from the application. The left part shows an overview of the VE.

The flag stands in the middle of the environment with spheres as additional obstacles for the BeBots. The right part shows a detailed view to the scene. The BeBot with the diamond on top identifies the hunter. A state machine with six states models the behavior of a BeBot. Each state represents a type of behavior: seek, flee, obstacle avoidance, robot avoidance, pursuit, and arrival. The BeBots decide autonomously which state is active; the decision is based on a rule system.

To test the visualization agents, the BeBots and one visualization (state diagram) have been implemented and represented by software agents. The task, the behavior, and the visualizations have been specified by the introduced RDF notation. The CtF task has been specified by a task model, the behavior by a function model, and the visualizations by a visualization model.

The task of the Vis-agents is to visualize the different states by a state diagram. Therefore, the VP-agent needs to identify the correct Vis-agent. Finally, the application has proven the correctness of our models and it was possible to identify a visualization.

5.6.4 Virtual Test Bench

Wolfgang Müller and Tao Xie

The complexity of self-optimizing systems requires a systematic and thorough verification methodology in order to guarantee their adaptive run-time behavior. In the context of the VE, our test bench is based on the principles of mutation analysis,

which we have extended towards a self-optimizing Virtual Test Bench (VTB) for the simulation-based analysis of self-optimizing systems.

Mutation analysis defines a unique coverage metric that assesses the quality of test cases of a test bench with respect to coding errors. It was originally introduced for software testing in the 90's [48]. Since 2007, mutation analysis was adopted for Register-Transfer Level (RTL) hardware design verification [191]. At that time, the professional mutation analysis tool Certitude(TM)d was introduced by CERTESS (now Synopsys) with the support of VHDL, Verilog, and C [89].

The remainder of this subsection first outlines the basic principles of mutation analysis before our self-optimizing test bench with a brief BeBot robot [99] application example is introduced.

5.6.4.1 Mutation Analysis and Simulation

Mutation testing is a fault-based simulation metric. It highlights an intrinsic requirement on simulation test data that they should be capable of stimulating potential design coding errors and propagating the erroneous behavior to check points. Mutation testing measures and enhances a simulation process as shown by Fig. 5.89.

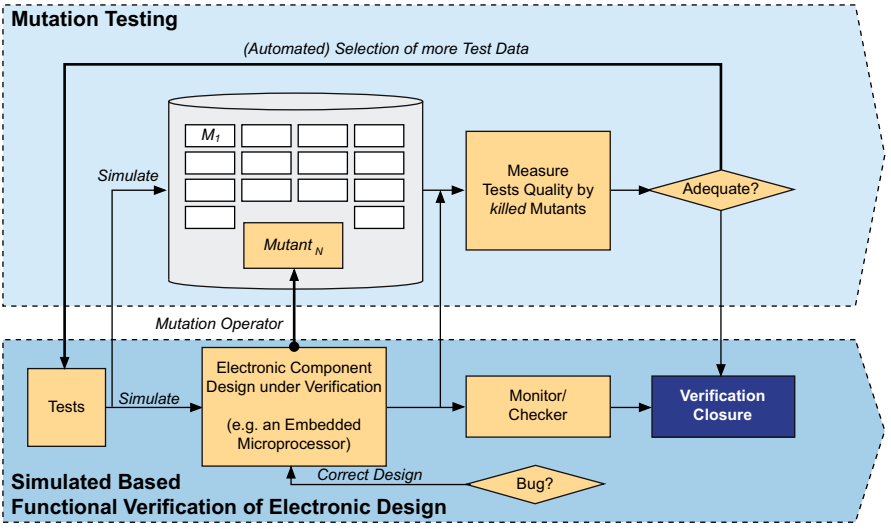

Fig. 5.89 Principle of mutation testing for the functional verification of electronic component designs

A so-called mutation is a single fault injection into a copy of the design under verification, such as this HDL statement modification:

$$\text{a} <= \text{b and c;} \xrightarrow{mutation} \text{a} <= \text{b or c;}$$

The fault-injected copy is denoted as a mutant of the design. For each test case, the mutant is simulated after the simulation of the original design and both simulation results are compared. If any simulation difference appears at the design output, this test is said to be able to kill the mutant. Each type of fault injection is called a mutation operator and dozens of such operators can be defined based on the design language under consideration. By applying these pre-defined mutation operators at different locations of a design, we can obtain a huge database of mutants. The number of killed mutants becomes the mutation coverage metric and measures the overall quality and thoroughness of a simulation process.

We consider employing random simulation as a long recognized useful lightweight method to support mutation testing. However, the lightweight nature of random simulation will conflict with the inherent computation expensiveness of mutation testing. Basically, each time a random test is generated, it should be simulated against not only the original design under verification but also all the mutants that are created as the coverage points, which can be numerous. Since the test is randomly selected and relatively aimless, this amplifies the mutation testing problem. We have addressed that problem by developing an approach for a self-optimizing test bench, which is outlined in the next paragraphs.

5.6.4.2 Self-optimizing Virtual Test Bench

Our self-optimizing Virtual Test Bench is based on the combination of mutation analysis with constrained random test pattern generation. Constrained random test pattern generation is a technique, which has been introduced in conjunction with the principles of functional verification and is an offline method to generate random test patterns for intervals, which are defined by constraints.

We apply constrained Markov chains to enable effective adjustment to the probability model of random simulation. An efficiency-improving heuristic is proposed to make this adjustment by utilizing two-phase mutation testing results. Such a test bench is shown in Fig. 5.90. The self-optimizing Virtual Test Bench integrates an in-loop heuristics that dynamically adapts the test probability model to a more efficient distribution for mutation coverage. As such, we finally arrived at a self-optimizing simulation-based test bench integrated into our VE that achieves higher mutation coverage for VPs under test within less simulation time.

As a prerequisite for the dynamic adjustment, we need a probability model on test sequences that provides the possibility of parameter steering. We consider that an electronic component design has a precisely defined instruction interface, such as the ISA of a microprocessor, or the communication protocol of a bus controller. For this, test inputs in a random test generator are modeled in two layers as shown in Fig. 5.90. First, a Markov chain is used to represent sequences of tests. Each node models one type of test instruction. The selection probability on edges enables us to establish the correlation between mutation analysis efficiency and a short pattern of test sequence. Second, weighted constraints are defined on the fields of an instruction. This provides the possibility for steering test patterns towards more effective areas like corner cases.

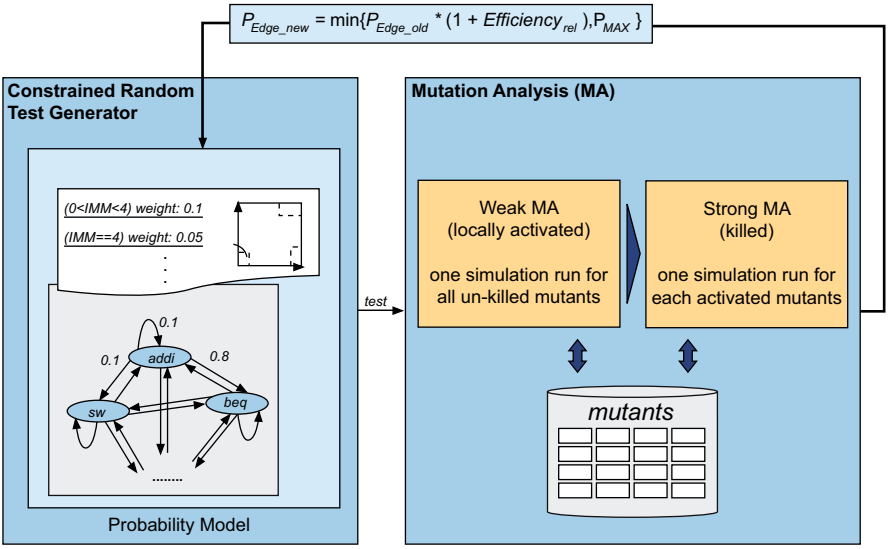

Fig. 5.90 A mutation testing directed adaptive simulation framework for the functional verification of electronic component designs

Each time a test is generated, we record the pair of Markov edge and constraint that is selected for the generation. The basic idea is to estimate the efficiency of this test on mutation analysis and use the estimation to adjust the probability of the corresponding Markov edge and constraint. This efficiency estimation should follow the unique simulation cost of mutation analysis. As the right half of Fig. 5.90 shows, we introduce at first an extra weak mutation analysis phase [104]. It uses one simulation cycle to identify the locally activated mutants. Only those are fed into a traditional, strong mutation analysis phase and fully simulated, to see, whether they are killed under the criterion that a different value appears at design output ports. Consider that φ is the test probability distribution from a Markov-chain/constraint model, which further implies $P_{M_i\_activated}$ and $P_{M_i\_kill}$ for each mutant M_i as its probabilities of being activated and killed under the current test model. On a set of N_{Mutant} design mutants, this leads to an *expected simulation effort* for the mutation analysis flow in Fig. 5.90 as

$$\max_{1\le i\le N_{mutant}} (1/P_{M_i\_kill}) + \sum_{1\le i\le N_{mutant}} (P_{M_i\_activate}/P_{M_i\_kill})$$

Based on this expected simulation effort, we use the number of mutants activated by the test $N_{activated}$ and the number of its mutants killed N_{killed} to estimate the efficiency of this test as

$$Efficiency = \frac{N_{killed}}{N_{activated}}$$

A low ratio means that too many mutants are merely activated and a lot of simulations are wasted in the second phase without killing the mutants. We also record this efficiency value for the last 10 tests generated and use the average $Efficiency_{average\_last\_ten}$ to derive a relative value that lies between 0 and 1.

$$Efficiency_{rel} = \frac{Efficiency}{Efficiency_{average\_last\_ten} + Efficiency}$$

By this, at the early stage of a random simulation, test patterns with high mutation kill/activation rates are encouraged. However, we observed in our experiment that in the last stage, it may well happen that no single mutant is killed in ten consecutive iterations. In such a case, the heuristic approach *changes to another mode that encourages more activation of mutants*, by first calculating *efficiency* as an adjustment value and then increasing the probability/weight of the corresponding Markov chain edge/constraint with the following value:

$$Efficiency_{rel_activation_mode} = \frac{N_{activated}}{N_{activated\_average\_last\_ten} + N_{activated}}$$

Here, it is safe for us to assume that there will always be some mutants activated. Initially, all Markov chain edges have the same probability to be selected and instruction constraints have the same weight. At the end of each iteration for test generation, the probability of the used edge, as well as the weight of the used constraint is adjusted by

$$\begin{cases} P_{Edge\_new} = \min\{P_{Edge_old} * (1 + Efficiency_{rel}), P_{MAX}\} \\ P_{Constr\_new} = \min\{W_{Constr_old} * (1 + Efficiency_{rel}), W_{MAX}\} \end{cases}$$

P_{MAX} and W_{MAX} are efforts to prevent the starvation of other edges/constraints, by setting an upper bound of probability to one edge/constraint. In the following example, with a model of 58 Markov edges, we set these two numbers to 0.9.

For each $Edge_i$ that flows out from the same instruction node and each $Constr_i$ on this node, we adjust their probability/weight proportionally to their old values

$$\begin{cases} P_{Edge_i\_new} = (1 - P_{Edge\_new}) * \dfrac{P_{Edge_i\_old}}{1 - P_{Edge_i\_old}} \\ P_{Constr_i\_new} = (1 - P_{Constr\_new}) * \dfrac{P_{Constr_i\_old}}{1 - P_{Constr_i\_old}} \end{cases}$$

5.6.4.3 Application Example

We applied our self-optimizing Virtual Test Bench to the BeBot robots [99] in order to indicate the strength and also the current limits of the approach in the context of a Virtual Prototyping Environment. As such, we consider a path finding algorithm implemented in C as a design under test, which navigates the BeBot by means of 12 infrared sensors inside the VE of a randomly generated labyrinth. For test automation, we used an automatically generated configuration file to parameterize each

simulation run, such as the terrain of the environment, starting point and target of the BeBot. A configuration generator tries to dynamically improve the test bench by utilizing results from the mutation analysis. After each run, the test bench monitored, whether the BeBot successfully finished the predefined route. The code of the path finding algorithm was mutated by the tool Certitude(TM). After applying our self-optimization heuristics, the configuration generator improves the test bench by utilizing results from the mutation analysis.

For our application example, with our original BeBot C source file as input, CERTITUDE(TM) initially generated 184 mutants by injecting various faults. All these mutants were compiled together with the VE, in the same way as the original code. Then, each of the generated184 mutants and the original BeBot code were simulated before new configurations were generated.

Figure 5.91 (top) shows results of the BeBot test experiments, as a summary from the Certitude(TM) report, as well as examples of mutants for the first configuration of the test environment. It shows that, at the end, the test was able to detect 68 BeBot mutants, among the total 184 mutants generated.

The remaining mutants could not be detected in this test configuration and revealed the weakness of the test patterns. These included 28 non-activated, 28 non-propagated, and 60 non-detected mutants. The status of a mutant and its injected fault is measured by Certitude(TM) as follows:

- **Non-Activated:** The fault-injected mutation statement was not executed in the simulation.
- **Non-Propagated:** The mutation statement was executed, but the execution had the same result as that from the original statement in the original design simulation.
- **Non-Detected:** The mutation statement was executed and introduced a wrong-valued behavior into the mutant simulation. However, the test bench was not able to distinguish this mutant as an incorrect design.
- **Detected:** The Test bench was able to tell that we had an error in the mutant simulation.

There were two reasons for the applied test bench not being able to detect a mutant. The first reason was that the mutant is created at a location of the code, which inherently does not induce any wrong behavior in the BeBot, like, for example, some debugging statements. The second reason was that the undetected mutant indeed reveals the weakness of our test bench. It can either be that the exercise from the test bench with the current labyrinth was not sufficient to stimulate the faulty behavior, or that the stimulated erroneous behavior did not have significant impact to be observed by the test bench.

Figure 5.91 shows at the bottom an example of such undetected mutant. The mutant with ID 46 was created by a fault injection of changing an && (logical AND) operator to || (logical OR). The Virtual Test Bench could not detect this artificial bug in the BeBot code, which indicates that an improvement of the test bench is necessary.

File	Mutants (faults injected)	Non-Activated	Non-Propagated	Non-Detected	Detected
src/Bebot.c	184	28	28	60	68

Mutant detail

Fault ID	Fault Type	Status
46	Operator && to ||	Non-Detected

With the fault 46 of type 'Operator && to ||', the code:

```
if (sensor_values[7] < 0x150 && sensor_values[10] > 0x100 && sensor_values[10] < 0x400) {
```

Is changed into:

```
if (sensor_values[7] < 0x150 || sensor_values[10] > 0x100 && sensor_values[10] < 0x400) {
```

Fig. 5.91 Snapshot from CERTITUDE(TM) report for BeBot virtual test. TOP: Overall results BOTTOM: A mutant detail

Certitude(TM) is widely and successfully applied for the mutation analysis of hardware models at register-transfer level (RTL), e. g. in VHDL and Verilog, and we also successfully demonstrated our self-optimizing approach for the test bench of the MicroBlaze processor at electronic system level (ESL). In summary, our BeBot evaluations indicate that the application of Certitude(TM) also makes sense for the mutation analysis of abstract self-optimizing behavior. However, though promising, our evaluations also demonstrate there is a considerable gap between RTL and our system level applications so that further studies are still required to draw a wider conclusion.

We developed a VR- and AR-based platform for the Virtual Prototyping of self-optimized mechatronic systems with real-time user interaction. The previous section focused on the automatic configuration of VEs and on Virtual Test Bench automation. The general concepts of that framework for the integrated simulation of multi-domain VPs can be found in [17, 172]. Here, we have demonstrated the feasibility of our approach for automatic configuration VEs by means of the BeBot robot application. However, as the degree of automation is partly based on the analysis of domain-specific models, it still requires further investigation of the semantic analysis for cross-domain application and model linking.

References

1. Adelt, P., Donoth, J., Gausemeier, J., Geisler, J., Henkler, S., Kahl, S., Klöpper, B., Krupp, A., Münch, E., Oberthür, S., Paiz, C., Porrmann, M., Radkowski, R., Romaus, C., Schmidt, A., Schulz, B., Vöcking, H., Witkowski, U., Witting, K., Znamenshchykov, O.: Selbstoptimierende Systeme des Maschinenbaus. In: Heinz Nixdorf Institut, Universität Paderborn, vol. 234. HNI-Verlagsschriftenreihe, Paderborn (2009)
2. Adelt, P., Esau, N., Hölscher, C., Kleinjohann, B., Kleinjohann, L., Krüger, M., Zimmer, D.: Hybrid Planning for Self-Optimization in Railbound Mechatronic Systems. In: Naik, G. (ed.) Intelligent Mechatronics, pp. 169–194. InTech Open Access Publisher, New York (2011)
3. Adelt, P., Esau, N., Schmidt, A.: Hybrid Planning for an Air Gap Adjustment System Using Fuzzy Models. Journal of Robotics and Mechatronics 21(5), 647–655 (2009)
4. Ali, M.I.A.H., Sitte, J., Witkowski, U.: Parallel Early Vision Algorithms for Mobile Robots. In: Proceedings of the 4th International Symposium on Autonomous Minirobots for Research and Edutainment, Buenos Aires, pp. 133–140 (2007)
5. Alur, R.: Formal Verification of Hybrid Systems. In: Proceedings of the 9th ACM International Conference on Embedded Software, Taipei, pp. 273–278. ACM, New York (2011)
6. Alur, R., Courcoubetis, C., Dill, D.: Model-Checking in Dense Real-time. Information and Computation 104, 2–34 (1993)
7. Alur, R., Courcoubetis, C., Halbwachs, N., Dill, D.L., Wong-Toi, H.: Minimization of Timed Transition Systems. In: Cleaveland, W.R. (ed.) CONCUR 1992. LNCS, vol. 630, pp. 340–354. Springer, Heidelberg (1992)
8. Alur, R., Dill, D.L.: A Theory of Timed Automata. Theoretical Computer Science 126, 183–235 (1994)
9. Angluin, D.: Learning Regular Sets from Queries and Counterexamples. Information and Computation 75(2), 87–106 (1987)
10. Antoulas, A.C., Beattie, C.A., Gugercin, S.: Interpolatory Model Reduction of Large-Scale Dynamical Systems. In: Mohammadpour, J., Grigoriadis, K.M. (eds.) Efficient Modeling and Control of Large-Scale Systems, pp. 3–58. Springer, Heidelberg (2010)
11. Asada, M., Noda, S., Tawaratsumida, S., Hosoda, K.: Vision-based Reinforcement Learning for Purposive Behavior Acquisition. In: Proceedings of the IEEE International Conference on Robotics and Automation, Nagoya, pp. 146–153 (1995)
12. Babitski, G.: Inferenzalgorithmen zur Auswahl ontologiebasierter Situationsbeschreibungen für ein kontextadaptives Dialogsystem. Ph.D. thesis, Technische Universität Darmstadt (2004)
13. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
14. Baldin, D., Kerstan, T.: Proteus, a Hybrid Virtualization Platform for Embedded Systems. In: Rettberg, A., Zanella, M.C., Amann, M., Keckeisen, M., Rammig, F.J. (eds.) IESS 2009. IFIP AICT, vol. 310, pp. 185–194. Springer, Heidelberg (2009)
15. Baolu, G., Shibo, X., Meili, C.: Research and Application of a Product Cooperative Design System Based on Multi-Agent. In: Proceedings of the 3rd International Symposium on Intelligent Information Technology Application, Nan Chang, pp. 198–201 (2009)
16. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the Art of Virtualization. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles, Bolton Landing (2003)

17. Bauch, J., Radkowski, R., Zabel, H.: An Explorative Approach to the Virtual Prototyping of Self-optmizing Mechatronic Systems. In: Proceedings of the ProSTEP iViP Science Days - Cross Domain Engineering, Darmstadt (2005)
18. Becker, S., Brenner, C., Brink, C., Dziwok, S., Heinzemann, C., Löffler, R., Pohlmann, U., Schäfer, W., Suck, J., Sudmann, O.: The MechatronicUML Design Method - Process, Syntax, and Semantics. Tech. Rep. tr-ri-12-326, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn (2012)
19. Bellifemine, F.L., Caire, G., Greenwood, D.: Developing Multi-agent Systems with JADE. John Wiley & Sons, Hoboken (2007)
20. Ben-Gal, I.: Bayesian Networks. In: Encyclopedia of Statistics in Quality and Reliability (2007)
21. Bengtsson, J.E., Yi, W.: Timed Automata - Semantics, Algorithms and Tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004)
22. Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. Scientific American (2001)
23. Berthelot, F., Nouvel, F., Houzet, D.: Partial and Dynamic Reconfiguration of FPGAs: A Top Down Design Methodology for an Automatic Implementation. In: Proceedings of the 20th International Parallel and Distributed Processing Symposium, Rhodes (2006)
24. Bertin, J.: Semiology of Graphics. University of Wisconsin Press, Wisconsin (1983)
25. Beyer, D., Henzinger, T.A., Théoduloz, G., Zufferey, D.: Shape Refinement Through Explicit Heap Analysis. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 263–277. Springer, Heidelberg (2010)
26. Blesken, M., Ruckert, U., Steenken, D., Witting, K., Dellnitz, M.: Multiobjective Optimization for Transistor Sizing of CMOS Logic Standard Cells Using Set-oriented Numerical Techniques. In: Proceedings of the 27th Norchip Conference, Trondheim, pp. 1–4 (2009)
27. Bludau, C., Welp, E.: Semantic Web Services for the Knowledge-based Design of Mechatronic Systems. In: Proceedings of the International Conference on Engineering Design, Melbourne (2005)
28. Böke, C.: Software Synthesis of Real-Time Communication System Code for Distributed Embedded Applications. In: Proceedings of the 6th Annual Australasian Conf. on Parallel and Real-Time Systems, Melbourne (1999)
29. Böke, C.: Automatic Configuration of Real-Time Operating Systems and Real-Time Communication Systems for Distributed Embedded Applications. Ph.D. thesis, Fakultät für Elektrotechnik, Informatik und Mathematik, Universität Paderborn, HNI-Verlagschriftenreihe, Band 142, Paderborn (2003)
30. de Boor, C.: A Practical Guide to Splines. Springer, Heidelberg (2001)
31. Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., Yovine, S.: Kronos - A Model-checking Tool for Real-time Systems. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 546–550. Springer, Heidelberg (1998)
32. Brutzman, D., Zyda, M., Pullen, M., Morse, K.: XMSF 2002 Findings and Recommendations (2002)
33. Burmester, S., Gehrke, M., Giese, H., Oberthür, S.: Making Mechatronic Agents Resource-Aware to Enable Safe Dynamic Resource Allocation. In: Proceedings of the 4th ACM International Conference on Embedded Software, Pisa (2004)
34. Campos, C., Junge, O., Ober-Blöbaum, S.: Higher Order Variational Time Discretization of Optimal Control Problems. In: Proceedings of the 20th International Symposium on Mathematical Theory of Networks and Systems, Melbourne (2012)

35. Chinapirom, T., Kaulmann, T., Witkowski, U., Rueckert, U.: Visual Object Recognition by 2D-Color Camera and On-Board Information Processing for Minirobots. In: Proceedings of the FIRA Robot World Congress, Busan (2004)
36. Chivukula, R.P., Böke, C., Rammig, F.J.: Customizing the Configuration Process of an Operating System Using Hierarchy and Clustering. In: Proceedings of the 5th IEEE International Symposium on Object-oriented Real-time Distributed Computing, Crystal City, pp. 280–287 (2002)
37. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (2000)
38. Commuri, S., Tadigotla, V., Sliger, L.: Task-based Hardware Reconfiguration in Mobile Robots Using FPGAs. Journal of Intelligent and Robotic Systems 49(2), 111–134 (2007)
39. Dahmann, J.S., Fujimoto, R.M., Weatherly, R.M.: The Department of Defense High Level Architecture. In: Proceedings of the 29th Conference on Winter Simulation, Atlanta, pp. 142–149 (1997)
40. David, A., Behrmann, G., Bulychev, P., Byg, J., Chatain, T., Larsen, K.G., Pettersson, P., Rasmussen, J.I., Srba, J., Yi, W., Joergensen, K.Y., Lime, D., Magnin, M., Roux, O.H., Traonouez, L.M.: Tools for Model-Checking Timed Systems. In: Roux, O.H., Jard, C. (eds.) Communicating Embedded Systems - Software and Design, pp. 165–225 (2009)
41. Dell'Aere, A.: Multi-Objective Optimization in Self-Optimizing Systems. In: Proceedings of the 32nd Annual Conference on IEEE Industrial Electronics, Paris, pp. 4755–4760 (2006)
42. Dell'Aere, A.: Numerical Methods for the Solution of Bi-level Multi-objective Optimization Problems. Ph.D. thesis, Fakultät für Elektrotechnik, Informatik und Mathematik, Universität Paderborn, HNI-Verlagschriftenreihe, Paderborn (2008)
43. Dell'Aere, A., Hirsch, M., Klöpper, B., Köster, M., Krupp, A., Krüger, M., Müller, T., Oberthür, S., Pook, S., Priesterjahn, C., Romaus, C., Schmidt, A., Sondermann-Wölke, C., Tichy, M., Vöcking, H., Zimmer, D.: Verlässlichkeit selbstoptimierender Systeme - Potenziale nutzen und Risiken vermeiden, vol. 235. HNI-Verlagsschriftenreihe, Paderborn (2009)
44. Dellnitz, M., Froyland, G., Junge, O.: The Algorithms Behind GAIO - Set Oriented Numerical Methods for Dynamical Systems. In: Fiedler, B. (ed.) Ergodic Theory, Analysis, and Efficient Simulation of Dynamical Systems, pp. 145–174. Springer, Heidelberg (2001)
45. Dellnitz, M., Ober-Blöbaum, S., Post, M., Schütze, O., Thiere, B.: A Multi-objective Approach to the Design of low Thrust Space Trajectories Using Optimal Control. Celestial Mechanics and Dynamical Astronomy 105(1), 33–59 (2009)
46. Dellnitz, M., Schütze, O., Hestermeyer, T.: Covering Pareto Sets by Multilevel Subdivision Techniques. Journal of Optimization Theory and Application 124(1), 113–136 (2005)
47. Dellnitz, M., Witting, K.: Computation of robust Pareto points. International Journal of Computing Science and Mathematics 2(3), 243–266 (2009)
48. DeMillo, R.A., Offutt, A.J.: Constraint-based Automatic Test Data Generation. IEEE Transactions on Software Engineering 17(9) (1991)
49. Ding, L., Davies, D., McMahon, C.A.: The Integration of Lightweight Representation and Annotation for Collaborative Design Representation 19(4), 223–238 (2009)
50. Ding, L., Matthews, J., Mullineux, G.: Annacon - Annotation with Constrains to Support Design. In: Proceedings of the International Conference on Engineering Design, Stanford, pp. 5–48 (2009)

51. Ditze, C.: Towards Operating System Synthesis. Ph.D. thesis, Fachgruppe Entwurf Paralleler Systeme, Universität Paderborn, HNI-Verlagsschriftenreihe, Band 76, Paderborn (2000)
52. Ebied, H.M., Witkowski, U., Rueckert, U., Abdel-Wahab, M.S.: Robot Localization Based on Visual Landmarks. In: Filipe, J., Andrade-Cetto, J., Ferrier, J.L. (eds.) Proceedings of the 5th IEEE International Conference on Informatics in Control, Automation and Robotics, Funchal, pp. 49–53 (2008)
53. Eckardt, T., Heinzemann, C., Henkler, S., Hirsch, M., Priesterjahn, C., Schäfer, W.: Modeling and Verifying Dynamic Communication Structures Based on Graph Transformations. Computer Science - Research and Development 28, 3–22 (2013)
54. Eckardt, T., Henkler, S.: Component Behavior Synthesis for Critical Systems. In: Giese, H. (ed.) ISARCS 2010. LNCS, vol. 6150, pp. 52–71. Springer, Heidelberg (2010)
55. Eckardt, T., Henkler, S.: Synthesis of Reconfiguration Charts. Tech. Rep. tr-ri-10-314, Software Engineering Group, University of Paderborn (2010)
56. Esau, N., Krüger, M., Rasche, C., Beringer, S., Kleinjohann, L., Kleinjohann, B.: Hierarchical Hybrid Planning for a Self-Optimizing Active Suspension System. In: Proceedings of the 7th IEEE Conference in Industrial Electronics and Applications, Singapore (2012)
57. Estler, H.C., Wehrheim, H.: Heuristic Search-based Planning for Graph Transformation Systems. In: Proceedings of the Workshop on Knowledge Engineering for Planning and Scheduling, Freiburg, pp. 54–61 (2011)
58. Feng, H.H., Kolesnikov, O.M., Fogla, P., Lee, W., Gong, W.: Anomaly Detection Using Call Stack Information. In: Proceedings of the 2003 IEEE Symposium on Security and Privacy, Berkeley (2003)
59. FG Rammig, University of Paderborn: ORCOS - Organic Reconfigurable Operating System, `https://orcos.cs.uni-paderborn.de/doxygen/html` (accessed August 12, 2013)
60. Flaßkamp, K., Murphey, T., Ober-Blöbaum, S.: Switching Time Optimization in Discretized Hybrid Dynamical Systems. In: Proceedings of the 51th IEEE Conference on Decision and Control, Maui, pp. 707–712 (2012)
61. Flaßkamp, K., Ober-Blöbaum, S.: Variational Formulation and Optimal Control of Hybrid Lagrangian systems. In: Proceedings of the 14th International Conference on Hybrid Systems: Computation and Control, Chicago, pp. 241–250. ACM Press, New York (2011)
62. Flaßkamp, K., Ober-Blöbaum, S., Kobilarov, M.: Solving Optimal Control Problems by Exploiting Inherent Dynamical Systems Structures. Journal of Nonlinear Science 22(4), 599–629 (2012)
63. Flaßkamp, K., Ober-Blöbaum, S., Ringkamp, M., Schneider, T., Schulte, C., Böcker, J.: Berechnung optimaler Stromprofile für einen 6-phasigen, geschalteten Reluktanzantrieb. In: Tagungsband Vom 8. Paderborner Workshop Entwurf mechatronischer Systeme. Heinz Nixdorf Institut Verlagsschriftreihe, Paderborn (2011)
64. Flaßkamp, K., Timmermann, J., Ober-Blöbaum, S., Dellnitz, M., Trächtler, A.: Optimal Control on Stable Manifolds for a Double Pendulum. In: Applied Mathematics and Mechanics, vol. 12, pp. 723–724. Springer, Heidelberg (2012)
65. Fox, M., Long, D.: PDDL 2.1: An Extension to PDDL for Expressing Temporal Planning Domains. Jornal of Artificial Intelligence Research, 189–208 (2003)
66. Frazzoli, E., Dahleh, M.A., Feron, E.: Maneuver-based Motion Planning for Nonlinear Systems with Symmetries. IEEE Trans. on Robotics 21(6), 1077–1091 (2005)

67. Galea, A., Borg, J., Grech, A., Farrugia, P.: Towards Intelligent Design Tools for Micro-scale components. In: Proceedings of the International Conference on Engineering Design, Stanford, pp. 5–84 (2009)
68. Gausemeier, J., Frank, U., Donoth, J., Kahl, S.: Specification Technique for the Description of Self-optimizing Mechatronic Systems. Research in Engineering Design 20(4), 201–223 (2009)
69. Gausemeier, J., Rammig, F.J., Schäfer, W., Sextro, W. (eds.): Dependability of Self-optimizing Mechatronic Systems. Springer, Heidelberg (2014)
70. Gausemeier, J., Schäfer, W., Greenyer, J., Kahl, S., Pook, S., Rieke, J.: Management of Cross-Domain Model Consistency During the Development of Advanced Mechatronic Systems. In: Proceedings of the 17th International Conference on Engineering Design, Stanford (2009)
71. Geiger, C., Lehrenfeld, G., Müller, W.: Authoring Communicating Agents in Virtual Environments. In: Proceedings of the Computer Human Interaction, Adelaide, pp. 22–29 (1998)
72. Geisler, J., Witting, K., Trächtler, A., Dellnitz, M.: Multiobjective Optimization of Control Trajectories for the Guidance of a Rail-bound Vehicle. In: Proceedings of the 17th IFAC World Congress, Seoul (2008)
73. Geisler, J., Witting, K., Trächtler, A., Dellnitz, M.: Multiobjective Optimization of Control Trajectories for the Guidance of a Rail-bound Vehicle. In: Proceedings of the 17th World Congress International Federation of Automatic Control, Milano (2008)
74. Ghallab, M., Nau, D., Traverso, P.: Automated Planning - Theory and Practice. Elsevier, Amsterdam (2004)
75. Giese, H., Tichy, M., Burmester, S., Schäfer, W., Flake, S.: Towards the Compositional Verification of Real-time UML Designs. In: Proceedings of the 9th European Software Engineering Conference Held Jointly with the 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Helsinki, pp. 38–47. ACM Press, New York (2003)
76. Gill, P.E., Jay, L.O., Leonard, M.W., Petzold, L.R., Sharma, V.: An SQP Method for the Optimal Control of Large-scale Dynamical Systems. Journal of Computational and Applied Mathematics 120, 197–213 (2000)
77. Gilles, K., Groesbrink, S., Baldin, D., Kerstan, T.: Proteus Hypervisor - Full Virtualization and Paravirtualization for Multi-Core Embedded Systems. In: Proceedings of the International Embedded Systems Symposium, Paderborn (2013)
78. Greenyer, J., Kindler, E.: Comparing Relational Model Transformation Technologies: Implementing Query/View/Transformation with Triple Graph Grammars. Software and Systems Modeling 9, 21–46 (2010)
79. Greenyer, J., Pook, S., Rieke, J.: Preventing Information Loss in Incremental Model Synchronization by Reusing Elements. In: Proceedings of the 7th European Conference on Modelling Foundations and Applications, Birmingham (2011)
80. Griese, B., Oberthür, S., Porrmann, M.: Component Case Study of a Self-optimizing RCOS/RTOS System: A Reconfigurable Network Service. In: Proceedings of the International Embedded Systems Symposium - From Specification to Embedded Systems Application, Manaos, pp. 267–277 (2005)
81. Griewank, A., Walther, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. Society for Industrial and Applied Mathematics, Philadelphia (2008)

82. Groesbrink, S.: A First Step Towards Real-time Virtual Machine Migration in Heterogeneous Multi-Processor Systems. In: Proceedings of the 1st Joint Symposium on System-Integrated Intelligence, Hannover (2012)
83. Groesbrink, S.: Basics of Virtual Machine Migration on Heterogeneous Architectures for Self-optimizing Mechatronic Systems - Necessary Conditions and Implementation Issues. In: Production Engineering Research & Development (11740) (2012)
84. Guckenheimer, J., Holmes, P.: Nonlinear Oscillations, Dynamical Systems, and Bifurcations of Vector Fields. In: Applied Mathematical Sciences, vol. 42, Springer, Heidelberg (1983)
85. Guleyupoglu, S., Ng, H.: Distributed Collaborative Virtual Reality Framework for System Prototyping and Training. In: Proceedings of the RTO IST Symposium on New Information Processing Techniques for Military Systems, Istanbul (2000)
86. Gutiérrez, M., Vexo, F., Thalmann, D.: Stepping into Virtual Reality. Springer, Heidelberg (2008)
87. Hagemeyer, J., Kettelhoit, B., Koester, M., Porrmann, M.: Design of Homogeneous Communication Infrastructures for Partially Reconfigurable FPGAs. In: International Conference on Engineering of Reconfigurable Systems and Algorithms, Las Vegas (2007)
88. Hagemeyer, J., Kettelhoit, B., Köster, M., Porrmann, M.: A Design Methodology for Communication Infrastructures on Partially Reconfigurable FPGAs. In: Proceedings of the 17th International Conference on Field Programmable Logic and Applications, Amsterdam (2007)
89. Hampton, M., Petithomme, S.: Leveraging a Commercial Mutation Analysis Tool for Research. In: Proceedings of the Testing Academic & Industrial Conference Practice and Research Techniques, Windsor (2007)
90. Heckel, R., Thöne, S.: Behavioral Refinement of Graph Transformation-based Models. In: Proceedings of the Workshop on Software Evolution through Transformations: Model-based vs. Implementation-level Solutions, Rom, pp. 101–111 (2005)
91. Heinzemann, C., Henkler, S.: Reusing Dynamic Communication Protocols in Self-Adaptive Embedded Component Architectures. In: Proceedings of the 14th International Symposium on Component Based Software Engineering, Boulder, pp. 109–118 (2011)
92. Heinzemann, C., Henkler, S.: Timed Story Driven Modeling. Tech. Rep. tr-ri-11-326, University of Paderborn (2011)
93. Heinzemann, C., Pohlmann, U., Rieke, J., Schäfer, W., Sudmann, O., Tichy, M.: Generating Simulink and Stateflow Models From Software Specifications. In: Proceedings of the 12h International Design Conference DESIGN, Dubrovnik (2012)
94. Heinzemann, C., Priesterjahn, C., Becker, S.: Towards Modeling Reconfiguration in Hierarchical Component Architectures. In: Proceedings of the 15th ACM SigSoft International Symposium on Component-Based Software Engineering, Bertinoro, pp. 23–28 (2012)
95. Heinzemann, C., Rieke, J., Schäfer, W.: Simulating self-adaptive component-based systems using matlab/simulink. In: Proceedings of the 7th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2013. IEEE Computer Society Press (2013)
96. Henkler, S., Meyer, J., Schäfer, W., Nickel, U.: Reverse Engineering mechatronischer Systeme. In: Proceedings of the 7th Paderborner Workshop Entwurf Mechatronischer Systeme, Paderborn (2010)

97. Henkler, S., Meyer, J., Schäfer, W., Nickel, U.A., von Detten, M.: Legacy Component Integration by the Fujaba Real-time Tool Suite. In: Proceedings of the 32nd International Conference on Software Engineering, Cape Town, vol. 2, pp. 267–270 (2010)
98. Henzinger, T.A.: The Theory of Hybrid Automata. In: Logic in Computer Science, p. 278 (1996)
99. Herbrechtsmeier, S., Witkowski, U., Rückert, U.: BeBot - A Modular Mobile Miniature Robot Platform Supporting Hardware Reconfiguration and Multi-standard Communication. In: Kim, J.-H., Ge, S.S., Vadakkepat, P., Jesse, N., Al Manum, A., Puthussery-pady, K.S., Rückert, U., Sitte, J., Witkowski, U., Nakatsu, R., Braunl, T., Baltes, J., Anderson, J., Wong, C.-C., Verner, I., Ahlgren, D. (eds.) Progress in Robotics. CCIS, vol. 44, pp. 346–356. Springer, Heidelberg (2009)
100. Hillermeier, C.: Nonlinear Multiobjective Optimization - A Generalized Homotopy Approach. Birkhäuser (2001)
101. Hitzler, P., Krötzsch, M., Rudolph, S., Sure, Y.: Semantic Web - Grundlagen. Springer, Heidelberg (2008)
102. Hölscher, C., Keßler, J.H., Krüger, M., Trächtler, A., Zimmer, D.: Hierarchical Optimization of Coupled Self-optimizing Systems. In: Proceedings of the 10th IEEE International Conference on Industrial Informatics, Beijing (2012)
103. Horta, E.L., Lockwood, J.W.: PARBIT: A Tool to Transform Bitfiles to Implement Partial Reconfiguration of Field Programmable Gate Arrays (FPGAs). Tech. rep. (2001)
104. Howden, W.E.: Weak Mutation Testing and Completeness of Test Sets. IEEE Transactions on Software Engineering 8(4) (1982)
105. Hussmann, M., Thies, M., Kastens, U., Purnaprajna, M., Porrmann, M., Rueckert, U.: Compiler-driven Reconfiguration of Multiprocessors. In: Proceedings of the Workshop on Application Specific Processors, Salzburg, pp. 3–10 (2007)
106. for Intelligent Physical Agents, F.: FIPA Propose Interaction Protocol Specification (2002), `http://www.fipa.org/specs/fipa00036/SC00036H.pdf` (accessed May 8, 2012)
107. Jantsch, A., Tenhunen, H.: Networks on Chip. Kluwer Academic Publishers, Dordrecht (2003)
108. Jennings, N.R., Wooldrige, M.: Applying Agent Technology. Applied Artificial Intelligence 9(4), 357–369 (1995)
109. Jungeblut, T., Ax, J., Porrmann, M., Rueckert, U.: A TCMS-based Architecture for GALS NoCs. In: Proceedings of the IEEE International Symposium on Circuits and Systems, Seoul (2012)
110. Jungeblut, T., Liss, C., Porrmann, M., Rueckert, U.: Design-space Exploration for Flexible WLAN Hardware. In: Zorba, N., Skianis, C., Verikoukis, C. (eds.) Cross Layer Designs in WLAN Systems, pp. 521–564. Troubador Publishing, Leicester (2011)
111. Jungmann, A., Kleinjohann, B., Kleinjohann, L., Bieshaar, M.: Efficient Color-Based Image Segmentation and Feature Classification for Image Processing in Embedded Systems. In: Proceedings of the 4th International Conference on Resource Intensive Applications and Services, St. Maarten (2012)
112. Kalte, H., Lee, G., Porrmann, M., Rückert, U.: REPLICA: A Bitstream Manipulation Filter for Module Relocation in Partial Reconfigurable Systems. In: Proceedings of the 19th International Parallel and Distributed Processing Symposium - Reconfigurable Architectures Workshop (2005)
113. Kastenberg, H., Rensink, A.: Model Checking Dynamic States in GROOVE. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 299–305. Springer, Heidelberg (2006)

114. Katzenbach, A., Haasis, S.: Virtual and Mixed Reality in a SOA Based Engineering Environment. In: Proceedings of the CIRP Design Conference Design Synthesis, Enschede (2008)
115. Kerstan, T., Oertel, M.: Design of a Real-time Optimized Emulation Method. In: Proceedings of the Design, Automation and Test in Europe, Dresden (2010)
116. Kettelhoit, B., Porrmann, M.: A Layer Model for Systematically Designing Dynamically Reconfigurable Systems. In: Proceedings of the 16th International Conference on Field Programmable Logic and Applications, Madrid (2006)
117. Klinker, G., Dutoit, A., Bauer, M., Bayer, J., Novak, V.: Fata Morgana - A Presentation System for Product Design. In: Proceedings of the International Symposium on Mixed and Augmented Reality, Darmstadt (2002)
118. Klöpper, B.: Ein Beitrag zur Verhaltensplanung für interagierende intelligente mechatronische Systeme in nicht-deterministischen Umgebungen. Ph.D. thesis, Fakultät für Wirtschaftswissenschaften, Universität Paderborn, HNI-Verlagsschriftenreihe, Band 253, Paderborn (2009)
119. Klöpper, B., Aufenanger, M., Adelt, P.: Planning for Mechatronics Systems - Architechture, Methods and Case Study. Engineering Applications of Artificial Intelligence 25(1), 174–188 (2012)
120. Koester, M., Kalte, H., Porrmann, M.: Run-time Defragmentation for Partially Reconfigurable Systems. In: Proceedings of the IFIP International Conference on Very Large Scale Integration, Madrid, pp. 109–115 (2005)
121. Koester, M., Kalte, H., Porrmann, M.: Task Placement for Heterogeneous Reconfigurable Architectures. In: Proceedings of the IEEE 2005 Conference on Field-Programmable Technology, Singapore, pp. 43–50 (2005)
122. Koester, M., Kalte, H., Porrmann, M.: Task Placement for Heterogeneous Reconfigurable Architectures. In: Proceedings of the IEEE 2005 Conference on Field-Programmable Technology, Singapore, pp. 43–50 (2005)
123. Koester, M., Luk, W., Hagemeyer, J., Porrmann, M.: Design Optimizations to Improve Placeability of Partial Reconfiguration Modules. In: Proceedings of the International Conference on Design, Automation and Test in Europe, Nice (2009)
124. Koester, M., Porrmann, M., Rückert, U.: Placement-oriented Modeling of Partially Reconfigurable Architectures. In: Proceedings of the 19th International Parallel and Distributed Processing Symposium - Reconfigurable Architectures Workshop, Phoenix (2005)
125. Köpper, B., Sondermann-Wölke, C., Romaus, C.: Probabilistic Planning for Predictive Condition Monitoring and Adaptation within the Self-Optimizing Energy Management of an Autonomous Railway Vehicle. Journal for Robotics and Mechatronics 24, 5–15 (2012)
126. Korf, S., Cozzi, D., Koester, M., Hagemeyer, J., Porrmann, M., Rückert, U., Santambrogio, M.D.: Automatic HDL-based Generation of Homogeneous Hard Macros for FPGAs. In: Proceedings of the IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines, Salt Lake City, pp. 125–132 (2011)
127. Kramer, J., Magee, J.: Analysing Dynamic Change in Software Architectures: A Case Study. In: Proceedings of the International Conference on Configurable Distributed Systems, Annapolis (1998)
128. Krause, F.L., Jansen, H., Kind, C., Rothenburg, U.: Virtual Product Development as an Engine for Innovation. In: Krause, F.L. (ed.) The Future of Product Development, pp. 703–713. Springer, Heidelberg (2007)

129. Krüger, M., Trächtler, A.: Approximation of Pareto-optimal Systems Using Parametric Model-order Reduction. In: 7th Vienna International Conference on Mathematical Modelling, Wien
130. Krüger, M., Witting, K., Dellnitz, M., Trächtler, A.: Robust Pareto Points with Respect to Crosswind of an Active Suspension System. In: Proceedings of the 1st Joint International Symposium on System-Integrated Intelligence, Hannover (2012)
131. Krüger, M., Witting, K., Trächtler, A., Dellnitz, M.: Parametric Model-order Reduction in Hierarchical Multiobjective Optimization of Mechatronic Systems. In: Proceedings of the 18th IFAC World Congress, Milano (2011)
132. Leyendecker, S., Ober-Blöbaum, S.: A Variational Approach to Multirate Integration. In: Proceedings of the 4th European Conference on Computational Mechanics, Paris (2010)
133. Leyendecker, S., Ober-Blöbaum, S.: A Variational Approach to Multirate Integration for Constrained Systems. In: Fisette, P., Samin, J.C. (eds.) Proceedings of the ECCOMAS Thematic Conference: Multibody Dynamics: Computational Methods and Applications, Brüssel (2011)
134. Leyendecker, S., Ober-Blöbaum, S., Marsden, J.E., Ortiz, M.: Discrete Mechanics and Optimal Control for Constrained Systems. Optimal Control, Applications and Methods 31(6), 505–528 (2010)
135. Li, C., McMahon, C., Newnes, L.: Annotation in Design Processes: Classification of Approcches. In: Proceedings of the International Conference on Engineering Design, Stanford, pp. 8–262 (2009)
136. Li, L., Littman, M.L., Littman, L.: Prioritized Sweeping Converges to the Optimal Value Function. Tech. Rep. DCS-TR-631 (2008)
137. Loginov, A., Reps, T., Sagiv, M.: Abstraction Refinement via Inductive Learning. In: Proceedings of the 17th International Conference on Computer Aided Verification, Edinburgh, pp. 519–533 (2005)
138. Luetkemeier, S., Porrmann, M., Jungeblut, T., Rueckert, U.: A 200 mV 32-bit Subthreshold Processor with Adaptive Supply Voltage Control. In: Proceedings of the 2012 IEEE International Solid-state Circuits Conference, San Francisco, pp. 484–485 (2012)
139. Lynch, N.A.: Distributed Algorithms, 1st edn. Morgan Kaufmann, Burlington (1997)
140. Marsden, J.E., Ratiu, T.S.: Introduction to Mechanics and Symmetry, 2nd edn. Springer, Heidelberg (1999)
141. Marsden, J.E., West, M.: Discrete Mechanics and Variational Integrators. Acta Numerica 10, 357–514 (2001)
142. Mendez, G., de Antonio, A.: An Agent-based Architecture for Collaborative Virtual Environments for Training. In: Proceedings of the 5th WSEAS Int. Conf. on Multimedia, Internet and Video Technologies, Corfu, pp. 29–34 (2005)
143. Mescheder, D., Tuyls, K., Kaisers, M.: POMDP Opponent Models for Best Response Behavior. In: Proceedings of the 23rd Benelux Conference on Artificial Intelligence, Gent (2011)
144. Milner, R.: A Calculus of Communicating Systems. Springer, Heidelberg (1982)
145. Moctezuma Eugenio, J.C., Arias Estrada, M.: Hardware/Software FPGA Architecture for Robotics Applications. In: Proceedings of the 5th International Workshop on Reconfigurable Computing: Architectures, Tools and Applications, Karlsruhe, pp. 27–38 (2009)
146. Moore, A., Ober-Blöbaum, S., Marsden, J.E.: Trajectory Design Combining Invariant Manifolds with Discrete Mechanics and Optimal Control. Journal of Guidance, Control, and Dynamics 35(5), 1507–1525 (2012)

147. Moore, A.W., Atkeson, C.G.: Prioritized Sweeping - Reinforcement Learning with less Data and less Time. Machine Learning 13(1), 103–130 (1993)
148. Münch, E., Gambuzza, A., Paiz, C., Pohl, C., Porrmann, M.: FPGA-in-the-Loop Simulations with CAMEL-View. In: Proceedings of the 7th International Heinz Nixdorf Symposium, Paderborn (2008)
149. Nava, F., Sciuto, D., Santambrogio, M.D., Herbrechtsmeier, S., Porrmann, M., Witkowski, U., Rueckert, U.: Applying Dynamic Reconfiguration in the Mobile Robotics Domain - A Case Study on Computer Vision Algorithms. ACM Transactions on Reconfigurable Technology and Systems 4(3), 29:1–29:22 (2011)
150. Niemann, J.C., Puttmann, C., Porrmann, M., Rückert, U.: Resource Efficiency of the GigaNetIC Chip Multiprocessor Architecture. Journal of Systems Architecture (JSA), Special Issue on Architectural Premises for Pervasive Computing 53(5-6), 285–299 (2007)
151. Ober-Blöbaum, S., Junge, O., Marsden, J.E.: Discrete Mechanics and Optimal Control: An Analysis. Control, Optimisation and Calculus of Variations 17(2), 322–352 (2011)
152. Ober-Blöbaum, S., Ringkamp, M., Zum Felde, G.: Solving Multiobjective Optimal Control Problems in Space Mission Design using Discrete Mechanics and Reference Point Techniques. In: Proceedings of the 51th IEEE Conference on Decision and Control, Maui, pp. 5711–5716 (2012)
153. Ober-Blöbaum, S., Walther, A.: Computation of Derivatives for Structure Preserving Optimal Control Using Automatic Differentiation. Proceedings of Applied Mathematics and Mechanics 10(1), 585–586 (2010)
154. Oberthür, S.: Towards an RTOS for Self-optimizing Mechatronic Systems. Ph.D. thesis, Fakultät für Elektrotechnik, Informatik und Mathematik, Universität Paderborn, HNI-Verlagschriftenreihe, Paderborn (2009)
155. Oberthür, S.: Towards an RTOS for Self-optimizing Mechatronic Systems. Ph.D. thesis, Fakultät für Elektrotechnik, Informatik und Mathematik, Universität Paderborn, HNI-Verlagschriftenreihe, Paderborn (2010)
156. Oberthür, S., Böke, C.: Flexible Resource Management - A Framework for Self-optimizing Real-time Systems. In: Gao, G.R., Kopetz, H., Kleinjohann, L., Rettberg, A. (eds.) Proceedings of IFIP Working Conference on Distributed and Parallel Embedded Systems, Toulouse (2004)
157. Oberthür, S., Zaramba, L., Lichte, H.S.: Flexible Resource Management for Self-X Systems: An Evaluation. In: Proceedings of the 1st IEEE Workshop on Self-Organizing Real-Time Systems, Carmona (2010)
158. Pahl, G., Beitz, W., Feldhusen, J., Grote, K.H.: Engineering Design - A Systematic Approach, 3rd edn. Springer, Heidelberg (2007)
159. Paiz, C., Hagemeyer, J., Pohl, C., Porrmann, M., Rückert, U., Schulz, B., Peters, W., Böcker, J.: FPGA-Based Realization of Self-Optimizing Drive-Controllers. In: Proceedings of the 35th Annual Conference of the IEEE Industrial Electronics Society, Porto (2009)
160. Panzer, H., Mohring, J., Eid, R., Lohmann, B.: Parametric Model Order Reduction by Matrix Interpolation. at - Automatisierungstechnik 58, 475–484 (2010)
161. Payne, T.: Agent-based Team Aiding in a Time Critical Task. In: Proceeding of the 44rd Hawaii International Conference on System Sciences, Maui, vol. 1 (2000)
162. Pohl, C., Paiz, C., Porrmann, M.: A Hardware-in-the-Loop Design Environment for FPGAs. In: Proceedings of the Design, Automation and Test in Europe, München (2008)

163. Pohl, C., Paiz, C., Porrmann, M.: vMAGIC - VHDL Manipulation and Automation for Reliable System Development. In: Proceedings of the 3rd International Workshop on Reconfigurable Computing Education, Karlsruhe (2008)
164. Porrmann, M., Hagemeyer, J., Pohl, C., Romoth, J., Strugholtz, M.: RAPTOR - A Scalable Platform for Rapid Prototyping and FPGA-based Cluster Computing. In: Proceedings of the Parallel Computing: From Multicores and GPUs to Petascale, Lyon, pp. 592–599 (2010)
165. Porrmann, M., Purnaprajna, M., Puttmann, C.: Self-optimization of MPSoCs Targeting Resource Efficiency and Fault Tolerance. In: Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems, San Francisco, pp. 467–473 (2009)
166. Priesterjahn, C.: Hazard Analysis of Self-optimizing Mechatronic Systems. In: Proceedings of the Doctoral Symposium of the 7th Joint Meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Amsterdam (2009)
167. Purnaprajna, M., Porrmann, M., Rueckert, U.: Run-time Reconfigurability in Embedded Multiprocessors. SIGARCH Computer Architecture News 37(2), 30–37 (2009)
168. Purnaprajna, M., Porrmann, M., Rueckert, U., Hussmann, M., Thies, M., Kastens, U.: Runtime Reconfiguration of Multiprocessors Based on Compile-time Analysis. ACM Transactions on Reconfigurable Technology and Systems (TRETS) 3(3), 17:1–17:25 (2010)
169. Purnaprajna, M., Puttmann, C., Porrmann, M.: Power Aware Reconfigurable Multiprocessor for Elliptic Curve Cryptography. In: Proceedings of the Conference on Design, Automation and Test in Europe, München, pp. 1462–1467 (2008)
170. Puttmann, C., Niemann, J.C., Porrmann, M., Rückert, U.: GigaNoC – A Hierarchical Network-on-Chip for Scalable Chip-Multiprocessors. In: Proceedings of the 10th EUROMICRO Conference on Digital System Design, Lübeck, pp. 495–502 (2007)
171. Radkowski, R.: Towards Semantic Virtual Prototypes for Automatic Model Combination. In: Proceedings of the 20th CIRP Design Conference, Global Product Development, Nantes (2010)
172. Radkowski, R., Waßmann, H.: Augmented Reality-based Approach for the Visual Analysis of Intelligent Mechatronic Systems. In: Proceedings of the Workshop at the IDETC/CIE Design Engineering Technical Conference & Computer and Information in Engineering Conference, New York (2008)
173. Radkowski, R., Waßmann, H.: Software-agent Supported Virtual Experimental Environment for Virtual Prototypes of Mechatronic Systems. In: Proceedings of the ASME 2010 World Conference on Innovative Virtual Reality, Ames (2010)
174. Rana, V., Santambrogio, M., Sciuto, D., Kettelhoit, B., Koester, M., Porrmann, M., Rückert, U.: Partial Dynamic Reconfiguration in a Multi-FPGA Clustered Architecture Based on Linux. In: Proceedings of the 21st International Parallel and Distributed Processing Symposium: Reconfigurable Architectures Workshop, Long Beach (2007)
175. Reinold, P., Nachtigal, V., Trächtler, A.: An Advanced Electric Vehicle for the Development and Test of New Vehicle-Dynamics Control Strategies. In: Proceedings of the 6th IFAC Symposium on Advances in Automotive Control AAC, München (2010)
176. Reps, T., Sagiv, M., Loginov, A.: Finite Differencing of Logical Formulas for Static Analysis (ESOP). In: Proceedings of European Symposium on Programming, Las Vegas, vol. 32, pp. 393–412 (2003)
177. Restrepo, J.: A Visual Lexicon to Handle Semantic Similarity in Design Precedents. In: Proceedings of the 16th International Conference on Engineering Design, Paris (2007)

178. Reynolds, C.W.: Steering Behaviors For Autonomous Characters. In: Proceedings of Game Developers Conference, San Jose, pp. 763–782 (1999)
179. Richter, U., Mnif, M., Branke, J., Müller-Schloer, C., Schmeck, H.: Towards a Generic Observer/Controller Architecture for Organic Computing. In: Tagungsband vom 36, pp. 112–119. Jahrestagung der Gesellschaft für Informatik - Informatik für Menschen, Dresden (2006)
180. Rieke, J., Dorociak, R., Sudmann, O., Gausemeier, J., Schäfer, W.: Management of Cross-domain Model Consistency for Behavioral Models of Mechatronic Systems. In: Proceedings of the 12th International Design Conference, Dubrovnik (2012)
181. Ringkamp, M., Ober-Blöbaum, S., Dellnitz, M., Schütze, O.: Handling High Dimensional Problems with Multi-objective Continuation Methods via Successive Approximation of the Tangent Space. Engineering Optimization 44(9), 1117–1146 (2012)
182. Ringkamp, M., Walther, A., Reinold, P., Witting, K., Dellnitz, M., Trächtler, A.: Using Algorithmic Differentiation for the Multiobjective Optimization of a Test Vehicle. In: Proceedings of EVOLVE, Mexico City (2012)
183. Romaus, C., Bocker, J., Witting, K., Seifried, A., Znamenshchykov, O.: Optimal Energy Management for a Hybrid Energy Storage System Combining Batteries and Double Layer Capacitors. In: Proceedings of the Energy Conversion Congress and Exposition, San Jose, pp. 1640–1647 (2009)
184. Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation. Foundations, vol. 1. World Scientific Publishing Co. Inc., River Edge (1997)
185. Sagiv, M., Reps, T., Wilhelm, R.: Parametric Shape Analysis via 3-valued Logic. ACM Transactions on Programming Languages and Systems 24(3), 217–298 (2002)
186. Schenk, M., Straßburger, S., Kissner, H.: Combining Virtual Reality and Assembly Simulation for Production Planning and Worker Qualification. In: Zaeh, M., Reinhart, G. (eds.) Proceedings of the International Conference on Changeable, Agile, Reconfigurable and Virtual Production, München, pp. 411–414 (2005)
187. Schneider, T., Schulz, B., Henke, C., Witting, K., Steenken, D., Böcker, J.: Energy Transfer via Linear Doubly-fed Motor in Different Operating Modes. In: Proceedings of the International Electric Machines and Drives Conference, Miami, pp. 598–605 (2009)
188. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
189. Schütze, O., Dell'Aere, A., Dellnitz, M.: On Continuation Methods for the Numerical Treatment of Multi-objective Optimization Problems. In: Proceedings of the Practical Approaches to Multi-objective Optimization, Dagstuhl (2005)
190. Schütze, O., Witting, K., Ober-Blöbaum, S., Dellnitz, M.: Set Oriented Methods for the Numerical Treatment of Multi-objective Optimization Problems. In: Tantar, E., Tantar, A.-A., Bouvry, P., Del Moral, P., Legrand, P., Coello Coello, C.A., Schütze, O. (eds.) EVOLVE- A bridge between Probability. SCI, vol. 447, pp. 187–219. Springer, Heidelberg (2013)
191. Serrestou, Y., Beroulle, V., Robach, C.: Functional Verification of RTL Designs Driven by Mutation Testing Metrics. In: Proceedings of the 10th Euromicro Conference on Digital System Design, Lebeck, pp. 222–227 (2007)
192. Spors, K., Martin, A., Leetz, A.: Möglichkeiten fotorealistischer Visualisierungen im Produktionsprozess eines Automobils. Automobiltechnische Zeitschrift 3, 1–8 (2009)

193. Steenken, D., Wehrheim, H., Wonisch, D.: Sound and Complete Abstract Graph Transformation. In: Proceedings of the Brazilian Symposium on Formal Methods, Sao Paulo, pp. 92–107 (2011)
194. Steenken, D., Wonisch, D.: Using Shape Analysis to verify Graph Transformations in Model Driven Design. In: Proceedings of the 9th IEEE International Conference on Industrial Informatics, Lisbon, pp. 457–462 (2011)
195. Groesbrink, S., Baldin, D.: Towards Adaptive Resource Management for Virtualized Real-Time Systems. In: Proceedings of the 4th Workshop on Adaptive and Reconfigurable Embedded Systems, Beijing (2012)
196. Suck, J., Heinzemann, C., Schäfer, W.: Formalizing Model Checking on Timed Graph Transformation Systems. Tech. Rep. tr-ri-11-316, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn (2011)
197. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction (1998)
198. Szyperski, C.: Component Software: Beyond Object-oriented Programming. Addison-Wesley, Bonn (1998)
199. Thiere, B., Ober-Blöbaum, S., Pergola, P.: Detecting Initial Guesses for Trajectories in the (P)CRTBP. In: Proceedings of the AIAA/AAS Astrodynamics Specialist Conference, Toronto (2010)
200. Tichy, M., Henkler, S., Holtmann, J., Oberthür, S.: Component Story Diagrams: A Transformation Language for Component Structures in Mechatronic Systems. In: Proceedings of the 4th Workshop on Object-oriented Modeling of Embedded Real-time Systems, Paderborn, pp. 27–39 (2008)
201. Timmermann, R., Horenkamp, C., Dellnitz, M., Keßler, J.H., Trächtler, A.: Optimale Umschaltstrategien bei Aktorausfall mit Pfadverfolgungstechniken. In: Gausemeier, J., Rammig, F.J., Schäfer, W., Trächtler, A. (eds.) Tagungsband vom 9. Paderborner Workshop Entwurf mechatronischer Systeme. HNI-Verlagsschriftenreihe, Paderborn (2013)
202. Tripakis, S., Yovine, S.: Analysis of Timed Systems Using Time-abstracting Bisimulations. Formal Methods in System Design 18(1), 25–68 (2001)
203. University of Paderborn: TGG Interpreter Tool Suite (2012), `http://www.cs.uni-paderborn.de/index.php?id=tgg-interpreter` (accessed August 13, 2013)
204. Wasson, C.S.: System Analysis, Design, and Development. John Wiley & Sons, Hoboken (2006)
205. Watkins, C.J.C.H., Dayan, P.: Q-Learning. Machine Learning 8(3-4), 279–292 (1992)
206. Witting, K.: Numerical Algorithms for the Treatment of Parametric Multiobjective Optimization Problems and Applications. Ph.D. thesis, Fakultät für Elektrotechnik, Informatik und Mathematik, Universität Paderborn, HNI-Verlagschriftenreihe, Paderborn (2011)
207. Witting, K., Ober-Blöbaum, S., Dellnitz, M.: A Variational Approach to Define Robustness for Parametric Multiobjective Optimization Problems. Journal of Global Optimization (2012)
208. Witting, K., Schulz, B., Dellnitz, M., Böcker, J., Fröhleke, N.: A new Approach for Online Multiobjective Optimization of Mechatronic Systems. International Journal on Software Tools for Technology Transfer STTT 10(3), 223–231 (2008)
209. Wittke, M.: AR in der PKW-Entwicklung bei Volkswagen. In: Schenk, M. (ed.) Tagungsband zur 4. Fachtagung zu Virtual RealityIFF-Wissenschaftstage - Virtual Reality und Augmented Reality zum Planen, Testen und Betreiben technischer Systeme, Magdeburg (2007)

210. Wolf, W., Jerraya, A., Martin, G.: Multiprocessor System-on-Chip (MPSoC) Technology. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27(10), 1701–1713 (2008)
211. Wonisch, D.: Increasing the Preciseness of Shape Analysis for Graph Transformation Systems. Ph.D. thesis, Institut für Informatik, Universität of Paderborn (2010)
212. Ye, J., Badiyani, S., Raja, V., Schlegel, T.: Applications of Virtual Reality in Product Design Evaluation. In: Jacko, J.A. (ed.) Human-Computer Interaction, Part IV, HCII 2007. LNCS, vol. 4553, pp. 1190–1199. Springer, Heidelberg (2007)
213. Zhang, J., Cheng, B.H.C.: Model-based Development of Dynamically Adaptive Software. In: Proceedings of the 28th International Conference on Software Engineering, Shanghai (2006)
214. Zilberstein, S.: Using Anytime Algorithms in Intelligent Systems. AI Magazine 17(3), 73–83 (1996)