A Design Methodology for Self-Optimizing Systems

Jürgen Gausemeier, Ursula Frank, Andreas Schmidt, Daniel Steffen
Heinz Nixdorf Institute, University of Paderborn
Fürstenallee 11, D-33102, Germany
Phone.: +49 (0) 5251/60-6262, Fax: +49 (0) 5251/60-6268
E-Mail: {Juergen.Gausemeier, Ursula.Frank, Andreas.Schmidt, Daniel.Steffen}@hni.upb.de

Holger Giese, Florian Klein, Matthias Tichy
Software Engineering Group, University of Paderborn
Warburger Str. 100, D-33098 Paderborn, Germany
Phone.: +49 (0) 5251/60-3312, Fax: +49 (0) 5251/60-3530
E-Mail: {hg, fklein, mtt}@uni-paderborn.de

Abstract

Innovative self-optimizing systems which go far beyond current approaches for mechatronic products become possible when systems are enabled to optimize their own behavior at run-time. Such self-optimizing systems are characterized by their ability to endogenously modify their objectives in response to changing conditions and autonomously adapt their parameters and structure and as a result their behavior to fulfill their objectives. This paper outlines a systematic approach for the development of self-optimizing systems. The approach helps to reduce the considerable additional development efforts resulting from self-optimization by employing different forms of patterns throughout the whole development process to enable the reuse of design knowledge. At first, the employed notion of patterns covers the multiple disciplines involved as well as different phases of the development process. In addition, the patterns are used to enable a systematic transition between the different milestones of conceptual design such as the function hierarchy, the active structure, and the construction and component structure. The approach is presented using the example of autonomously driving shuttles which self-optimize their behavior.

1. Introduction

The integration of advanced information technology offers considerable potential for innovations in the field of conventional mechanical engineering. Most modern mechanical engineering products already rely on the close interaction between mechanical engineering, electronical engineering, software engineering and control engineering that is known as "mechatronics". The aim of mechatronics is to improve the behavior of technical systems by using sensors to obtain information about the environment and the system itself and processing this information to enable the system to adapt to its current situation.

Given the tremendous pace of development in information technology, we can identify further options that go far beyond mechatronics – systems with inherent intelligence. We call such systems "self-optimizing". Self-optimization can be characterized by the presence of the three joint actions of self-optimization (cf. [FGK+04], p. 22): (1) analyze current situation, (2) determine objectives, and (3) adapt system behavior. A self-optimizing system is thus capable to analyze and detect relevant modifications of the environment or the system itself, to endogenously modify its objectives in response to changing influence on the technical system from its surroundings, the user, or the system itself, and to autonomously adapt its behavior by means of parameter changes or structure changes to achieve its objectives.

The paradigm of self-optimization opens up fascinating prospects for mechanical engineering and its associated fields. The challenge is to effectively design such self-optimizing systems. Typically, self-optimizing systems are characterized by the high interconnectivity of its system elements and to other self-optimizing systems. To enable autonomous adaptation of their behavior, self-optimizing systems need to have a high degree of freedom concerning alternative structures and configurations. This implies a shift of decisions that are conventionally made at design-time towards the deployment phase of the product itself. In the Collaborative Research Center 614, "Self-Optimizing Concepts and Structures in Mechanical Engineering" (cf. [SFB-ol]), we are developing a novel design methodology for self-optimizing systems which addresses this challenge. A very important aspect of the design of self-optimizing systems constitutes the specification of their behavior. We make use of the general observation that the most successful reusable building blocks for the interaction and structuring of complex system at the design level are patterns. Patterns are best characterized by a problem specific structuring and interaction of elements (cf. [AIS+77]). Each pattern often defines a partial view on the system in form of a set of roles which has to be realized by the different system elements. This concept has found widespread use for the development of complex systems. Design patterns [GHJ+96] and architectural patterns [BMR+96] are very successfully employed in software engineering. In mechanical engineering and electrical engineering a slightly more general form of patterns – so called active principles are frequently used (cf. [PB03]).

This paper describes our approach by means of an example of autonomously driving shuttles as parts of a novel transportation system - the New Railway Technology Paderborn [NBP-ol] project. These shuttles practice self-optimization by exchanging their experience to improve performance.

We will first review relevant related work for development processes and pattern notions in Section 2. Then, the development process and its elements are outlined in Section 3. The conceptual design which rests upon the application of so-called active patterns for self-optimization follows (Section 4). In Section 5, the elaboration of the conceptual design for the software by means of design patterns and rigorously defined coordination patterns is presented. A final conclusion closes the paper.

2. Related Work

Each domain that participate in the development of self-optimizing systems – mechanical engineering, electrical engineering and software engineering – apply an own specific domain dependant developmentprocess model, e.g. in mechanical engineering [Rot00] and [PB03], in digital electronics [Esc93] and in software engineering [Pre94] and [PB96]. For the development of mechatronic systems, the domain-dependent process models are not sufficient because they do not consider the synergetic collaboration of the domains with each other. The key-aspect is within an integrative, cross-domain development process. There are several approaches for the development of mechatronic systems where the VDI guideline 2206 and the process model of Isermann constitute two of the most important models.

VDI guideline 2206 “A design methodology for mechatronic systems” depicts the current consensus of the experts. The result of the first step - the integrative conceptual design - is a domain-spanning conception of the desired system by all domain-experts, the so-called principle solution. Based on the principle solution, the subsequent elaboration takes place in parallel. The emergence of a functioning product that fulfils all requirements is guaranteed by frequent reconciliations and a coordinated system integration phase [VDI04].

Isermann is focussing on systems with intelligent control which for example include error detection and adaptation. He proposes an approach for the system design that assumes a mechanic basic-construction. In an analysis-step, potentials for optimization are identified which realize functions easier and more cost-efficient on the basis of digital electronics. Models for the evaluation of the behavior are composed and an information processing is build up which consists of controlling elements and also in parts optimizing elements [Ise99].

The Collaborative Research Center 241 “Integrated Mechatronic Systems” (IMES) has demonstrated the potential of mechatronics to increase performance on several systems ([IBH02], [GSS+00]).

None of today’s approaches are suited for the development of self-optimizing systems. New requirements for the design methodology result from the paradigm of self-optimization: self-optimization allows for autonomous decisions at runtime. Not all imaginable behavior needs to be anticipated before – a great amount of functionality is realized by the use of software which initiates and implements coordination-, adaptation- and transformation-processes of the systems. All of this needs to be considered already at the beginning of product development process. In the course of the work of SFB 614, appropriate approaches for the development of self-optimizing systems are developed. They are based on the design methodology for mechatronic systems.

An important aspect of our design methodology is the use of patterns to describe intelligent behavior of self-optimizing systems in a generalized domain- and application-independent way and to reuse already successfully applied design knowledge in new contexts. This way we regard patterns as reusable building blocks for complex system design processes. [LRS01] brings forward requirements for the discovery, characterization and catalogue of patterns as a core of the methodology for software adaptivity. However, only patterns for the late development phases of knowledge-based systems are investigated, such as self-monitoring, self-diagnosis and self-recovery. Neither there is a concept on how to apply patterns in the early phases, e.g. when designing the active structure and the principle solution, nor are the proposed patterns adequate for a cross-domain design of self-optimizing systems.

The Unified Problem-Solving Method Description Language (UPML) brings forward a concept and specification for the reuse of so-called “problem-solving methods” in the domain of artificial intelligence [GFR+04]. UPML specifies abstract patterns for problem-solving processes which describe intelligent behavior. However, the behavior is based on inference-processes in the sense of rule-based systems only. The approach cannot model a wide range of intelligent behavior beyond the inference mechanisms. Furthermore, there is no concept for the combination of the patterns with real mechatronic systems.

Patterns which are to be used for the software of mechatronic or self-optimizing systems must exhibit special properties. These properties include honoring real-time requirements, appropriate abstraction, formal semantics, and design for verification.

Patterns for the design of adaptive and safety critical software systems are presented in [SFO03]. Those patterns do not respect the real-time requirements of mechatronics. They additionally lack a formal specification.

Design patterns in the software engineering domain have typically been specified informally [GHJ+96]. The usage of informally specified patterns for safety critical software is not appropriate, since the behavior of the resulting software systems is not foreseeable. In the last years, formal pattern specifications (such as [KFG04], [KFG+03], [SH04]) have been introduced to overcome this problem. The structure as well as the behavior of the patterns must be formally specified. Based on these formal specifications, verification techniques like model checking are applicable. Verification techniques allow for proving that the software does behave in accordance with its specifications.

In [SH04], patterns are formally specified on a programming language level. For the considered domain, patterns should be specified on a modeling level, as the abstraction which is provided by the models allows for easier verification.

The Role-Based Metamodeling Language (RBML) [KFG04] is a formal pattern specification notation which can be used to express domain-specific patterns. In this approach, a pattern specification consists of (1) a static pattern specification which describes the structure of the pattern and (2) an interaction pattern specification which describes constraints on the allowed interaction between the structural pattern elements. The RBML approach does not support the specification of behavior which conforms to real-time requirements. In addition, the employed refinement notion [KFG+03] allows arbitrary refinement and thus does not enable compositional verification techniques.

Besides the observed lack of appropriate pattern notions for the principle solution as well as elaboration phase, all existing approaches are restricted to either the mechanical engineering or software engineering domain. However, for the intended development of self-optimizing systems the seamless support for reuse of design know-how in form of patterns is required such that the transitions between results of “key milestones” of the development process, like requirements, principle solution, and elaboration phase can efficiently be bridged.

3. Definition and Use of Solution Patterns

The demonstrator of our overall research-project is a rail-bound transportation system consisting of autonomous shuttle-vehicles for the transport of persons and goods. Several concepts of self-optimization are validated by means of the transportation system example. The shuttles shall use the existing railway system, travel in single- or convoy drive-mode and behave optimal in any situation. Among others, to behave optimal may mean to achieve best comfort for the passengers. Technically speaking, comfort depends on the movement and the acceleration of the shuttle-chassis which must be minimized by the implementation of appropriate compensation measures. The minimization of the acceleration is carried out by an active suspension/tilt module. Conventional control uses a so-called “skyhook”-approach for the dampening of the active suspension/tilt module. This approach allows the suspension/tilt module to adjust to the track-profile in such a way that the shuttle-chassis moves along a predefined straightened trajectory. However, once the preset trajectory is implemented in operational mode, this approach does not consider changes in the track-profile which inevitably occur because of wear-out of the tracks or the like. The aim of a self-optimizing solution is to provide experience-based trajectories for the shuttles in operational mode and to make this approach efficient by the cooperation within a whole shuttle community.

The development of self-optimizing systems of the complexity such as the New Railway Technology Paderborn project can be compared with the design-procedure of mechatronic systems e.g. in the automobile or aerospace industry. Fundamental decisions are taken in the early phases of the development process. A domain-spanning conceptual design phase constitutes how the system is going to be constructed and how the functionality can be achieved. After that a elaboration of the particular modules is conducted. The requirements are the starting point of the conceptual design phase. The functionality of the product is extracted and described in a solution-independent way with the help of a function hierarchy. Solutions of the participating domains are searched for. These solutions result in a principle solution that specifies the aimed product concept by a set of coherent partial models. Essentially, these models are the active structure, which describes the connectivity of the system elements, the raw construction and component structure which designates the shape of the single elements and their position in space and the behavior of the system.

Starting from the principle solution, the partial solutions are concretized with applying domain-specific methodologies. In the case of software controlling the self-optimization behavior, the development is carried out with the help of active patterns for self-optimization

(AP_{SO}) which specify schemas for the system-behavior. AP_{SO} are selected at conceptual design phase and realized by well-proven software patterns later on.

Different types of solutions are applied for the design of self-optimizing systems. We have developed a classification schema for these solutions and their concretization levels. As a superordinated concept we use the term pattern or solution pattern. According to [AIS+77] a pattern describes a recurrent problem within our environment and the core of a solution for this problem. The core of the solution pattern is specified by the characteristics of its elements and their collaboration. In our context, solution patterns are applied to work-out product concepts, drafts, realizations and implementations; they lead to mechanical and software components. Figure 1 depicts the overall classification scheme.

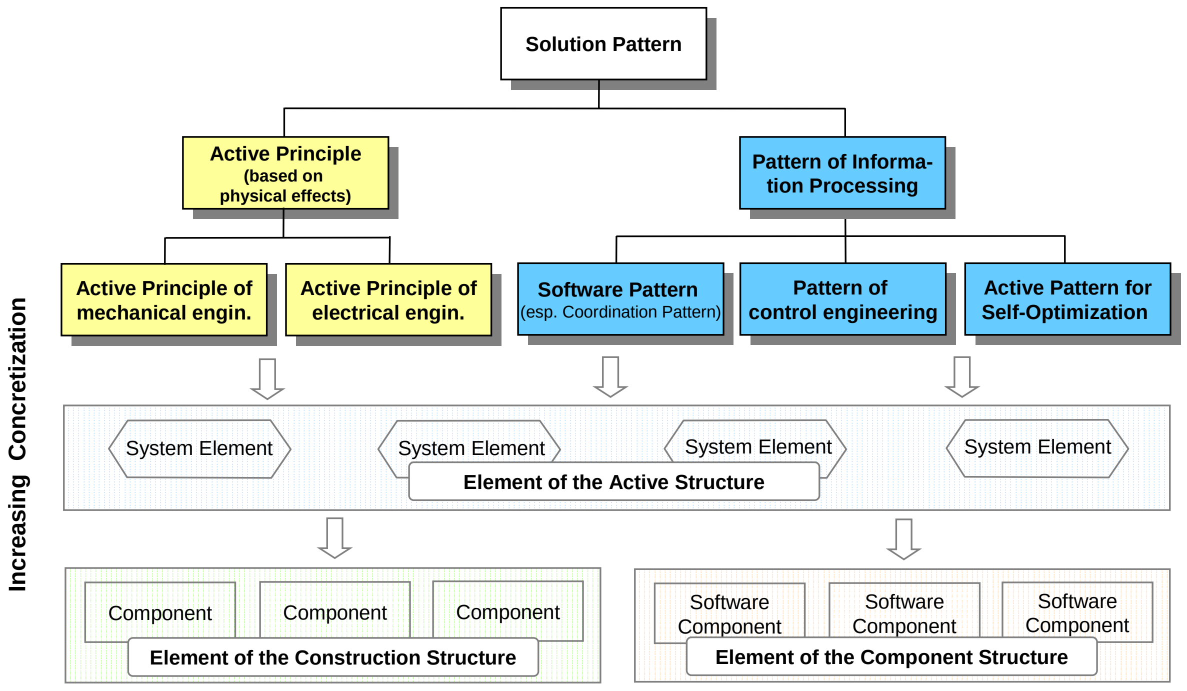
Solution Pattern
Active Principle (based on physical effects)
Pattern of Information Processing
Active Principle of mechanical engin.
Active Principle of electrical engin.
Software Pattern (esp. Coordination Pattern)
Pattern of control engineering
Active Pattern for Self-Optimization
System Element
System Element
System Element
System Element
Element of the Active Structure
Component
Component
Component
Element of the Construction Structure
Software Component
Software Component
Software Component
Element of the Component Structure
Increasing Concretization

Figure 1: Categories of Patterns

We differentiate solution patterns that are based upon physical effects and patterns that contain information processing. We call solution patterns that rely on physical effects as “Active Principles”. In particular, active principles of mechanical engineering and electrical engineering are relevant for self-optimizing systems. According to the definition of Pahl / Beitz [PB03] active principles describe the relationship of physical effects and material and geometrical characteristics (active geometry, active motions and material properties).

To a great extend, self-optimization is realized by information technology. We subsume pattern of control engineering, self-optimization and software engineering under the general term of “Pattern of Information Processing”. Software patterns consist of a problem-solution pair which makes well-proven software engineering knowledge applicable for new problem contexts. Patterns of control engineering specify how a plant is modelled, influenced or quantities are measured and observed. Active patterns for self-optimization (AP_{SO}) depict schematic solutions for the self-optimization process as described in [FGK+04]. We use the following terms at the elaboration phase: system elements constitute the elements of the active structure which is designed at the phase of conceptual design. They represent parts of the system which are not developed in detail, yet. After a further concretization, system elements with a spatial geometry are transformed into components of the construction structure and software-containing elements are transformed into software components of the component-structure. Figure 2 depicts at which phases of the design process solution patterns are applied and how they relate to each other.

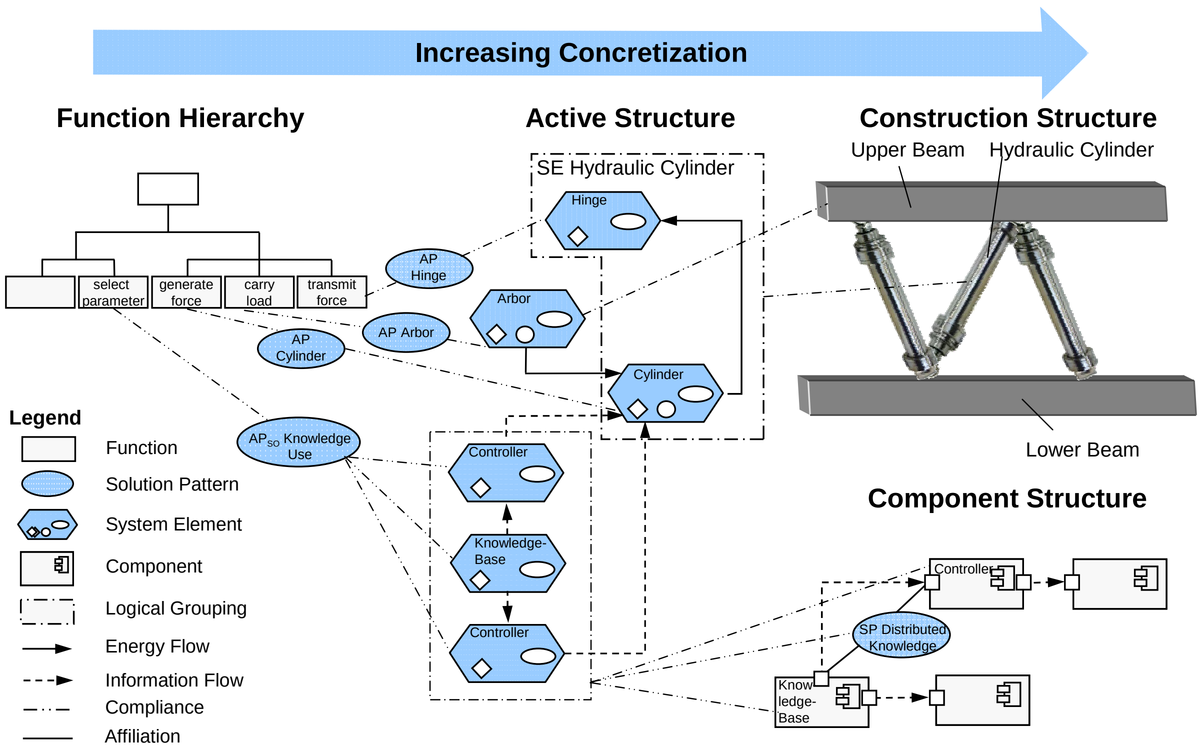

Figure 2: Transitions from Functions to Components

The design methodology is based upon two basic steps - first the conceptual design, here reduced to the transition from the function hierarchy to the active structure; second the elaboration which describes the transition from the active structure to the construction and component structure. Starting point for the first step is the function hierarchy, which specifies the product functionality. The function hierarchy primarily results from the requirements. Solutions are determined for specific functions. These may be active principles, software patterns, patterns of control engineering, active patterns for self-optimization or solution elements, if already known. By solution elements we understand a realized and well-proven solution for the fulfilment of one or more functions. In general, this means a module, a component, a group of components or a software component, which relies on one or more solution patterns. The linkage of all system elements by means of energy-, material-, and information-flow leads to the active structure. The active structure describes the physical and logical interaction between all participating system elements. Already known solution elements are treated as system elements within the active structure. Furthermore, ideas about the shape of the system arise. Therefore system elements with a spatial geometry are going to be concretized towards modules and components and are positioned in space under special consideration of geometric constraints. Thus, details about the number, shape, position, alignment and type of active surface and active location can be made. The subsequent elaboration develops the construction structure with geometry-determining components and component groups. In parallel, information-processing system elements are concretized, assembled to software components and depicted in the component structure. This is done on the basis of software patterns where applicable. The development of software for self-optimization using active patterns of self-optimization is detailed in the following sections.

4. Active Patterns for Self-Optimization

Active patterns for self-optimization (AP_{SO}) realize functions of self-optimization[1]. AP_{SO} constitute templates which specify generally accepted, autonomous and intelligent behaviour of self-optimizing systems with the help of principle-concepts, application-scenarios, structures, behaviour and methods (Figure 3). AP_{SO} cover the whole self-optimization process or only parts of it. Essential is the fact that system statechanges are caused, supported and / or deployed by autonomous, intelligent behaviour. AP_{SO} are iteratively concretized throughout the whole system development process.

The principle concept characterizes the basic idea of the AP_{SO}. It is used to allow the engineer an intuitive access to the AP_{SO}.

Application-scenarios depict situations in which the AP_{SO} have already been applied successfully in the past. Those scenarios shall help the engineer to select an appropriate AP_{SO} for the task at hand.

[1] Apart from conventional functions of mechanical engineering, we research so-called functions of self-optimization such as autonomous planning, cooperation, and learning for the description of the functionality of self-optimizing systems.

The structure specifies necessary participating system elements and their relations among each other. One or more behavior models describe adaptation processes, which an AP_{SO} shall execute.

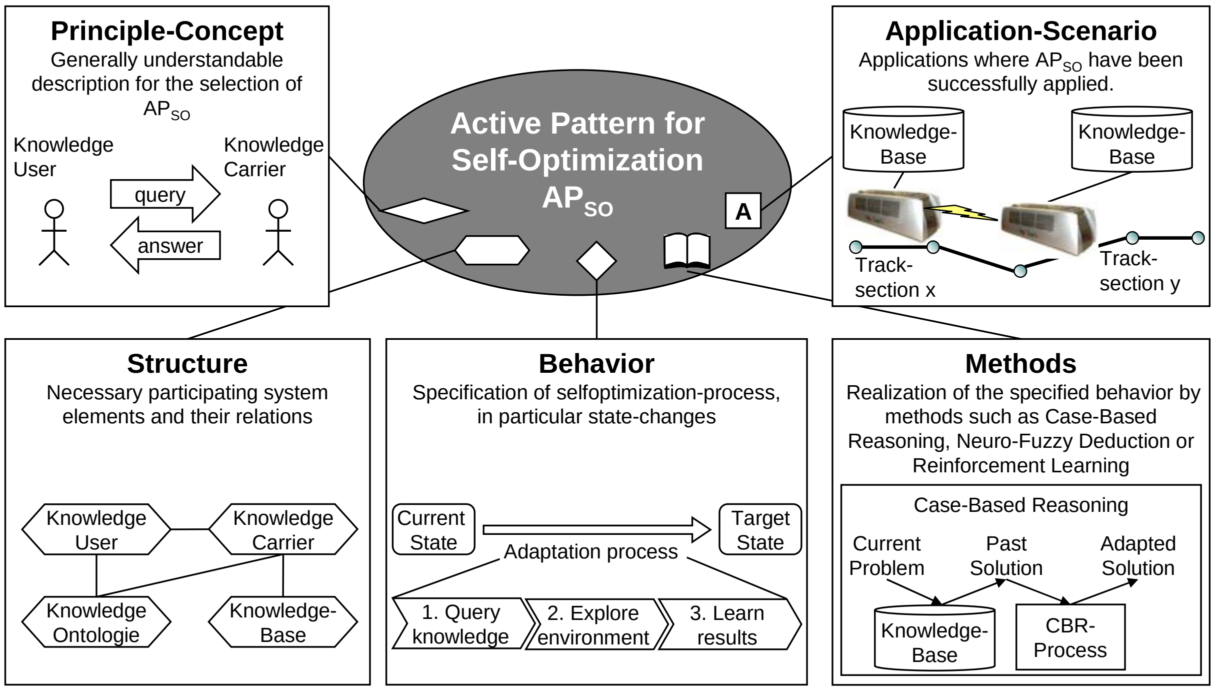

Figure 3: The Active Pattern for Self-Optimization „Experience-Based Exploration“

The focus is on the modelling of autonomous intelligent behavior, which activates, supports and/or executes the statechange. The following example is based on an adaptation process consisting of three activities:

1. Query knowledge: Knowledge of other systems is used to better achieve a task at hand.
2. Explore environment: The environment of the system is explored to enrich and extent the queried knowledge such that new experience is build up.
3. Learn results: New experience that was made when exploring the environment is learned and distributed among the participating system elements so that the knowledge-level of the whole system is continually increased with time passing by.

Finally it is shown how a system is transformed from a given current state to a desired target-state by the use of specific methods, e.g. Case-Based Reasoning for the query and adaptation of knowledge.

In the course of the conceptual design phase the experience-based exploration of trajectories is elaborated in the application-scenario “Cooperative Learning when Driving on a Track”. Starting point is the active structure of a rail-bound transportation system. Figure 4. shows an extract of the active structure where shuttles drive on track-segments that power and direct the shuttles. The active structure also describes how track deviations affect shuttles negatively.

According to the above mentioned task, the function hierarchy of the conventional rail-bound transportation system is extended by functions of self-optimization like “Determine Track-profile”, “Calculate Trajectory”, and “Adapt Behavior”.

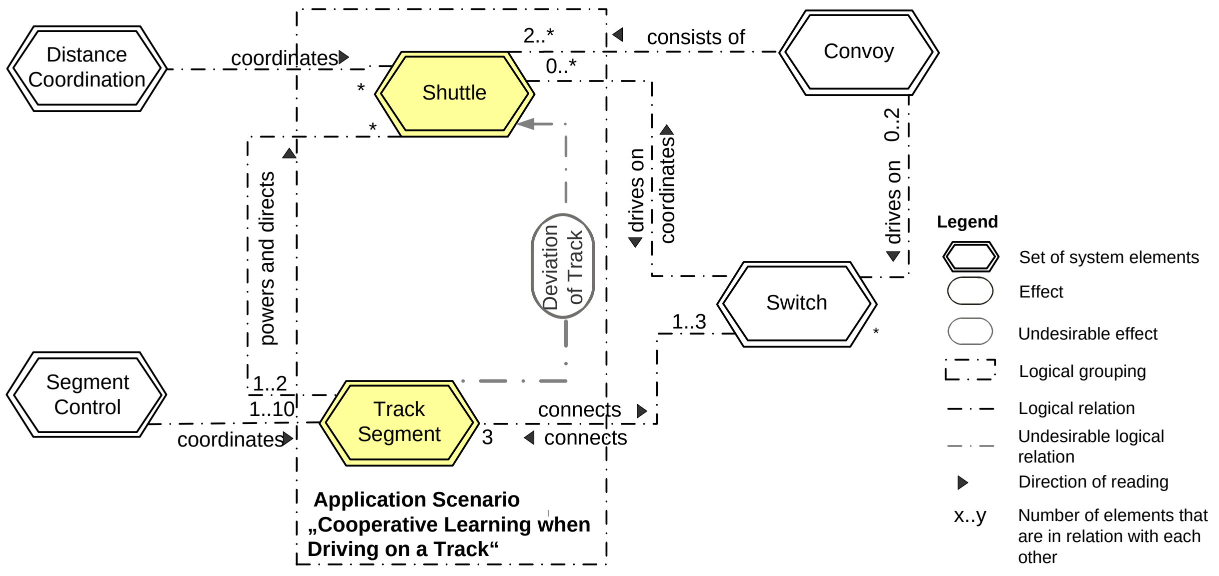

Figure 4: Extract of an Active Structure for a Rail-bound Transportation System

The active pattern for self-optimization is converted into the active structure as follows. According to the structure aspect of the AP_{SO} at least one knowledge carrier and one knowledge user is necessary. The upper part of figure 5 depicts the realization of the active patterns for self-optimization at the type-level. The middle part shows the instantiated active structure for this application-scenario. Decisive for the solution is that shuttle Sh_2 changes its state on the grounds of the access to the experience of others. In this case, the internal states are based on a mental state-space model[2]. The state-change is achieved by the above mentioned three activities: 1. query knowledge, 2. explore the environment – here: exploring the track-profile, and 3. learn from results and distribute the experience among all other system elements such as shuttles and switches.

[2] Mental state-space models are used in the domains of epistemology and artificial intelligence to model thought processes such as planning and problem solving (cf. [MA02]).

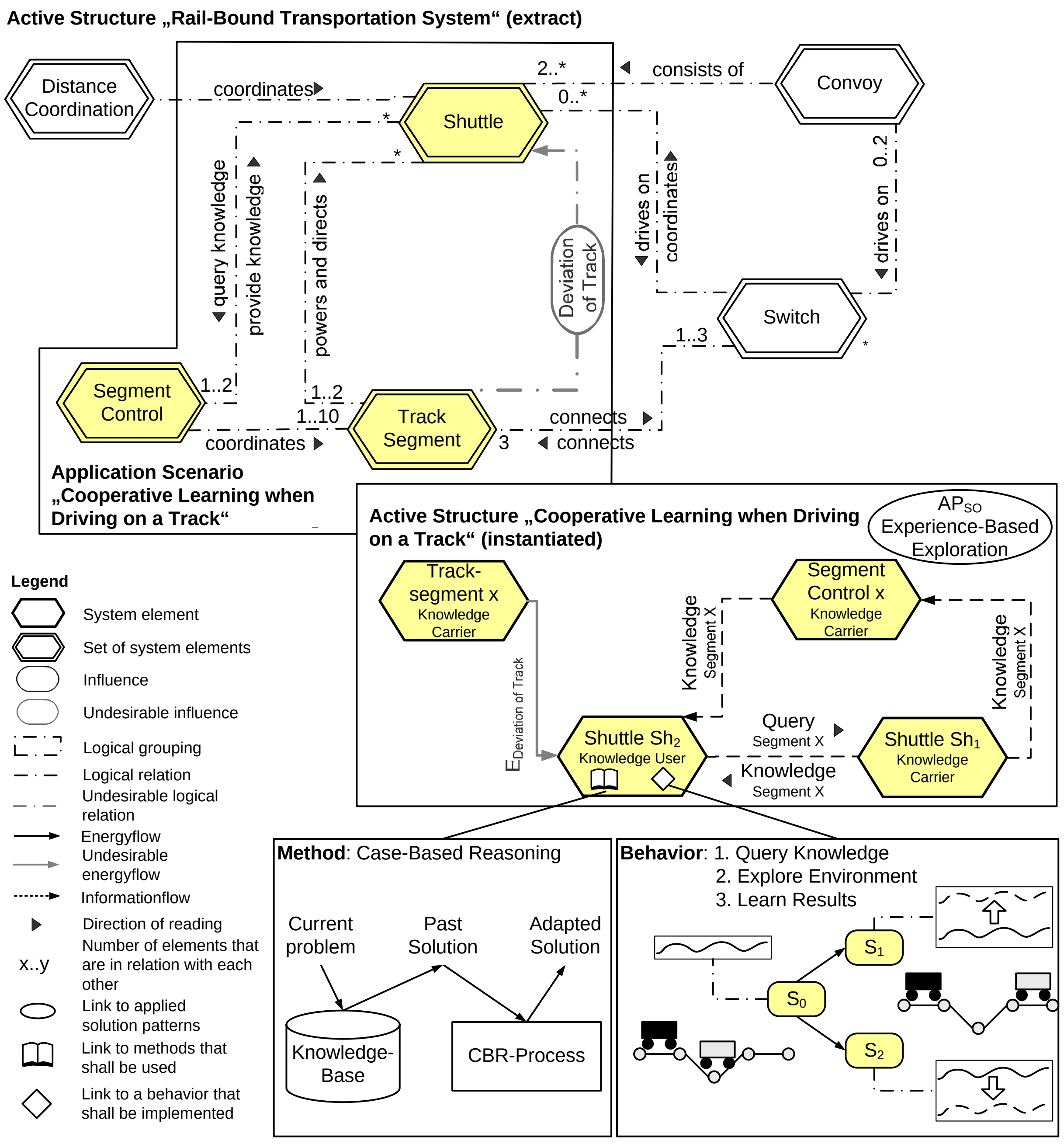

Figure 5: Active Structure for „Cooperative Learning when Driving on a Track“

The lower part of figure 5 depicts components of the active pattern of self-optimization “Experience-based Exploration” where above mentioned activities are carried out by the method of Case-Based Reasoning. This method provides a multi-criteria search for similar problems as well as the adaptation of historic solutions to the current situation. This way the knowledge of shuttle Sh_1 in terms of successfully deployed past trajectories is used for the adaptation of the behavior of shuttle Sh_2. Starting in state S_0, this initial knowledge is the basis for the exploration of an optimum trajectory. The exploration activity may lead to one of the two subsequent states S_1 and S_2. Shuttle Sh_2 of the active structure contains links to the applied AP_{SO} and the semi-formal specification of the behavior as well as to the deployed method. The exploration of reference trajectories in a multi-agent system setting is detailed in [SSO+04].

5. Elaboration – From the Principle Solution to the Component Structure

In order to realize the transition from the active structure to the component structure, (1) the relevant elements of the active structure are mapped to a corresponding UML component architecture, (2) active pattern present in the active structure have to be realized by related software patterns, and (3) additional requirements might be realized by additional software pattern (e.g., to enable the exchange of software at run-time).

Here, we further consider step (2) and present how to realize the above employed active pattern for self-optimization named “Experience-Based Exploration” in the software design. At first, we have to identify the corresponding general, reusable design pattern describing the main actors and sequences of events at the information processing level. Based on this, we have to identify in a second step the related more detailed coordination pattern which also fulfills the domain specific requirements. We also have to make sure that the relevant timing constraints are present and that the pattern can be subject to formal verification.

Design Pattern

Abstracting from the physical aspects of the system and focusing on the knowledge-related interactions, we have the Knowledge Carrier, the Knowledge Users and the Subject Matter as the principal elements (roles) of the pattern. The Knowledge Carrier has knowledge about a specific Subject Matter. The Knowledge Users ask the Knowledge Carrier about the Subject Matter and use the obtained information to guide them in their exploration. They may report their experiences back to the Knowledge Carrier. These relationships are documented by a UML Class Diagram (see Figure 6).

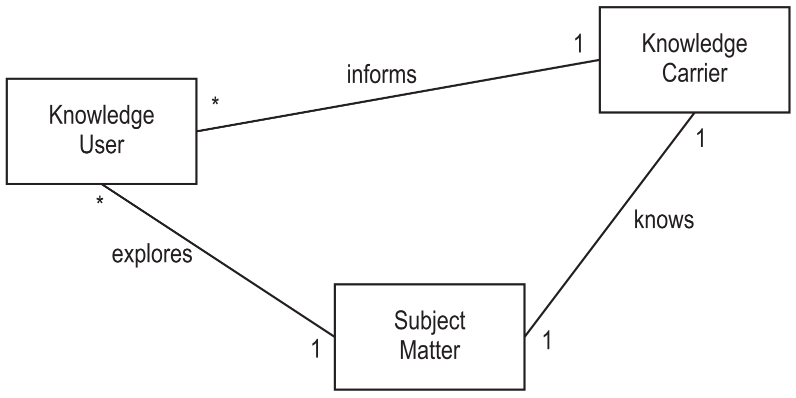

Figure 6: Class Diagram Specifying the Main Actors and their Relationships

Nonetheless, a more detailed description of the interactions is required in order for the pattern to be useful. The exchanges between the objects, or rather the roles they are playing, are thus documented by means of one or more scenarios. Scenarios are idealized sequences of concrete messages passed between roles that serve to illustrate desirable behavior.

The most important generic scenario for this pattern is depicted by the UML Sequence Diagram in Figure 7. It describes a Knowledge Carrier tailoring an optimized response to the specific query of a Knowledge User, who then proceeds to explore the Subject Matter based on this recommendation. The resulting experiences are sent back to the Knowledge Carrier, who processes them and makes them available for future users. The diagram is annotated with comments in bold face that point out the abstract steps (Query, Exploration, Learning) and the characteristic joint actions of self-optimization, (1) analyze current situation, (2) determine objectives, and (3) adapt system behavior.

Our application-scenario “Cooperative learning when driving on a track” can be seen as a specific instantiation of this pattern. The shuttle receives a trajectory encoded as a mathematical function that allows it to adapt its active suspension to the actual profile of the current track section. The parameters of the trajectory can be adapted online in order to fine-tune it in accordance with the current objectives, e.g. weighting efficiency against perceived comfort. The trajectory is based on the experiences of the shuttles that have previously used the track section.

The corresponding self-optimization process is distributed between the shuttles and the track section. The shuttle analyzes its current status concerning preferences and objectives, payload, and energy reserves (first analysis of the current situation) and communicates the results to the track section. Based on the communicated configuration and the stored experiences, the track section now computes a suitably optimized trajectory reflecting the shuttle's preferences with respect to its objectives (determination of objectives) and transmits it to the shuttle.

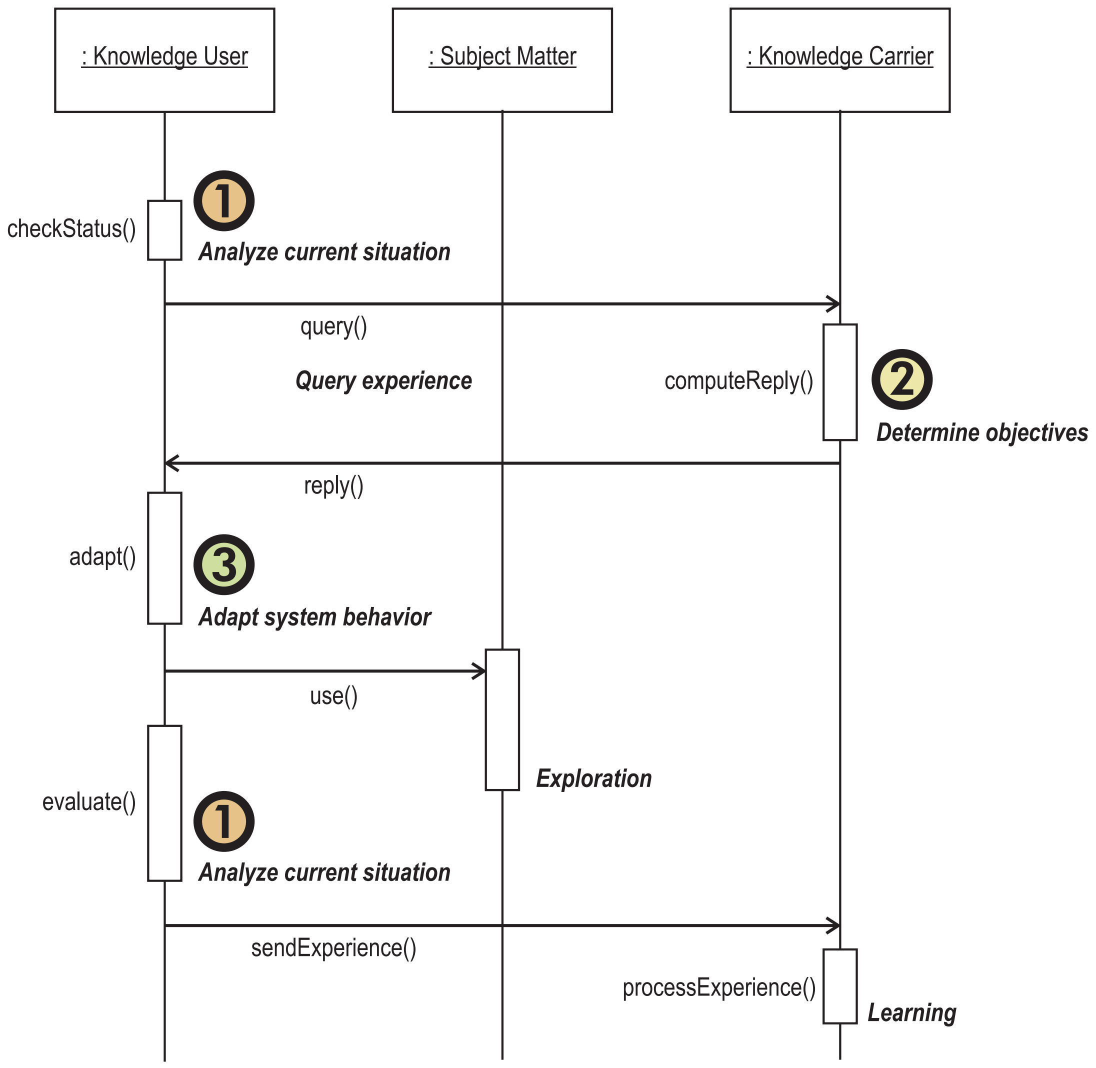

Figure 7: An Idealized Exchange between Knowledge Carrier and Knowledge User

The shuttle adapts the reference input for the active suspension system based on the trajectory (adaptation of the system behavior). After leaving the track section, the shuttle analyses the perceived comfort and the expended energy (second analysis of the current situation) and transmits its experiences back to the track section. The new experience is incorporated into the track section's repository and thus used in the trajectory optimization of subsequent shuttles. The optimization process thus depends on a distributed analysis of the current situation – the assessment of the shuttle's state by the shuttle itself (first analysis of the current situation) and the reports about the track profile by the preceding shuttles (second analysis of the current situation).

Coordination Pattern

Based on the design pattern, we can now derive a coordination pattern [GTB+03] that precisely defines the behavior that is required of the actors. While the more informal scenario captures the underlying idea in an intuitively accessible form, it does not specify which behavior is required and which is optional or incidental, nor does it provide any timing information. It is therefore insufficient as the sole specification of the – typically safety-critical – software of a mechatronic system. Coordination patterns are specifically suited to this task, as they allow the specification of verifiable real-time requirements. The diagram in Figure 8 describes the abstract structure of the derived coordination pattern. The two roles are linked by a connector representing the communication channel. On the conceptual level, the roles are linked by the communication protocol which is specified in the behavior of the two roles.

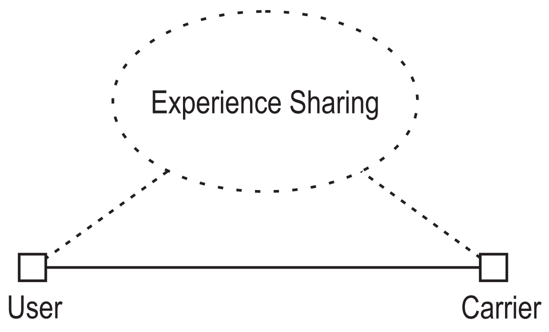

Figure 8: The Actors are Mapped to Roles of a Coordination Pattern

Our approach is based on the current UML 2.0 specification [OMG03], which in turn is based on ROOM [SGW94] and UML/RT [SR98]. Structure is specified using component diagrams; behavior is specified by a real-time variant of UML state machines called Real-Time Statecharts [BG03].

Real-Time coordination patterns (in short coordination patterns) as in Figure 8 capture the coordination behavior between abstract entities. They are subsequently applied to components, which need to implement the required coordination behavior in a way that respects all specified constraints. Coordination patterns consist of a number of abstract entities (roles) and their coordination behavior (role behavior). The role structure is specified by component diagrams; roles are displayed as ports. Communication between roles is indicated by connectors between the participating roles.

Each role of the coordination pattern is specified by a protocol state machine, i.e. a Real-Time Statechart without side effects other than message sending. As the focus of coordination patterns is on the exact specification of the (safe) interaction between components, they largely abstract from internal behavior that is irrelevant for the determination of communication behavior. Important steps like analysis, determination of goals or learning therefore do not explicitly figure in the specification, but are abstracted into non-deterministic behavior, bounded by appropriate time guards.

The User queries the Carrier for data which encode the experience. If it does not receive a reply before a certain deadline, the user has to handle the situation without this information. If, however, the information is provided in time, the user employs it and sends feedback based on the gathered experience (see Figure 9).

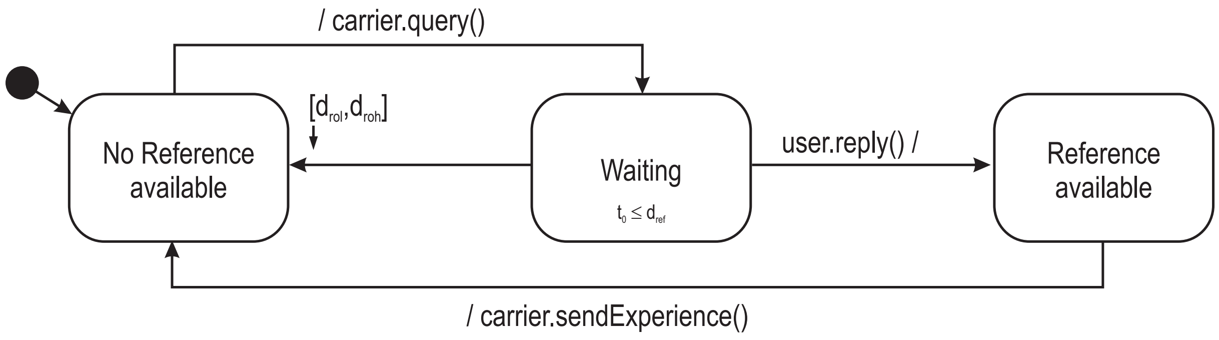

Figure 9: The Real Time Behavior of the User

The Carrier basically waits for requests by users and the experiences they send as feedback (see Figure 10).

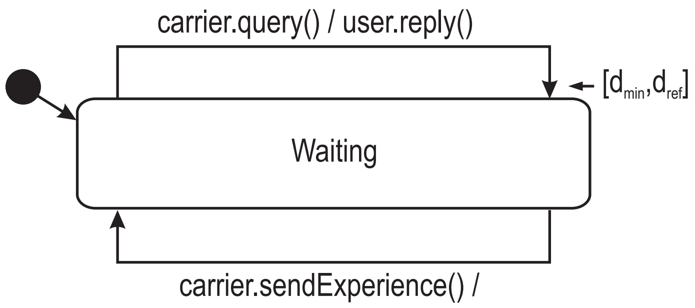

Figure 10: The Real Time Behavior of the Carrier

The behavior of the connector role is also specified by Real-Time Statecharts. The connector models the assumed properties of the communication channels like message losses, message delays, etc.

Safety critical constraints on the behavior are the final part of a coordination pattern specification. The constraints are written in OCL-RT [FM02] or a temporal logic (ATCTL).

In order to ensure that no unsafe behavior due to a violation of the constraints may occur, the model checker Uppaal [LPY97] is used to verify that all constraints for the specified behavior hold under the assumed channel behavior [BGH+04]. If the constraints hold, the pattern specification is valid and is stored in a pattern library for future reuse.

The software of a self-optimizing system consists of a number of components, which are connected via ports and channels. The components are developed by reusing some of the verified coordination patterns, which are stored in the pattern library. First, an appropriate coordination pattern is loaded from the pattern library. The roles of the pattern are added to the component as ports. A refinement of the role behavior is then added as parallel state to the behavior of the component. The refinement must respect certain restrictions in order that the results of the pattern verification still hold for the component. A component typically does not only refine one pattern role. Instead several different pattern roles are refined and added to the component behavior. Typically, a synchronization behavior is also added which coordinates between the refined roles.

Real-Time Statecharts are used for the specification of discrete event-based behavior. As mechatronic and self-optimize systems contain continuous behavior, continuous controllers are added to the components [BTG04, GBS+04]. The states of the discrete behavior are annotated by controller structures. Only, the controllers are executed during runtime which are associated with the current state of the Real-Time Statechart. This integration of continuous behavior (controllers) and discrete event-based behavior (Real-Time Statecharts) is specified using hybrid components and Hybrid Statecharts. Special fading functions are used for the specification of switching between the states and the annotated controller structures.

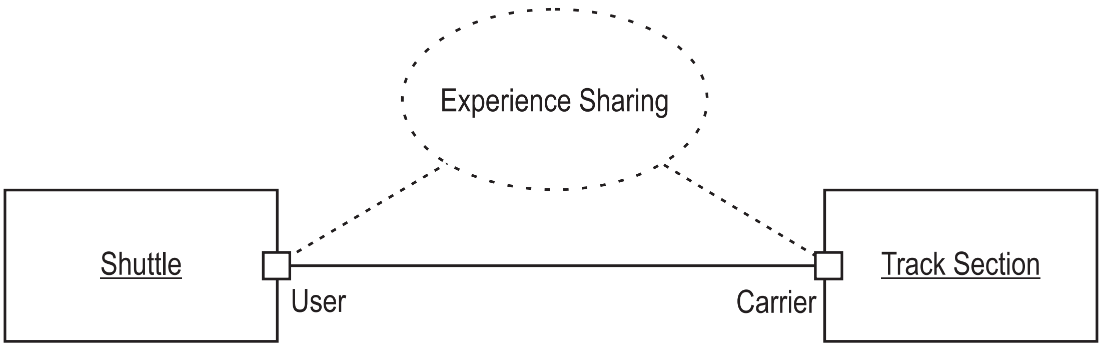

Figure 11: Shuttle and Registry Implement the Coordination Pattern

Based on the system structure, we define the components Shuttle and Track Section and apply the coordination pattern Experience Sharing to them. The Shuttle acts as the Knowledge User, the Track Section as the Knowledge Carrier. The pattern thus specifies the way shuttle and track section exchange reference trajectories and gathered experiences. The relevant part of the component structure in specified by the component diagram in Figure 11.

The Shuttle (Knowledge User) queries the Track Section (Knowledge Carrier) for a reference trajectory. If it does not receive a reply before a certain deadline, the shuttle assumes that no trajectory is available and switches to a robust controller that can safely operate the active suspension system without a reference trajectory, albeit in a less comfortable and efficient manner. If, however, the trajectory is provided in time, the shuttle uses it for traversing the track section and sends feedback based on the experience gathered through its sensors to the track section (see Figure 9).

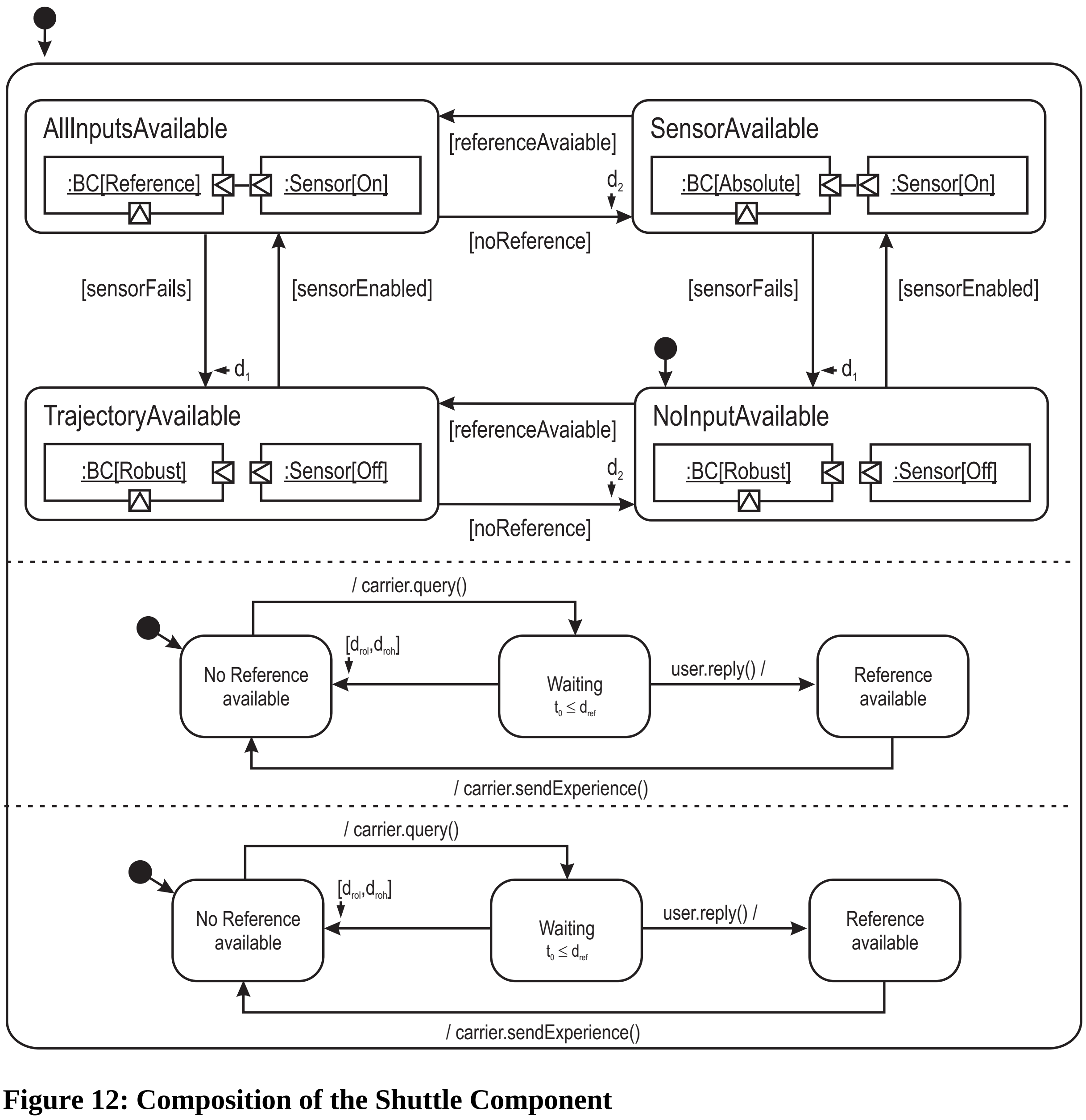

Figure 12: Composition of the Shuttle Component

The Shuttle component is defined by combining and refining all applicable coordination patterns (see Figure 12). In order to allow the shuttle to obtain the trajectory for the upcoming track section in time while still communicating with the current section, the experience sharing pattern is executed twice in parallel. It is refined by specifying additional internal communications where before there were only non-deterministic constraints.

The Hybrid Statechart in the diagram’s upper half specifies which controller should be active, depending on the available inputs. When data from the acceleration sensor is available, the Absolute controller may be used. If additionally the Track Section has provided a trajectory, the Reference controller is activated. When the required inputs fail, the system needs to quickly switch back to the Robust controller to ensure safe behavior (see [GBS+04]).

During the elaboration phase the active patterns for self-optimization, which are part of the principle solution, are refined using design patterns and subsequently coordination patterns. These coordination patterns further enable the development of systems of interacting hybrid software components which can be compositionally verified to ensure safety.

6. Conclusion

Self-optimization makes innovative mechanical systems possible which go far beyond current approaches for mechatronics. The outlined approach addresses this challenge by means of a specifically tailored design methodology. It main elements are a development process adjusted to the specific needs of self-optimizing systems, the exploration of design alternatives during the principle design, the reuse of the building blocks for self-optimization in form of active patterns of self-optimization in the principle solution, and their subsequent refinement during the elaboration of the information processing with coordination patterns within UML component diagrams.

The additional development efforts for self-optimizing systems can be drastically reduced as the generalized pattern concept is employed throughout the whole development process to enable the reuse of design knowledge. The application of the pattern covers the multiple disciplines involved such as mechanical engineering and software engineering as well as different phases such as the conceptual design and the elaboration. Furthermore, it bridges the results of the different phases as the active patterns are used to derive an adequate active structure from the function hierarchy and enable to derive the UML design by refining the active structure and included active patterns by means of component structures and coordination patterns.

References

[AIS+77] ALEXANDER, C.; ISHIKAWA, S.; SILVERSTEIN, M.; JACOBSON, M.; FIKSDAHL-KING, I.; ANGEL, A.: A Pattern Language. Oxford University Press, New York, 1977

[BG03] BURMESTER, S.; GIESE, H.: The Fujaba Real-Time Statechart PlugIn. In Proc. of the Fujaba Days 2003, Kassel, Germany, October 2003

[BGH+04] BURMESTER, S.; GIESE, H.; HIRSCH, M.; SCHILLING, D.: Incremental design and formal verification with UML/RT in the FUJABA real-time tool suite. In Proc. of the International Workshop on Specification and validation of UML models for Real Time and embedded Systems, SVERTS2004, Satellite Event of the 7th International Conference on the Unified Modeling Language, UML2004, October 2004

[BMR+96] BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H.; SOMMERLAD, P.: Pattern-Oriented Software Architecture - A System of Patterns. Wiley, 1996

[BTG04] BURMESTER, S.; TICHY, M.; GIESE, H.: Modeling Reconfigurable Mechatronic Systems with Mechatronic UML. In Proc. of Model Driven Architecture: Foundations and Applications (MDAFA 2004), Linköping, Sweden, June 2004

[Esc93] ESCHERMANN, B.: Funktionaler Entwurf digitaler Schaltungen - Methoden und CAD-Techniken. Springer Verlag, Berlin, 1993

[FGK+04] FRANK, U.; GIESE, H.; KLEIN, F.; OBERSCHELP, O.; SCHMIDT, A.; SCHULZ, B.; VÖCKING, H.; WITTING, K.; GAUSEMEIER, J. (Hrsg.): Selbstoptimierende Systeme des Maschinenbaus - Definitionen und Konzepte. HNI-Verlagsschriftenreihe Band 155, Paderborn, 2004

[FM02] FLAKE, S.; MUELLER, W.: An OCL Extension for Real-Time Constraints. In Object Modeling with the OCL: The Rationale behind the Object Constraint Language, volume 2263 of LNCS, pages 150–171. Springer, February 2002

[GBS+04] GIESE, H.; BURMESTER, S.; SCHÄFER, W.; OBERSCHELP, O.: Modular Design and Verification of Component-Based Mechatronic Systems with Online-Reconfiguration. In Proc. of 12th ACM SIGSOFT Foundations of Software Engineering 2004 (FSE 2004), Newport Beach, USA. ACM, November 2004

[GFR+04] GAUSEMEIER, J.; FRANK, U.; REDENIUS, A.; STEFFEN, D.: Development of Self-Optimizing Systems. In: Proceedings of, Mechatronics & Robotics 2004 (MechRob 2004 (IEEE)). 13.-15. September 2004, Sascha Eysoldt Verlag, Aachen, 2004

[GHJ+96] GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, S, J.: Entwurfsmuster. Addison-Wesley, Bonn, 1996

[GTB+03] GIESE, H.; TICHY, M.; BURMESTER, S.; SCHÄFER, W.; FLAKE, S.: Towards the Compositional Verification of Real-Time UML Designs. In Proc. of the European Software Engineering Conference (ESEC), Helsinki, Finland, pages 38–47. ACM Press, September 2003

[GSS+00] GRUBER, S.; SEMSCH, M.; STROTHJOHANN, F.; BREUER, B.: Elements of an Mechatronic Vehicle Corner. In: 1st IFAC Conference on Mechatronic Systems. Darmstadt, 2000

[IBH02] ISERMANN, R.; BREUER, B.; HARTNAGEL, H.L. (Hrsg.): Mechatronische Systeme für den Maschinenbau - Ergebnisse aus dem Sonderforschungsbereich 241 „Integrierte mechanisch elektronische Systeme für den Maschinenbau (IMES)“. WILEY-VCH Verlag GmbH, Weinheim, 2002

[Ise99] ISERMANN, R.: Mechatronische Systeme - Grundlagen. Springer Verlag, Berlin, 1999

[KFG+03] KIM, D.-K.; FRANCE, R.; GHOSH, S.; SONG, E.: A UML-Based Metamodeling Language to Specify Design Patterns. In: Proceedings of the 2nd Workshop in Software Model Engineering (WiSME, UML 2003 Workshop). San Francisco, USA, 2003

[KFG04] KIM, D.-K.; FRANCE, R.; GHOSH, S.: A UML-based language for specifying domain-specific patterns. In: Journal of Visual Languages and Computing 15 (2004). Nr. 3–4, S. 265–289

[LPY97] LARSEN, K.; PETTERSSON, P.; YI, W.: UPPAAL in a Nutshell. Springer International Journal of Software Tools for Technology, 1(1), 1997

[LRS01] LADDAGA, R.; ROBERTSON, P.; SHROBE, H.: Results of the Second International Workshop on Self-adaptive Software. In: Robertson, P.; Shrobe, H.; Laddaga, R. (Eds.): The Second International Workshop on Self-Adaptive Software (IWSAS 2001), LNCS 1936, 2001. Springer-Verlag, Berlin, Heidelberg, 2001

[MA02] MEYSTEL, M.A.; ALBUS, J.S.: Intelligent Systems - Architecture, Design and Control. Wiley Series on Intelligent Systems, John Wiley & Sons, New York, 2002

[NBP-ol] Homepage of the New Railway Technology Paderborn project (NBP), http://nbp-www.upb.de/en/index.php

[OMG03] OBJECT MANAGEMENT GROUP. UML 2.0 Superstructure Specification, 2003. Document ptc/03-08-02

[PB96] POMBERGER, G.; BLASCHEK, G.: Software-Engineering. Carl Hanser Verlag, 2. Auflage, München, Wien, 1996

[PB03] PAHL G. BEITZ W.: Konstruktionslehre – Methoden und Anwendungen, Springer-Verlag, 5. Auflage, Berlin, 2003

[Pre94] PRESSMANN, R.S.: Software Engineering - a practioneer´s approach. McGraw-Hill, 3. Auflage, 1994

[Rot00] ROTH, K.-H.: Konstruieren mit Konstruktionskatalogen. Band 1 Konstruktionslehre, Springer Verlag, 3. Auflage, Berlin, 2000

[SFB-ol] Homepage of the Collaborative Research Center 614 „ Self-Optimizing Concepts and Structures in Mechanical Engineering", http://www.sfb614.de/eng/index_e.htm

[SFO03] STEINBERG, R.; FJELLHEIM, R.; OLSEN, S.A.: Design pattern for safety-critical knowledgebased systems. In: Vermesan, A.; Coenen, F. (Hrsg.): Validation and Verification of Knowledge Based Systems - Theory, Tools, Practice. Kluwer Academic Publishers, Boston, 2003, S. 131–147

[SGW94] SELIC, B.; GULLEKSON, G.; WARD, P.: Real-Time Object-Oriented Modeling. John Wiley & Sons, Inc., 1994

[SH04] SOUNDARAJAN, N.; HALLSTROM, J.O.: Responsibilities and Rewards: Specifying Design Patterns. In: Proceedings of the 26th International Conference on Software Engineering. Edinburgh, Scotland, IEEE Computer Society, 2004, S. 666–675

[SR98] SELIC, B.; RUMBAUGH, J.: Using UML for Modeling Complex Real-Time Systems. Techreport, ObjectTime Limited, 1998

[SSO+04] SCHEIDELER, P.; OBERSCHELP, O.; SCHMIDT, A.; HESTERMEYER, T.: Distributed Optimization of Reference Trajectories for Active Suspension with Multi-Agent Systems. In: Proceedings of 18th European Simulation Multi-Conference (ESM 2004). 13.-16.Juni 2004, Magdeburg, Deutschland

[VDI04] The Association of Engineers (VDI): Design methodology for mechatronic systems. VDI 2206, VDI Guidelines, Beuth Verlag, 2004

ACKNOWLEDGEMENT

This contribution was developed in the course of the Collaborative Research Center "Self-Optimizing Concepts and Structures in Mechanical Engineering" (Speaker: Prof. Gausemeier) funded by the German Research Foundation (DFG) under grant number SFB614.