CREATE: A Co-Modeling Approach for Scenario-based Requirements and Component-based Architectures - A Detailed View

Björn Schindler, Marcel Ibe, Martin Vogel, and Andreas Rausch
Technische Universität Clausthal
Clausthal-Zellerfeld, Germany
{bjoern.schindler, marcel.ibe, m.vogel, andreas.rausch}@tu-clausthal.de

***Abstract*—Requirements engineering and architectural design are key activities for successful development of software-intensive systems and are strongly interrelated. Particularly, in early development stages requirements and architecture decisions are frequently changing. The fundamental problem addressed in this paper is the development of inconsistencies at the iterative evolution of requirements and architectures. Inconsistencies between requirements and architectures lead to an incorrect consideration of requirements by the system under development and consequently to unfulfilled requirements. Thus, advanced systematic approaches are needed, which could minimize the risks of wrong early decisions during the iterative evolution of requirements and architectures. Our model-based approach supports the iterative evolution of requirements and architectures by defining a concrete description technique. It provides simplified scenario-based models for a precise description of requirements, which are suitable for validation by stakeholders. Furthermore, the approach provides a component-based model for a precise and entire description of architectures. Strict interrelations between scenario-based and component-based models support the consistence maintenance. These interrelations enable even an automation of this task. In this paper, the model-based approach CREATE is described in all details. For example, all interrelations are introduced completely.**

Keywords-requirements; architecture; evolution; consistency.

I. INTRODUCTION

Requirements Engineering (RE) and Architectural Design (AD) are essential for successfully developing high-quality software-intensive systems [1]. RE and AD activities are intertwined and iteratively performed [2]. The architecture of a software system must satisfy its requirements. In practice, architectural constraints frequently prohibit an entire realization of all requirements. This might imply a change to the initial requirements or the selection of a different appropriate architecture. Further, additional requirements might be discovered during the development process, leading to changes in the architecture. Design decisions that are made early in this iterative process are the most crucial ones, because they are very hard and costly to change later in the development process.

In classical development processes (e.g., the waterfall model [3]), artifacts like, for instance, the requirements specification or the architecture are developed sequentially. This is also the case at iterative process models like the spiral life cycle model of Böhm [4]. The iterative, concurrent evolution of requirements and architectures demands that the development

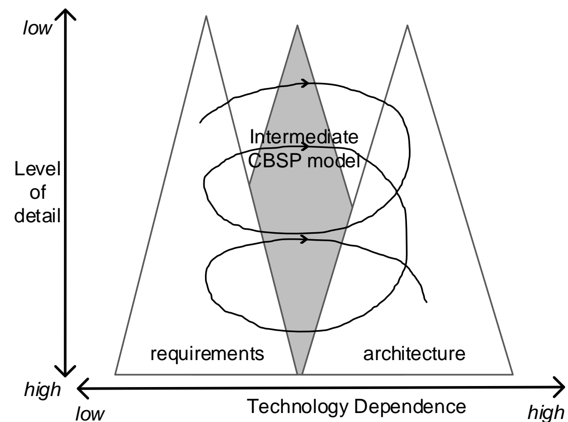

Figure 1. Intermediate model within the twin peaks [5]

of an architecture is based on incomplete requirements. Also, certain requirements can only be understood after modeling or even partially implementing the system architecture. Nuseibeh [2] describes an advanced approach, which adapts the spiral life cycle model and aims at overcoming the often artificial separation of requirements specification and design by intertwining these activities in an iterative evolutionary software development process. This approach is called the twin peaks model. To map requirements onto architectures and maintain the consistency and traceability between the two Grünbacher et al. [5] introduces an intermediate model called Component Bus System Property (CBSP) (see Fig. 1). This model maps requirements to architecture elements by the CBSP model, which allows a systematic way to reconcile requirements with stakeholders.

Nevertheless, the advanced twin peaks model is kept very general. For instance, it does not specify the level of detail of requirements in relation to the architecture [6]. Due to the fact that there is no concrete advanced approach supporting the iterative evolution of requirements and architecture we were commissioned by the german armed forces and the german government to undertake a research project [1]. In order to be able to consider all required aspects, we made an expert survey. Therefore, we interviewed staff and leaders of three medium to big sized development projects with up to 30 project participants on customers and contractors side about their problems in the field of RE and AD.

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

A general mentioned problem was that the developed systems did not fulfill all requirements of the customers. The result of the survey was a list of the following reasons and derived guidelines:

- For the contractor the requirements were too informal, imprecise and incomplete. Requirements had to be repeatedly elicited and specified during architecture design. Hence, requirements on a software system should be complete and precise.
- At the elicitation process, the reconcilement of more precise and formal descriptions was too costly. The reason was the need of a detailed explanation by the contractor. For an improved reconcilement requirements descriptions should be precise as well as comprehensible. These guidelines are also mentioned by Nuseibeh [7]. Furthermore, the complexity of the models have to be manageable for validation by stakeholders.
- The most serious problem was caused by frequently changing requirements during architecture design. These changes cause inconsistencies between requirements and architectures. Thus, many requirements were not fulfilled by the developed systems. In consequence, an architecture should not only describe the system by a definition of its structure and behavior. It should also describe precisely how the system under development fulfills the given requirements. Consistency constraints between requirements and architectures should be defined, which support the consistency maintenance.

Starting from this initial situation the target of the project was the development of a domain specific model-based approach, which fulfills the mentioned guidelines. The result of the project was the model-based approach CREATE [1]. The main goal of CREATE is to provide a concrete description technique for requirements and architectures, which supports the iterative evolution in the sense of twin peaks. Requirements descriptions have to be precise and comprehensible. This necessitates a well-balanced trade-off between expressiveness and manageability of models for the description of requirements. This trade-off is achieved by providing simplified scenario-based requirements models. Furthermore, CREATE provides a component-based architecture model for a precise desciption of how the system under development fulfills the given requirements. A definition of strict interrelations support the iterative evolution of requirements and architectures. Complex interrelations between requirements and architectures cause a high complexity for consistence maintenance. Thus, CREATE proposes interrelations between requirements and architecture models, which concretize a well-balanced trade-off between expressiveness and manageability.

In this paper, the CREATE approach is described in all details. Furthermore, the experiences at the application in practice are described. The presented approach is domain-specific, since it is developed for information systems like web-based systems and modern communication systems. In Section II, existing model-based approaches for the iterative evolution of requirements and architectures are considered. In Section III, the overall approach is introduced and in Section IV the description technique is described in detail at an example. The support of the consistency maintenance is described in Section V. Section VI contains a description of our experiences at the development and application of the approach in practice. Section VII includes a discussion of the results and pending points for future work.

II. Related Work

Existing model-based development approaches for requirements and architectures can be categorized into model-based approaches for requirements engineering, model-based approaches for architecture design and combined approaches.

Representative model-based approaches for requirements engineering are described in [8]–[10]. In [8], requirements are described by Unified Modeling Language (UML) [11] activity diagrams. A formal operational semantics enables execution of activity diagram specifications. The executed activity diagram specification serves as prototype for visualization of requirements. In the approach illustrated in [9], UML collaboration diagrams are enriched by user interface information in order to specify elicited requirements. These diagrams are transformed into complete dynamic specifications of user interface objects represented by state diagrams. These state diagrams are used for generation of prototypes. In [10], use case and user interface information are recorded at stakeholder interviews. Therefore, use case steps are enriched by scribbled dialog mockups. Prototypes are created, which visualize dialog mockups of use case steps in sequence for fast feedback of stakeholders. In general, these approaches have a well elaborated model structure for requirements engineering and improve the validation of requirements by stakeholders. On the other side, the mapping to the architecture is not precisely enough defined at these approaches to support a iterative evolution of requirements and architectures.

Representative approaches defining models for architectural design are described in [12] and [13]. Model-Driven Architecture (MDA) [12] is a framework for software development. In MDA, the Computation Independent Model (CIM) can be used to describe business processes. The Platform Independent Model (PIM) may describe the structure and behavior of the software system. Component models like KobrA [13] are concrete model-based approaches based on MDA. In general, these approaches have a well elaborated model structure for architecture design and enable a detailed description of the structure and behavior of the software system. On the other side, these approaches do not support an iterative evolution of requirements and architectures. The mapping between requirements and architectures is not precisely enough defined for this field of application.

Representative combined modeling approaches for requirements and architectures are described in [14], [5], and [15]. In [14], a Requirements Definition Language (RDL) is used, which allows a structured definition of requirements. Meta model elements of the RDL are mapped to corresponding meta model elements of the Architecture Description Language (ADL). The approach described in [5] uses the intermediate model CBSP to map requirements to architecture elements. Different subtypes of CBSP elements allow classification of

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

requirements. Requirements exhibit overlapping CBSP properties can be split and refined until no stakeholder conflicts exist. The Software Architecture Analysis Method (SAAM) [15] describes a method for a scenario-based analysis of software architectures. SAAM defines also the activities of the scenario-based analysis. In SAAM, scenarios and architecture descriptions are developed iteratively [15]. For each scenario it is determined whether a change of the architecture is required for execution. Based on the importance and conflicts of required changes an overall ranking of the developed scenarios is determined. An advantage of these approaches is the combination of techniques for the description of requirements and architectures. On the other side, these approaches are very abstract and do not specify concrete models and mappings, which fulfill the conditions defined in the introduction for an adequate description of requirements and architectures.

Besides the stated existing approaches further approaches are conceivable, which are based on synthesis approaches [16] of complete state-based models from scenario-based models. Scenario-based and state-based models can potentially be used for the description of requirements and architectures. Consistency is, for instance, a subject of the approaches described in [17] and [18]. Unfortunately, these approaches are generally maintaining a complete consistency by means of a bijection. Architectures need to describe more details about the software system. These details have to be well separated from the requirements. Hence, an alternating correction of inconsistencies and not a bijection is required for the support of a iterative evolution of requirements and architectures.

Other approaches are focusing on the consistence maintenance of models. Representative approaches of this kind are described in [21], [22]. In [22] an approach for the automatic consistence check of behavioral requirements and design models is described. The approach introduced in [21] allows an automatic consistence check of general UML models. These approaches focus on the consistence maintenance and not on the provision of a concrete description technique for requirements and architectures. In consequence, these approaches do not provide a complete set of concrete models and mappings, which fulfill the conditions defined in the introduction for an adequate description of requirements and architectures.

The main goal of CREATE is to provide a concrete description technique for requirements and architectures, which supports the iterative evolution by fulfilling all guidelines mentioned in the introduction. It defines a complete set of concrete models for the description of requirements and architectures. Further, it defines concrete interrelations between these models, which support the consistence maintenance.

III. Overall Approach

Our domain specific model-based approach supports concurrent development of requirements and architectures. An appropriate process for concurrent development is described by the twin peaks model [2]. In this model, requirements and architectures have an equal status and are evolved iteratively. This is illustrated by twin peaks (see Fig. 2).

Our domain specific model-based approach concretizes twin peaks by defining a concrete description technique. Diagrams

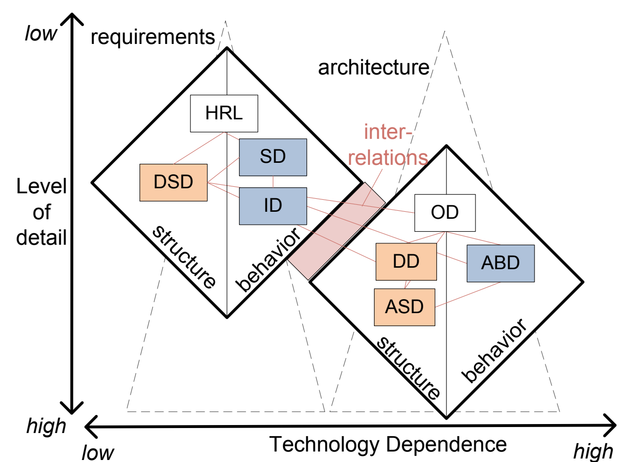

Figure 2. CREATE approach within twin peaks

are used for a precise description of requirements and architectures. These diagrams are illustrated within diamonds in the twin peaks model (see Fig. 2). The process flow of our approach begins with a formal description of inital requirements. Afterwards, the architecture is developed and consistency to the requirements is maintained continuously. Inconsistencies are resolved by changing requirements or the architecture.

The main contribution of CREATE is the concrete description technique with defined interrelations between requirements and architecture descriptions. It is well known that scenarios help to elicit and validate requirements [16]. A precise description of elicited requirements can be achieved by scenario-based models [16]. The co-modeling approach provides simplified scenario-based models for the description of requirements. Furthermore, the description is reduced to representative and concrete scenarios. Hence, the complexity of these models is manageable for the validation by stakeholders. The validation is improved by combining these models with models enabling visualization of requirements by user interface mockups [10]. An architecture specified by CREATE describes the behavior and the resulting structure of the software system precisely. This description is supported by a component-model. Component-Based Software Engineering (CBSE) [19] has been continuously improved and successfully applied over the past years. Systems are composed by existing software 'parts' called software components. Component models enable a precise description of component-based architectures [20].

In our domain specific model-based approach, diagrams are used to model structural or behavioral aspects of requirements and architectures. The domain structure (e.g., business structure) defines important requirements on the system. In CREATE, it can be described by a Domain Structure Diagram (DSD), which is assigned to the structure part of the requirements diamond (see Fig. 2). Elicitation and specification of processes at the domain (e.g., business processes) is an important aspect at requirements engineering. In our approach, these processes can be described by a Scenario Diagram (SD)

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

in combination with an Interaction Mockup Diagram (ID). The SD and ID are assigned to the behavior part of the requirements diamond. Additional text-based requirements can be captured in the Hierarchical Requirements List (HRL). These texts can describe structural as well as behavioral aspects. CREATE diagrams have not to be developed in a strict order. Typically, rough text-based requirements are captured first. The DSD as well as the SD and ID are, in general, evolved iterativly beginning with a rough first scenario. At the architecture design, the definition of the system boundary and the provided functions is crucial. These aspects of the architecture can be described precisely by the System Overview Diagram (OD) of the CREATE architecture (see Fig. 2). The process of a function is described by the Architectural Behavior Diagram (ABD). Data objects used in these processes are defined in the Data Diagram (DD). A suitable structure of the system can be derived from the process descriptions. This structure is described by the Architectural Structure Diagram (ASD). The diagrams of the CREATE architecture are iteratively developed as well. Typically, a small set of system functions are described by the OD, ABD and DD first. Afterwards, the ASD is derived, and finally, additional system functions are described. During the development of the architecture, consistency to the requirements is maintained continuously. In Section IV, the description technique is described in all details.

Existing languages, such as UML [11], include among others structural and behavioral diagrams for the modeling of systems. In this paper, CREATE uses exemplarily a subset of UML diagrams and their available model elements to formally describe requirements and architectures. Additional diagrams are used to enable a visualization of requirements by user interface mockups. Interrelations between these diagrams are precisely defined. Consistency maintenance during the development of requirements and architectures is supported by defined interrelations between scenario-based requirements models and component-based architecture models (see Fig. 2). Interrelations are also defined within these models. One interrelation between requirements and architectures is, for instance, that the system boundary of the OD represents a part of the domain structure in the DSD. An examplary interrelation within the requirements description is that every object used in a scenario has to be a part in the domain structure. In Section V, all interrelations are described in detail. Interrelations are defined by constraints. The definition of these constraints support the consistency maintenance, because they can easily be checked. Furthermore, they enable an automatization of the consistency maintenance [24].

IV. Description Technique

In this section, details of the description technique are described by a case study. The subject of this study is the development of a library system. Requirements on the library system are described by the scenario-based model of CREATE, which is suitable for the validation by stakeholders. The architecture of the library system is described by the CREATE component model. This component model allows the description of the behavior of the system under development and the resulting internal structure. In the following, initial requirements on the library system are described by the provided diagrams. Afterwards, the architecture of the system is described. Based on the requirements and the architecture of the library system the description technique of CREATE is explained in detail.

A. Requirements Description

In CREATE, requirements on a system are precisely described by a scenario-based model. The scenario-based model of CREATE consists of the provided SD, ID and DSD, which are the core diagrams of the requirements description. The domain structure like the business structure defines important requirements on the system. In CREATE, it can be described by the DSD. Processes on this domain like business processes are described by the SD in combination with ID. Since requirements are frequently mentioned text-based (e.g., in protocols or meetings) the HRL is provided. The HRL allows the capturing of text-based requirements. A strict order for the development of the CREATE diagrams is not prescribed. In practice, rough text-based requirements are elicited initially. Afterwards, the scenario-based model is developed. A rough first scenario is refined by an iterative development of the DSD, the SD and the ID. In the following, the HRL is explained by exemplary text-based requirements on the library system. Afterwards, the scenario-based model is explained in all details. For this detailed explanation the DSD, the SD and the ID are used for the definition of precise requirements on the library system.

1) HRL: Requirements are frequently mentioned text-based (e.g., in protocols or meetings). The HRL allows the capturing of text-based requirements. In a requriements specification, several HRLs can be introduced in ordner to distinguish between different classes of requirements. In our case study, we introduce two HRLs for the capturing of functional and non-functional requirements (see Fig. 3). Contents of the HRL can be structured hierarchical. In this way, it is possible to refine one requirement by several other requirements. In our example, the HRL *functional requirements* contains, for instance, the requirement *2*. This requirement demands that the system must be able to process requests for book orders of users. This is refined by requirement *2.1*, which demands that a user should be able to send a book order request to the manager by the library system.

2) DSD: In general, the system under development has to be integrated in a domain structure (e.g., a business structure). The domain structure defines important requirements on the system. The DSD allows a precise description of the domain structure and is based on the UML Composition Structure Diagram [11]. The most important modelling elements of the DSD are parts and connectors [11]. Parts are used to describe the system under development, external systems, persons and entities of the domain. Every part must have a type. In return, one type can be used for several parts. In our example, the parts *LibrarySystem* and *Printer* represent systems (see Fig. 4). The multiplicity defines the minimal and maximal number of objects in this part. According to the DSD, the library domain contains, for instance, exactly one library system. The box that represents the part *LibrarySystem* is filled gray to mark it as

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

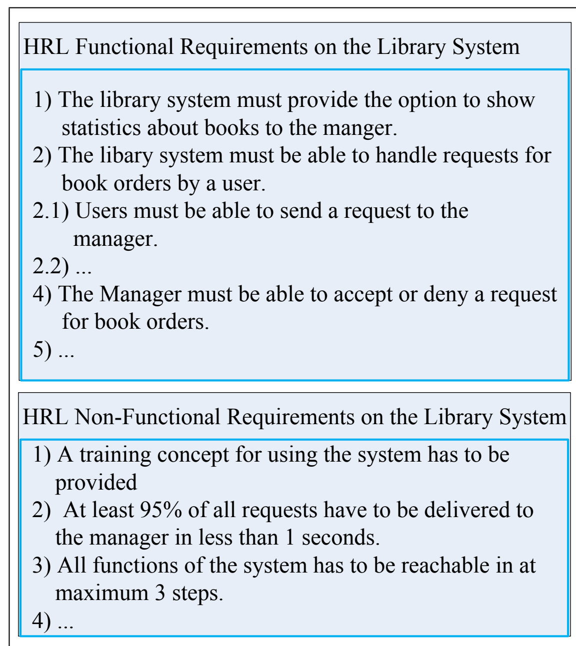

Figure 3. Hierarchical Requirements List of the Library System

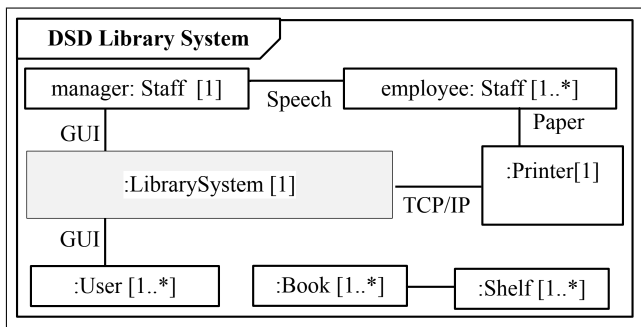

Figure 4. Domain Structure Diagram of the Library System

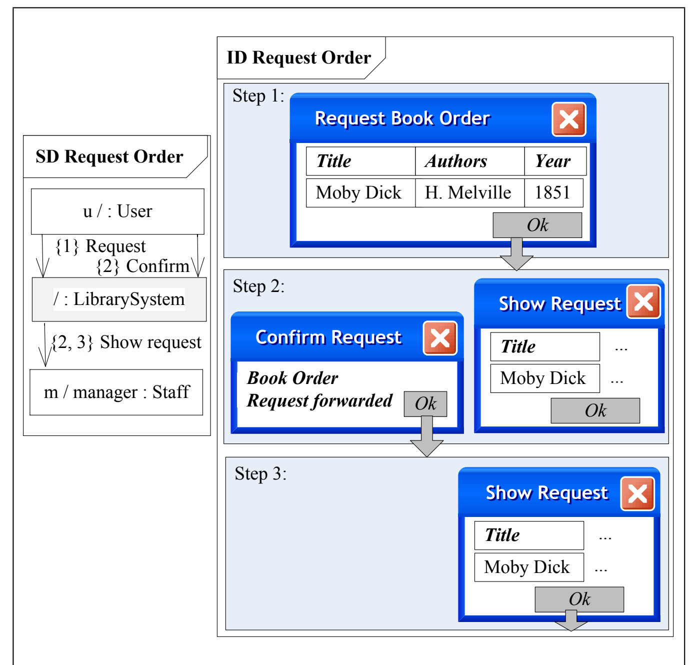

Figure 5. Scenario - Request Book Order

the system under development. Three parts in the domain are representing persons: *User*, *Manager* and *Employee*. *Manager* and *Employee* are both of the type *Staff*. The parts *Book* and *Shelf* describe typical domain entities. Connectors describe the ability of two connected parts to interact with each other. A connection between two parts means that every object of one part can interact with all objects of the other part. It is also possible, to define the type of the connector. This type describes the kind of the interaction in more detail. The domain structure of the library system describes several connectors between parts, which specify the kind of interaction. For example, users can interact with the library system by the graphical user interface (GUI). The printer interacts via TCP/IP with the library system. Interaction between external systems and persons can also be modelled, e.g., the manager and an employee can communicate via speech.

3) SD and ID: Elicitation and specification of processes at the domain (e.g., business processes) is an important aspect at requirements engineering. In our approach, these processes can be described by a SD in combination with an ID. SD is based on scenario-based UML Communication Diagrams [11] and describes representative scenarios at the domain, which have to be supported by the system under development. The most important elements of an SD are lifelines and interactions. Lifelines represent instances of systems, persons and entities of the domain structure. The interactions are described by messages of the UML communication diagram and represent the interaction between the instances during a specific scenario. They are visualized by an arrow between two lifelines. The SD *Request Order* and SD *Process Request* describe scenarios in the library domain (see Fig. 5 and Fig. 6).

The SD *Request Order* describes the interactions between a user, the library system and the manager in the case of a request for a book order. An ID visualizes and describes one SD in more detail. The ID describes a set of scenario steps. The scenario *Request Order* has, for instance, the scenario steps *1*, *2* and *3*. Every interaction of an SD can be active at a set of scenario steps. The interaction *request* of the SD *Request Order* of the user with the system is, for instance, active at scenario step *1* and is labeled with this number. An active interaction is described by an interaction mockup in the corresponding ID, which is visualized by a dialog. The interaction *request* is, for instance, visualized by the interaction mockup *Request Book Order* in scenario step *1*. The visualization of the ID is suitable for the validation of scenarios by stakeholders, since the used interaction mockups give a representative view on the exchanged data. In a transition to a next scenario step, interactions can be activated and deactivated. The begin of a scenario, the end of a scenario, and a completion of an interaction can be the trigger of a transition. The completion of the interaction *Request* visualized by the

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

interaction mockup *Request Book Order* is, for instance, the trigger of the transition to scenario step *2*.

Several interactions can be active concurrently in one scenario step [25]. In this case, one scenario step of the ID contains two interaction mockups. The scenario step *2* contains, for instance, the interaction mockups *Confirm Request* and *Show Request*. These mockups are visualizing the interactions *Confirm* and *Show Request* of the system to the user resp. the manager. Both interactions are labeled with the number *2* of the corresponding scenario step. In the case of concurrency, an interaction might be active at several scenario steps. Consequently, several sequenced scenario steps can contain the same interaction mockup [25]. The interaction *Show Request* is active in the scenario steps *2* and *3*. Hence, the visualizing interaction mockup is contained in these steps. The interaction is labeled with the numbers *2* and *3*.

An SD and ID can also be used to describe alternative scenario sequences [25]. The SD and ID *Process Request* describes, for instance, a scenario with two alternative scenario sequences for the processing of a request for a book order (see Fig. 6). In the first alternative, the book is not ordered. In the second, alternative the book is ordered by forwarding the request to the employee and printing the book data. These alternatives are described by the scenario steps *3a* and *3b*, which can follow scenario step *2*. If the book is not ordered, the manager returns to the overview of all books after showing the books statistic and exits the overview. The interaction of exiting the overview is, for instance, described by the message *Exit* in the SD. Since this interaction is active in scenario step *3a*, it is labeled by this number. If the book is ordered, the request is forwarded to an employee and the book data is printed in parallel. The printing of the book data is, for instance, described by the interaction *Print Book*, which is active in the scenario steps *3b* and *4*.

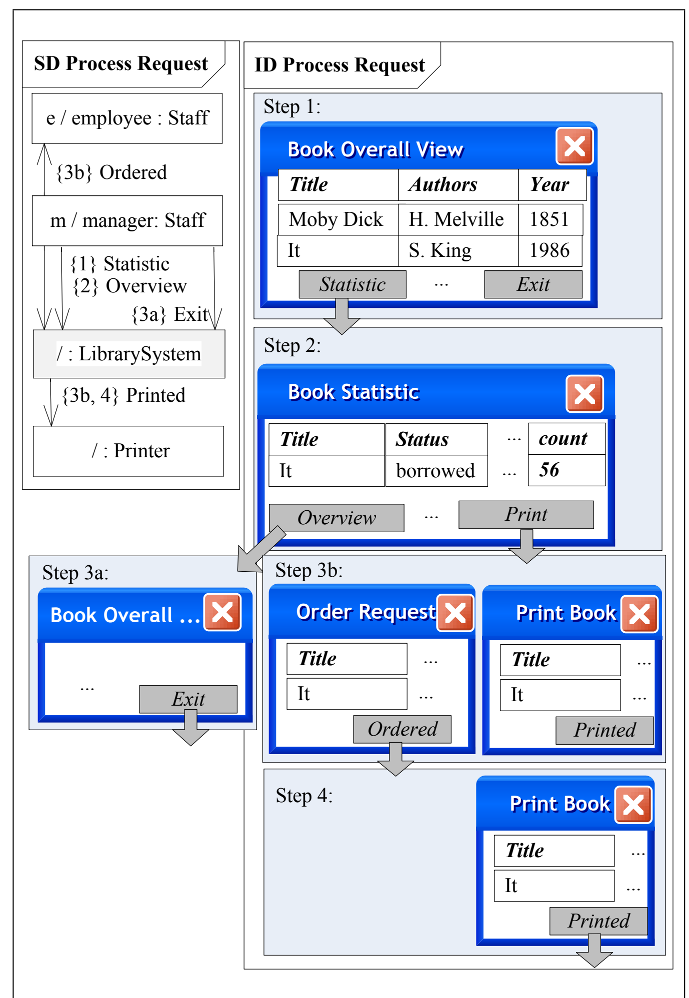

Figure 6. Scenario - Process Request

B. Architecture Description

The architecture of the library system is described by the CREATE component model. This component model allows the description of the behavior of the system under development and the resulting internal structure. The component model consists of the provided OD, ABD, DD and ASD. At the architecture design, the definition of the system boundary and the provided functions is crucial. The system boundary and the provided functions can be described by the OD of CREATE. The process of a function is described by the ABD. Data objects used in these processes are defined in the DD. A suitable structure of the system can be derived from the process descriptions. This structure is described by the ASD. The diagrams of the CREATE architecture have not to be developed in a fixed order. Typically, a small set of system functions are described by the OD, ABD and DD first. Afterwards, the ASD is derived, and finally, additional system functions are described. During the development of the architecture, consistency to the requirements is maintained continuously. In the following, all diagrams of the CREATE architecture are explained in detail by an exemplary architecture design of the library system.

1) OD: A crucial step in the architecture design is the definition of the system boundary and the privided functions of the system under development. The system boundary and the provided functions can be described precisely by the OD, which is based on the UML Use Case Diagram [11]. It describes the most abstract structure and behavior of the system and its context by the system boundary and the associated use cases, which are called *functions*. The OD of the library system describes the system with the provided functions *ShowBooksStatistic* and *PrintBookStatistic* (see Fig. 7).

The OD describes additionally all actors, which are directly involved in functions of the system under development. An actor can be a person, an external system or a hardware device. The OD of the library system describes, for instance, the actors *User*, *Staff* and *Printer*. These actors are involved in at least one function of the library system. The actor Staff is, for instance, using the function *ShowBooksStatic* and *PrintBooksStatistic*. The printer is used by the system at the function *PrintBooksStatistic*. The process for each function is described by an Architectural Behavior Diagram.

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

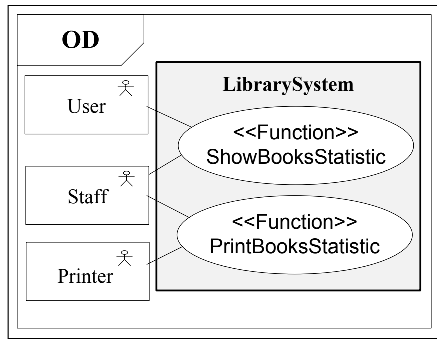

Figure 7. OD of the library system

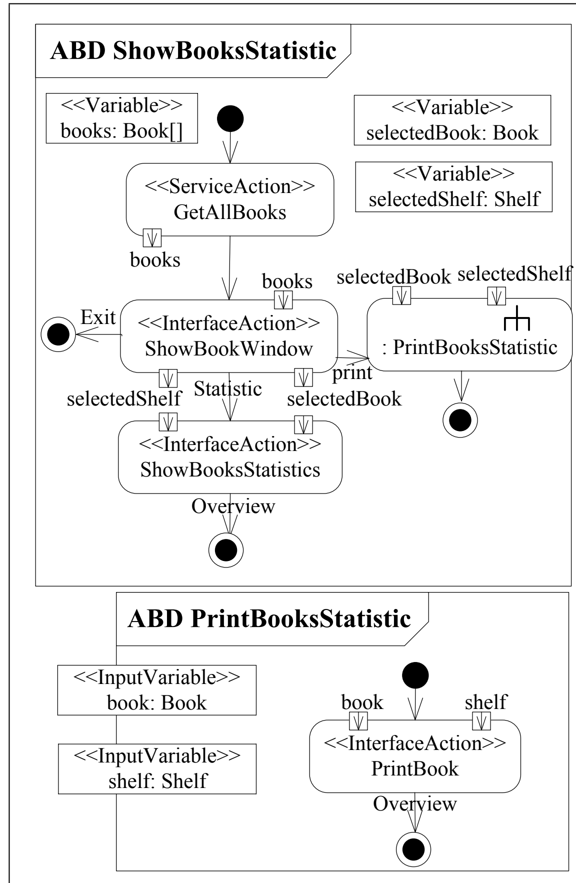

Figure 8. ABDs of the functions ShowBookStatistic and PrintBookStatistic

2) ABD: The ABD defines the behavior of the software system by describing the processes of the functions defined in OD completely. It is based on the UML Activity Diagram [11] including data flow. An activity of an ABD describes a process. In this process actions can be perfomed. The process can be described by control nodes. Control nodes allow, among others, a description of parallel and alternative execution sequences [11]. The function *ShowBooksStatistic* is, for instance, described by the ABD *ShowBookStatistic* (see Fig. 8).

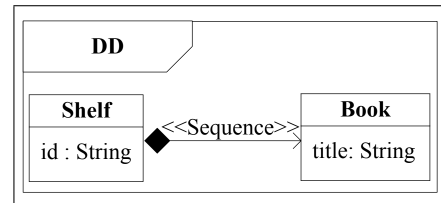

Figure 9. DD of the library system

Within the ABD, different action types like *InterfaceAction* and *ServiceAction* are used. An *InterfaceAction* describes an interaction of the system with its environment. The action *ShowBookWindow* describes, for instance, an interaction with the user by showing a dialog (see Fig. 8). A *ServiceAction* is performed by the system (e.g., a database call). The action *GetAllBooks* reads all books from the data base. An ABD supports the usage of call behavior actions [11], which describe the execution of a called activity within the execution of another activity. The action *:PrintBooksStatistic* describes the call of the ABD *PrintBooksStatistic*.

The ABD uses variables for the description of the data flow of processes [25]. In the description of the function *ShowBook-Statistic*, the variables *books*, *selectedBook* and *selectedShelf* are used. Variables can have different data types and can hold a set of objects. The variable *books* holds, for instance, a set of objects of the type *Book* (see Fig. 8). Reading and writing of variables by actions can be described by input pins and output pins. The action *GetAllBooks* writes, for instance, all selected book objects into the variable *books*. This variable is read by the action *ShowBookWindow*, which shows, among others, an overview about the selected books.

At a call of another ABD parameters can be passed. For the passing of parameters a copy semantics is used. The action *:PrintBooksStatistic* of the ABD *ShowBooksStatistic* describes, for instance, a call of the ABD *PrintBooksStatistic*. The ABD *PrintBooksStatistic* has the input variables *book* of the type *Book* and *shelf* of the Type *Shelf* (see Fig. 8). The action *:PrintBooksStatistic* has two input pins, which are referring to the variables *selectedBook* and *selectedShelf*. At a call, the objects in the referred variables are copied to the input variables of the called ABD. At a return, the objects of the output variables are copied to the referred variables of the output pins of the call behavior action.

3) DD: At a function, described by ABD, data objects can be used by the system. The DD is based on UML Class Diagrams [11] and describes the data types of each data object. For example, the DD of the library system describes a type *Book* and *Shelf*, which are types of the ABD *ShowBooksStatistic* variables (see Fig. 9).

Data types described in the DD can have attributes similar to classes in the UML class diagramm. Due to the copy semantics of the ABD and the complexity of copying object networks, attributes can only have a primitive type in the DD. Relations to other data types are only described by sequences and generalisations. A sequence is a special composite aggregation [11], which allows no cylces and no membership of one object to more than one other object. In the DD of the library system

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

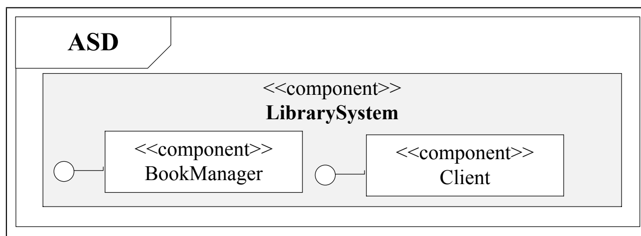

Figure 10. ASD of the library system

the data type *Shelf* has a sequence of objects of the type Book. The meaning of a generalization of data types is equal to the generalization of classes in the UML class diagram [11].

4) ASD: The ASD is based on UML Component Diagrams [11] and describes the internal components of the system under development and their offered interface as a black-box view. Subsequently, the components are further decomposed to refine their internal structure. The ASD *LibrarySystem* describes, for instance, the internal structure of *LibrarySystem* of the OD (see Fig. 10). The internal structure is derived from the actions of the ABD. Hence, each component must be associated with at least one action of an ABD. The component *LibrarySystem* is, for example, refined by a component *BookManager*, which is associated with the action *GetAllBooks* of the ABD.

V. Consistency constraints

In CREATE, consistency between requirements and architectures is maintained continuously during the architecture design. Consistency maintenance is supported by defined interrelations between scenario-based requirements models and component-based architecture models. Interrelations are defined by constraints. The definition of these constraints support the consistency maintenance, because they can easily be checked. Furthermore, they enable an automation of the consistency maintenance [24]. In this section, these interrelations are explained in detail based on the library system example described in Section IV. A summary of these interrelations is given in Fig 11. We distinguish three kinds of interrelations:

1) Interrelations within requirements (see gray dotted lines between the requirements diagrams in Fig. 11, e.g., between HRL and SD).
2) Interrelations within architecture (see gray dotted lines between the architecture diagrams in Fig. 11, e.g., between OD and ASD).
3) Interrelations between requirements and architecture (see red dotted lines between diagrams of the requirements and diagrams of the architecture in Fig. 11, e.g., between the DSD and the OD).

One interrelation between requirements and architectures is, for instance, that the system boundary of the OD represents a part of the domain structure in the DSD. In the following, the interrelations within requirements, the interrelations within architectures and the interrelations between requirements and architectures are explained completely and in detail. The interrelations are explained by the example of the requirements model and architecture model of the library system.

A. Constraints within requirements

CREATE defines strict interrelations between the requirements models. These interrelations are defined by consistency constraints. An inconsistency means a violation of these constraints. Within requirements the following consistency constraints are defined:

1) Every lifeline in a SD must have a corresponding part with the same type in the DSD.
2) If an interaction in a SD takes place between two lifelines, a connector has to exist between the corresponding parts in the DSD.
3) In a scenario described by an SD the number of instances have to comply with the multiplicity of the corresponding parts in the DSD.
4) Every interaction of the SD is described by exactly one interaction mockup of the ID and every interaction mockup describes exactly one interaction.
5) Every text-based requirement on the system of the HRL is described by at least one SD.
6) Marked subjects in the HRL are described in the DSD.
7) Every marked subject in the HRL must be used in the SD, which describes this text-based requirement.

The requirements model of the library system example comply with all of these constraints. For instance, the lifeline *manager* of the SD *Process Request* has a corresponding part with the same type *Staff* in the DSD (see Fig. 11). A connection exists between the part *manager* and the part of the library system. In this way, the interaction *Statistic* between the manager and the library system complies to the constraint *2*. Further, exactly one library system is used in the SD *ProcessRequest*. This complies with the constraint *3*. The constraint *3* is fulfilled for every scenario of the library system described by an SD.

In the following, typical changes of requirements are introduced in order to show the inconsistency detection and solving of the CREATE approach. A typical change during the development of a software system is the addition of a new scenario. In a new scenario of the library system example, the employee has a look at the book statistics. This scenario is described by a new ID *ShowBookStatistic* and a new SD *ShowBookStatistic* (see Fig. 12). Furthermore, the text-based requirement *1* in the HRL is changed, which states that employees and managers need to have a look at book statistics. During the development of this scenario, it is discovered that employees should not be able to see all book information like the manager. Hence, the lifeline of the employee in the SD is not of the type *Staff*, but of the type *Employee* (see Fig. 12). Employee is not defined in the initial DSD (see Fig. 4). In consequence, the consistency constraint *1* is violated. This inconsistency is fixed by introducing the types *Employee* and *Manager* in the DSD and assigning these types to the parts *employee* resp. *manager*. This leads to another inconsistency with the SD *Process Request*. The lifeline *manager* and *employee* are of the type *Staff* in the initial version of the SD. But this type is not defined in the DSD anymore. This inconsistency is fixed by changing the type of the part *employee* from *Staff* to *Employee* and the type of the *manager* from *Staff* to *Manager*. Furthermore,

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

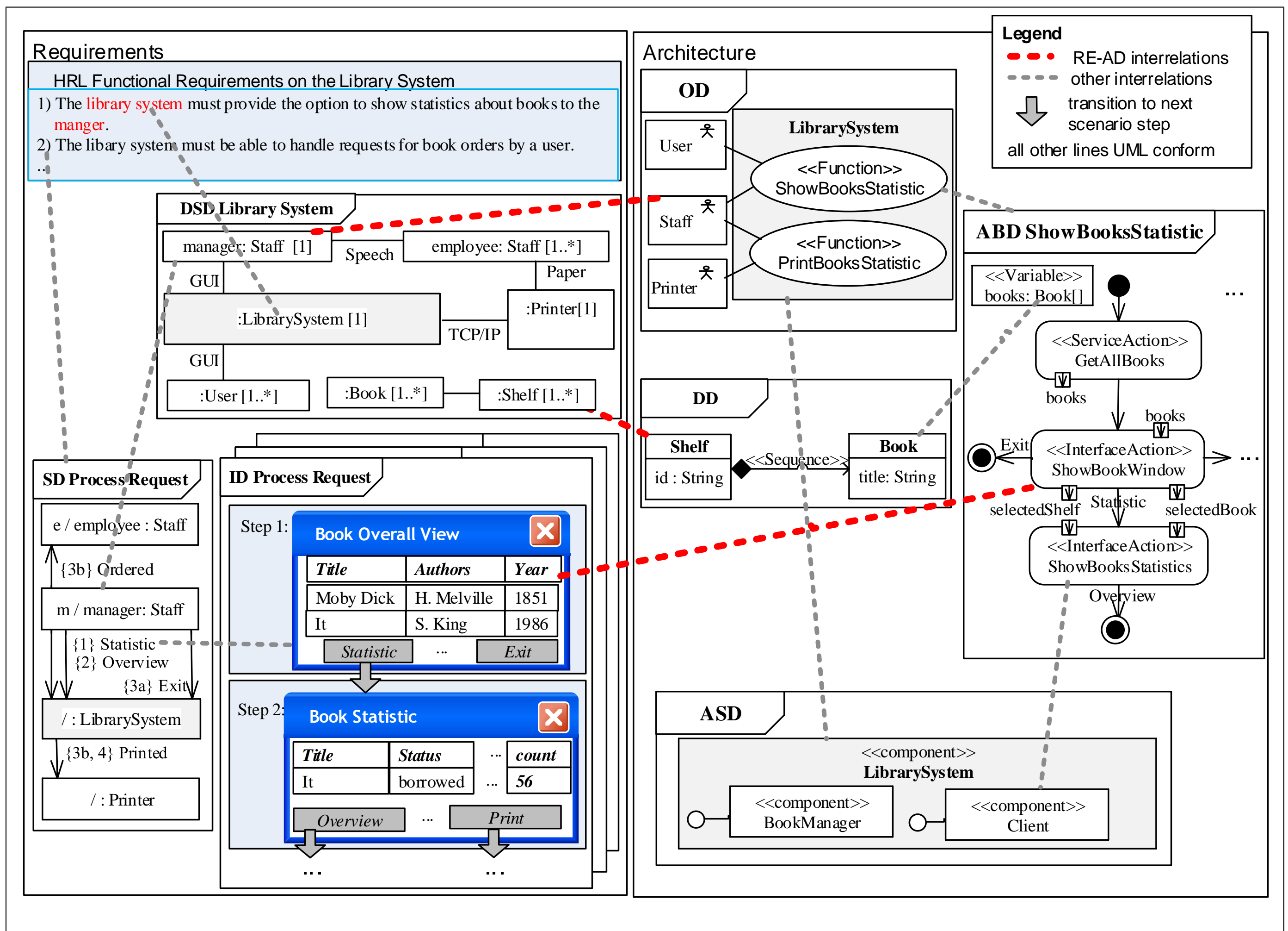

Figure 11. Requirements models, architecture models and interrelations

the consistency constraint *2* is initially violated by introducing the SD *ShowBookStatistic*. The employee interacts with the library system, but a connector is not existing between the corresponding parts of the DSD. To solve this inconsistency a new connector has to be added between the parts for the employees and the library system.

B. Constraints within architectures

In CREATE, strict interrelations between architectures diagrams are defined. These interrelations are also defined by consistency constraints. A violation of a consistency constraint points out an inconsistency. Within architectures the following consistency constraints are defined:

1) Every system function has to be described by one ABD.
2) Every type of a variable in an ABD has to be primitive or must be defined in the DD.
3) Every input pin and output pin of a call behavior action must have a corresponding input variable resp. output variable of the called ABD and vice versa.
4) The types of the pins of a call behavior action must match the types of the corresponding variables of the called ABD.
5) The system boundary of the OD has to be decomposed by the ASD.
6) Every action described in one ABD has to be realized by exactly one component of the ASD.
7) Every component of the ASD has to realize at least one action of one ABD.
8) An actor has to be involved in a system function.

The architecture model of the library system example comply with all of these constraints. For instance, the function *ShowBookStatistic* of the OD is described by the ABD *ShowBookStatistic* (see Fig. 11). The type *Book* is defined in the DD and the input pin of the call behavior action *:PrintBookStatistic* referring to the variable *selectedBook* is, for instance, corresponding to the input variable book of the called ABD *PrintBookStatistic*. Further, the library system is decomposed in the ASD. The action *ShowBookStatistic* is, for instance, realized by the component *Client* in the DSD.

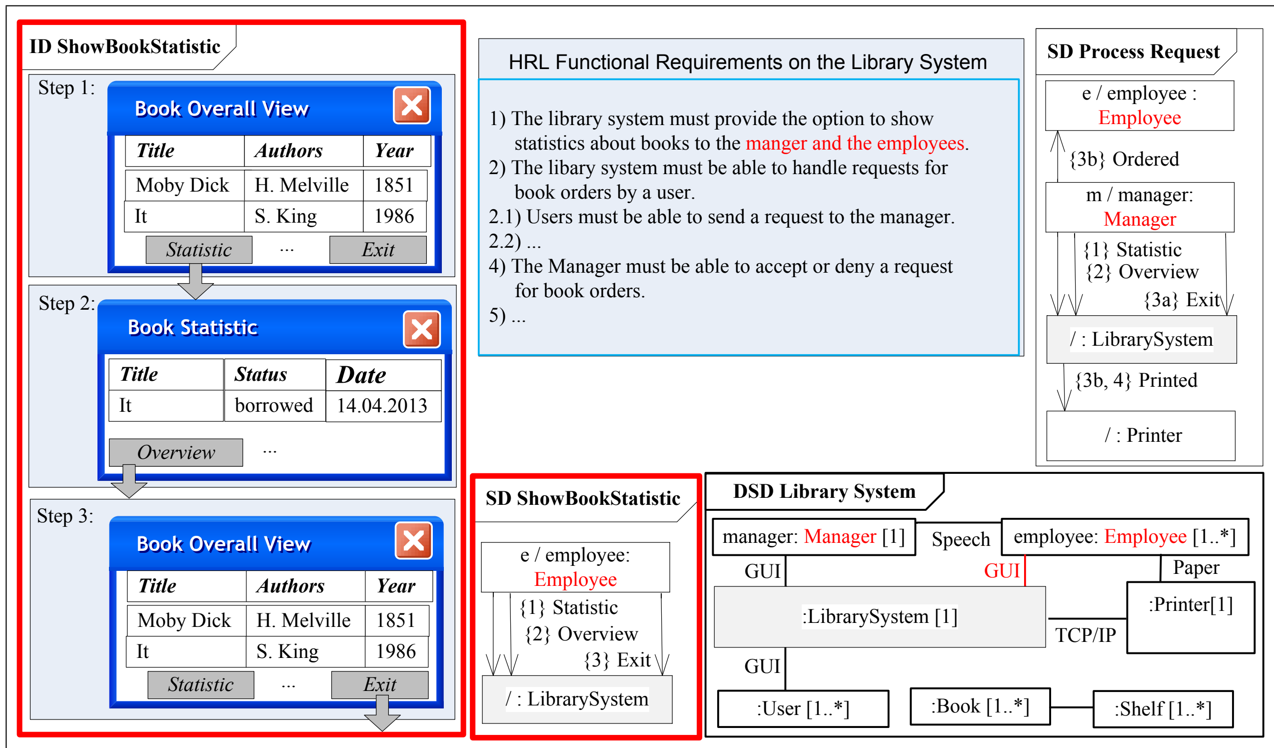

Figure 12. Change within requirements

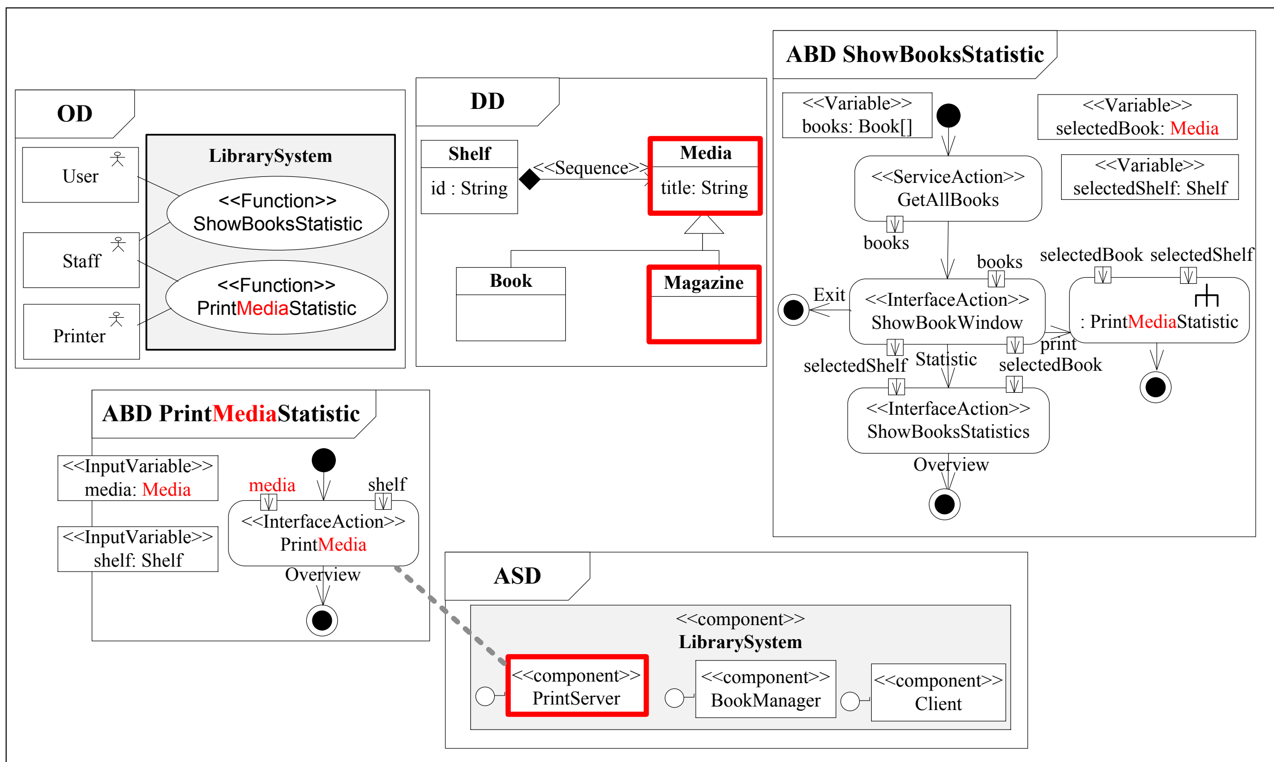

Figure 13. Change within architecture

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

In the following, typical changes of architectures are introduced in order to show the inconsistency detection and solving of the CREATE approach. A typical change during the development of an architecture is that the function can be reused in another context. In the library system example, the function *PrintBookStatistic* can only print the statistics of books. This function is changed to be able to print all data about media in general. Especially the library system should be able to handle also magazines. The ABD *PrintBookStatistic* is changed to *PrintMediaStatistic* (see Fig. 13). Due to this change, the type of the input variable is changed from *Book* to *Media*. The type Media is not defined in the initial DD (see Fig. 9). In consequence, the consistency constraint *2* is violated. This inconsistency is fixed by introducing this type in the DD. Further, the condition *4* is violated. The input pin of the call behavior action *:PrintBookStatistic* is, initially, referring to the variable *selectedBook* of the type *Book* (see Fig. 10) and the corresponding input variable of the called ABD *PrintBookStatistic* is of the type *Media* (see Fig. 13). One way to fix this inconsistency is to change the type of the passed variable *selectedBook* to *Media*. In order to allow the printing of book data by the function *PrintMediaStatistic*, the generalization between the type *Book* and the new type *Media* in the DD is introduced. Further, a new type *Magazin* is added, which is also generalized by the type *Media*. The action *PrintMedia* of the ABD *PrintMedaStatistic* is realized by a new component *PrintServer* of the ASD.

C. Constraints between requirements and architectures

Finally, CREATE defines strict interrelations between requirements and architectures. These interrelations are again defined by consistency constraints. An inconsistency means a violation of these constraints. Within architectures the following consistency constraints are defined:

1) The system boundary of the OD has to represent one type of the parts in the DSD.
2) Every actor of the OD has to represent one type of the parts in the DSD.
3) The existence of a type in DSD whose part is directly connected with the part of the system to build implies the existence of a corresponding actor in the OD.
4) The existence of an entity in the DD implies the existence of a corresponding type in the DSD.
5) The existence of a relation between two entities in the DD implies the existence of a connection between parts of the corresponding types in the DSD.
6) Every interaction mockup in the ID visualizing an interaction with the system must be realized by exactly one *InterfaceAction* in an ABD.
7) A system function of the OD realizes a set of interactions described in at least one SD.
8) A type in the DSD can either be represented by an actor of the OD or by an entity of the DD.

The initial requirements model and the initial architecture model of the library system example comply with all of these constraints. For instance, the system boundary of the library system in OD represents the type *LibrarySystem* of the DSD (see Fig. 11). The actor *Manager* represents, for instance, the type *Staff*. In this way, the connection between the parts of the library system and the part manager is valid. Between the entities *Book* and *Shelf* exists, for instance, a sequence relation and between the parts of the corresponding types in the DSD exists a connection. Every interaction mockup described in the IDs is realized by one interface action. For instance, the interaction mockup *Book Overall View* is realized by the interface action *ShowBookWindow* (see Fig. 11).

In the following, the changes of the requirements and the architecture of the library system in the previous sections are considered in order to show the inconsistency detection and solving of the CREATE approach. Due to the changes in the requirements specification the type *Employee* is introduced in the DSD. The part *employee* of this type is connected with the system in the DSD. Since no corresponding actor is described in the OD, the consistency constraint *2* is violated (see Fig. 14). The new data types *Media* and *Magazin* are added to the architecture during the further development of the architecture design according to the previous sections. In the DSD no part of these types is defined (see Fig. 14). As a result of this, the consistency constraint *4* is violated. Further, in our example the new interaction mockup *Book Statistic* is introduced in the new scenario described in the ID *ShowBookStatistic*. This interaction mockup is not realized by an interface action of an ABD in the architecture (see Fig. 14). Hence, the consistency constraint *6* is violated.

To correct these inconsistencies, a few further changes have to be made. It is necessary to add the entities *Media* and *Magazine* to the DSD. After this consistency condition *4* holds again (see Fig. 15). To comply with the consistency constraint *2*, a new actor for the employee has to be introduced into the OD (see Fig. 15). Finally, a mapping from the added interaction mockup to an interface action is missing. One could map the new interaction mockup to an existing interface action or extend the ABD by a new interface action. By extending the ABD by the interface action *EmployeeStats* the interaction mockup can be mapped on it (see Fig. 15). In this way, every interaction mockup is realized by one interface action and the model complies with the constraint *6*.

As shown above, the defined consistency conditions help at the consistency maintenance. The conditions can easily be checked. In this way, inconsistencies can be detected and solved. Furthermore, these consistency conditions enable an automatic support of the consistency maintenance [24]. An automatic consistency maintenance can, for instance, be realized by permitting changes to a next version not until all inconsistencies are solved.

VI. Evaluation

The development of CREATE took place at research projects in cooperation with a public institution over a period of four years. At these research projects, we gave advice and supported to system development projects in order to test our results in practice. The goal of the overall approach is to support consistency maintenance of requirements and architectures in early development phases. In these early phases, requirements

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

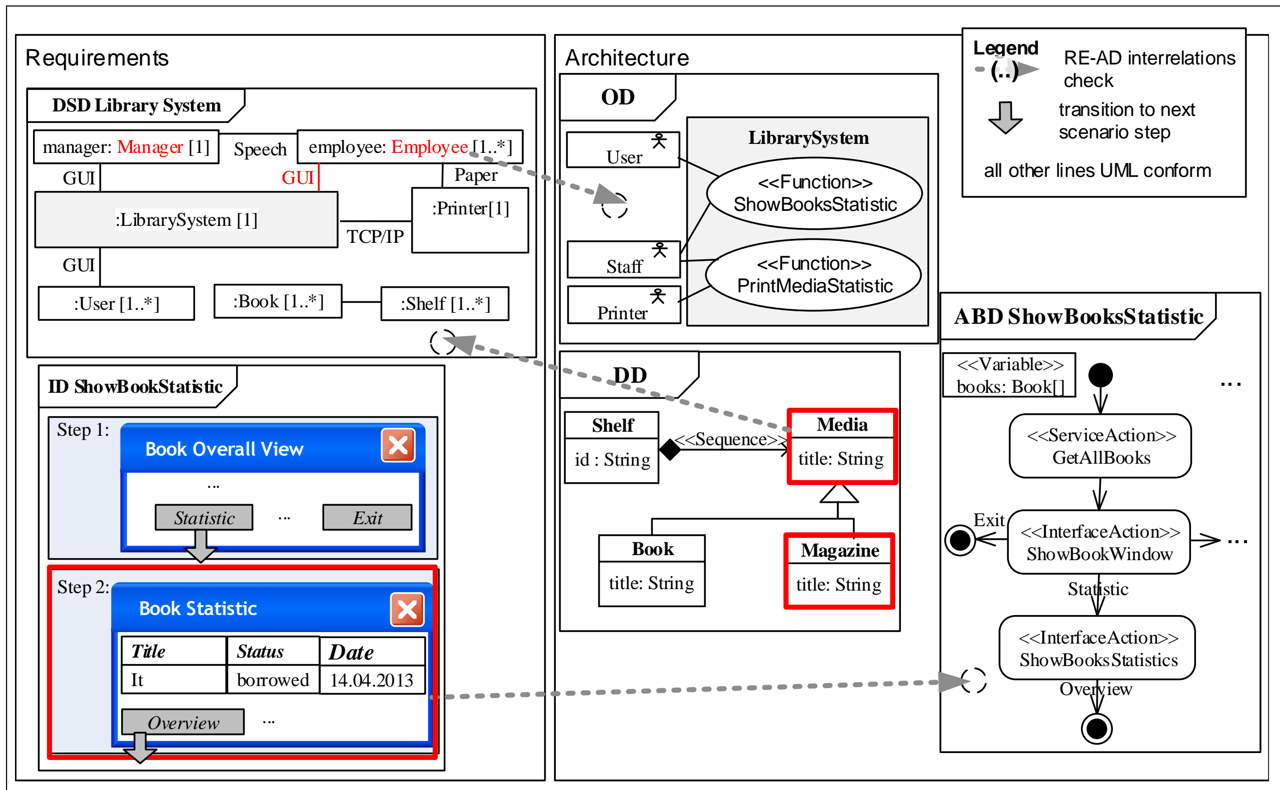

Figure 14. Changes at the requirements and architecture model

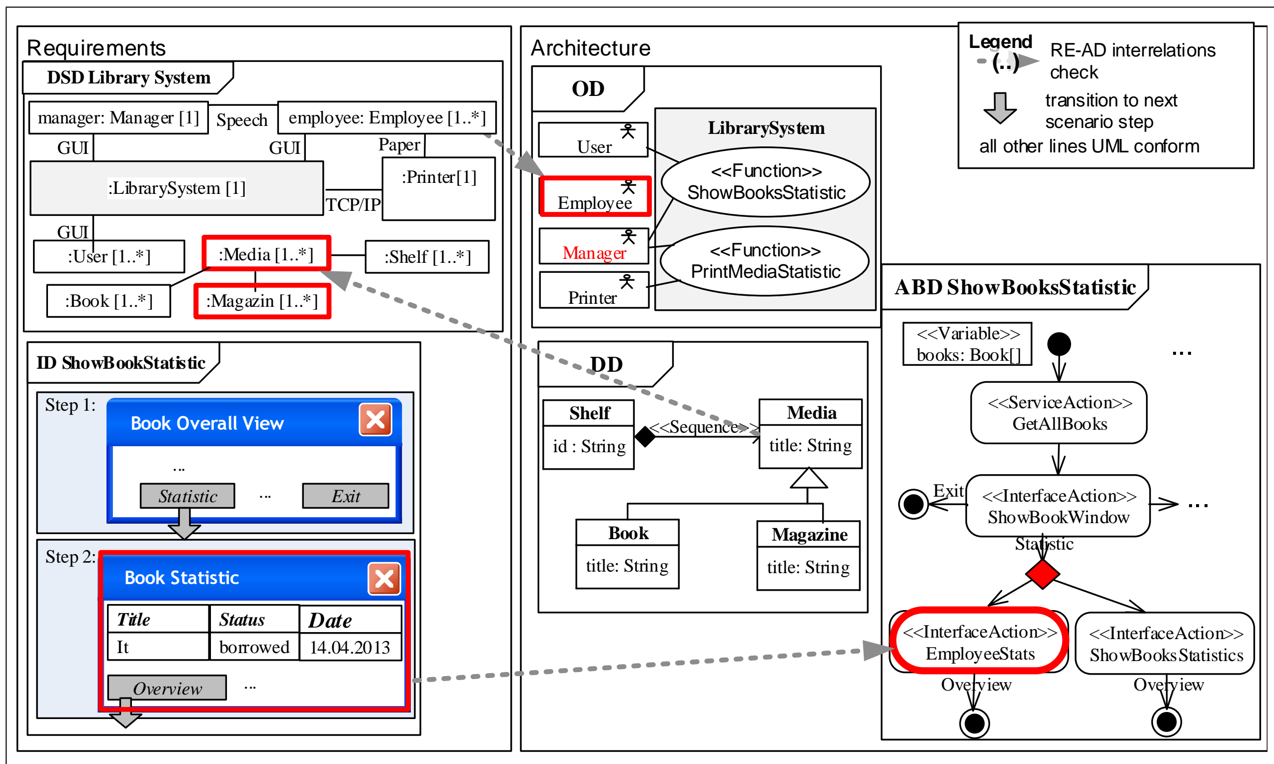

Figure 15. Changes to solve the inconsistencies

and architecture decisions are frequently changing. Further, feedback of stakeholders is crucial since the requirements on the system have still to be understood more clearly. The goal of the evaluation was to test the usability and the inconsistency prevention of our approach.

In a first step, we developed the component-based architecture model for a precise description of the architecture. For reconcilement with stakeholders we developed a prototype generator, which is able to interpret the developed models. The stakeholders should validate the architecture and the consistency to their requirements with the aid of the prototypes. This approach was tested at a system development project over a period of one year. The subject of this project was a communication system. At this project, a model of the complete system was developed comprising 20 system functions, 253 activity nodes and 35 data types. The models were developed in the sparx enterprise architect tool [26]. Conclusive, it revealed that the usability of the approach has to be improved. The number of possible states described by the component-based architecture leads to less comprehensibility to stakeholders. They were not able to agree to the developed specifications. Furthermore, the impact of changes was not readily understandable to stakeholders. Consequently, the consistency maintenance of requirements and architectures could not be supported by this approach.

Based on the results of this practice test we extended the approach by scenario-based models. This extended co-modeling approach was the model-based approach CREATE, which is described in detail in this paper. It was tested in practice at a further system development project with a similar subject over a period of one year. Besides the sparx enterprise architect we used balsamiq mockups tool for the description of the interaction mockups. In this period, the usability was significantly better. The required resource demand for creating and maintaining the models was still high with round about 4 person month. But stakeholders were able to agree to the visualized and scenario-based requirements. Furthermore, they were able to give helpful feedback, which leads to a big number of changes. We measured at three milestones the number of changes, the detected errors and especially remaining inconsistencies. Between these milestones we documented 500 changes and 67 errors. 8 of these errors were inconsistencies. The rate of inconsistencies to changes is low. For an indication, at a study described in [23], change data of requirements documents are analyzed. In this study, 88 changes, 79 errors, and 16 inconsistencies were detected. Furthermore, the resource demand for consistence maintenance was low. Only 8 inconsistencies had to be resolved. Parts of the the consistence checks could even be checked automatically.

VII. Conclusion and Future Work

The fundamental problem addressed in this paper was the development of inconsistencies between requirements and architectures at the advanced approaches for the iterative evolution. In this paper, the model-based approach CREATE [1] was described in all details, which supports the iterative evolution of requirements and architectures. The approach uses a scenario-based model for a precise description of requirements and a component-based model for the description of architectures. The architecture of CREATE describes precisely how requirements are fulfilled by the system under development. Requirements and architectural decisions lead frequently to inconsistencies between requirements and architectures. CREATE supports the consistency maintenance during the development of requirements and architectures by defined interrelations between scenario-based requirements models and component-based architecture models. The definition of these constraints support the consistency maintenance, because they can easily be checked. Furthermore, they enable an automation of the consistency maintenance [24]. This addresses the important concern of the scalability of the method when it is applied in complex industrial systems. During the development of such a system a large variety of requirements and architectural decisions have to be made. Since the consistence maintenance can be automated the approaches scales well with the size of the project.

A frequently stated argument is the entailment of high costs for the development of precise requirements and architecture models at a software project. This can be countered by the fact that an incorrect consideration of requirements not uncommonly leads to complete project failures. Thus, maintaining the consistency at the iterative evolution of requirements and architectures is important. Supporting this task by models enabling an automatic consistency maintenance reduces the risk of a project failure and costs for consistency maintenance. Furthermore, the developed models can be reused for automatic generation of code, test cases and documents like, for instance, requirements specifications. Nevertheless, the usage of formal models at a development project should, among others, be made conditional on the size of the project. At the beginning of a development project, the advantages and disadvantages of using formal models have to be weighed.

As future work, a further evaluation is planned to compare the effectivity of CREATE to other model-based approaches for requirements and architectures. Furthermore, it is planned to develop a tool for the automatic consistency maintenance.

References

[1] M. Ibe, M. Vogel, B. Schindler, and A. Rausch, "CREATE: A co-modeling approach for scenario-based requierements and component-based architectures," in Proceedings of the International Conference on Software Engineering Advances (ICSEA), IARIA XPS Press, 2013, pp. 220-227.

[2] B. Nuseibeh, "Weaving together requirements and architectures," IEEE Computer Society Press, vol. 34, March 2001, pp. 115–117.

[3] W. W. Royce, "Managing the development of large software systems: concepts and techniques," in Proceedings of the 9th International Conference on Software Engineering, IEEE Computer Society Press, 1970, pp. 1–9.

[4] B.W. Böhm, "A spiral model of software development and enhancement," IEEE Computer Society Press, vol. 21, May 1988, pp. 61–72.

[5] P. Grünbacher, A. Egyed, E. Egyed, and N. Medvidovic, "Reconciling software requirements and architectures with intermediate models," in Software and Systems Modeling. Springer, 2003, pp. 202–211.

[6] R. Ferrari and N. H. Madhavji, "The impact of requirements knowledge and experience on software architecting: an empirical study," in Working IEEE/IFIP Conference on Software Architecture, 2007, pp. 44–54.

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

[7] B. Nuseibeh and S. Easterbrook, ”Requirements engineering: a roadmap,” in Proceedings of the Conference on The Future of Software Engineering, ACM Press, 2000, pp. 35–46.

[8] C. Knieke and U. Goltz, ”An executable semantics for UML 2 activity diagrams,” in Proceedings of the International Workshop on Formalization of Modeling Languages, ACM Press, 2010, pp. 3:1–3:5.

[9] M. Elkoutbi, ”Automated prototyping of user interfaces based on UML scenarios,” in Journal of Automated Software Engineering, vol. 13, Kluwer Academic Publishers, 2006, pp. 5–40.

[10] K. Schneider, ”Generating fast feedback in requirements elicitation,” in Proceedings of the 13th international working conference on Requirements engineering: foundation for software quality, Springer-Verlag, 2007, pp. 160–174.

[11] OMG, ”UML, version 2.2. OMG specification superstructure and infrastructure,” 2009.

[12] A. G. Kleppe, J. Warmer, and W. Bast, ”MDA explained: the model driven architecture: practice and promise,” Addison-Wesley Longman Publishing Co. Inc., 2007

[13] C. Atkinson, J. Bayer, and D. Muthig, ”Component-based product line development: the KobrA approach,” in Software Product Line Conference, Denver, Kluwer Academic Publishers, 2000, pp. 289-309.

[14] R. Chitchyan, M. Pinto, A. Rashid, and L. Fuentes, ”COMPASS: composition-centric mapping of aspectual requirements to architecture,” in Transactions on AspectOriented Software Development, 2007, pp. 3–53.

[15] R. Kazman, G. Abowd, L. Bass, and P. Clements, ”Scenario-based analysis of software architecture,” in IEEE Software, vol. 13, IEEE Computer Society Press, Nov. 1996, pp. 47–55.

[16] H. Liang, J. Dingel, and Z. Diskin, ”A comparative survey of scenario-based to state-based model synthesis approaches,” in Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools, ACM Press, 2006, pp. 5–12.

[17] Y. Bontemps, P. Schobbens, and C. Löding, ”Synthesis of open reactive systems from scenario-based specifications,” in Proceedings of Application of Concurrency to System Design, 2003, pp. 41–50.

[18] V. Garousi, L. Briand, C. Lionel, and Y. Labiche, ”Control flow analysis of UML 2.0 sequence diagrams,” in Model Driven Architecture Foundations and Applications, 2005, pp. 160–174.

[19] C. Szyperski, ”Component software: beyond object-oriented programming,” Addison-Wesley Longman Publishing Co. Inc., 2002.

[20] A. Rausch, R. Reussner, R. Mirandola, and F. Plasil, ”The common component modeling example: comparing software component models,” ser. Springer Lecture Notes in Computer Science, vol. 5153, 2008.

[21] A. Egyed, E. Letier, and A. Finkelstein, ”Fixing inconsistencies in UML design models,” in Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, 2008, pp. 99–108.

[22] Y. Aoki, H. Okuda, S. Matsuura, and S. Ogata, ”Data lifecycle verification method for requirements specifications using a model checking technique,” in Proceedings of the International Conference on Software Engineering Advances (ICSEA), IARIA XPS Press, 2013, pp. 194–200.

[23] V. R. Basili and D. M. Weiss, ”Evaluation of a software requirements document by analysis of change data,” in Proceedings of the 5th International Conference on Software Engineering, IEEE Press, 1981, pp. 314–323.

[24] B. Schindler, and A. Rausch, ”Automatic consistence maintenance of requirements and architectures,” in Proceedings of the IASTED International Conference on Software Engineering, ACTA Press, 2014, pp. 15–22.

[25] B. Schindler, ”Konsistenzsicherung von Anforderungen und Architekturen,” Technische Universität Clausthal, 2014.

[26] D. Steinpichler, ”Project management with UML and Enterprise Architect,” SparxSystems Eigenverlag, 2011

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org