AUTOMATIC CONSISTENCE MAINTENANCE OF REQUIREMENTS AND ARCHITECTURES

Björn Schindler and Andreas Rausch
Institut für Informatik
Technische Universität Clausthal
Clausthal-Zellerfeld, Germany
{bjoern.schindler, andreas.rausch}@tu-clausthal.de

ABSTRACT
Requirements engineering and architectural design are important for a successful development of large-scaled software systems. Requirements and architecture decisions are frequently changing in early development stages. Hence, these activities are strongly interrelated. The development of inconsistencies at the iterative evolution of requirements and architectures is a fundamental problem. Inconsistencies cause an incorrect consideration of requirements on the software system and unfulfilled requirements. Existing model-based approaches allow a precise and formal description of requirements and architectures by structural and behavioral models in order to avoid misinterpretations. An automatic consistence maintenance of requirements and architectures would solve the problem of inconsistencies. But an automatic consistence maintenance of behavioral models is challenging. In this paper, an approach for an automatic consistence maintenance of requirements and architectures is presented. This approach supports the consistence maintenance of structural as well as behavioral models.

KEY WORDS
Requirements architecture automatic consistence maintenance.

1 Introduction
Requirements Engineering (RE) and Architectural Design (AD) are crucial for a successful development of large-scaled software-intensive systems. In classical development processes, artifacts like requirements specifications and designs are developed sequentially. This is also the case in iterative process models like the spiral life cycle model of Böhm [1]. Unfortunately, the fulfillment of particular requirements is frequently prohibited by architectural constraints [3]. This frequently causes a change to requirements or a selection of a different appropriate architecture. A further reason of the iterative evolution of requirements and architectures is the discovery of additional requirements during the development process. If design decisions were made early in the development process, they are very hard and costly to change later.

Process models like SCRUM [2] consider the change of early design decisions by an iterative and incremental development of software-intensive systems. Nuseibeh describes the twin peaks model [3], which focuses on the iterative evolution of requirements and architectures at the development of large-scaled systems. It intertwines the RE and AD activities into an iterative evolutionary software development process. In this process, requirements and architectures have an equal status and are iteratively and progressively refined. This is illustrated by twin peaks (see Fig. 1). The development of inconsistencies is a fundamental problem of the iterative evolution of requirements and architectures. Inconsistencies cause an incorrect consideration of requirements on the system and unfulfilled requirements. A solution would be an automatic consistence maintenance of requirements and architectures. Therefore, consistence constraints have to be formally defined. The twin peaks model by itself is kept very general. For instance, it does not specify the level of detail of requirements in relation to the architecture [4]. The description technique has to be precisely defined for a definition of consistence constraints. Requirements and architectures should be described according to the following well-known guidelines:

- Requirements description should be precise and complete to avoid misinterpretations. At the elicitation process, stakeholders should be integrated. Therefore, requirements descriptions should be precise and complete as well as comprehensible. These guidelines are also mentioned by Nuseibeh [5].

Figure 1. The twin peaks model [3]
• The architecture of a software system should describe how requirements are satisfied. At AD, abstract descriptions need to be progressively refined. The additional details have to be separated from requirements description in order to enable a verification of the fulfillment of requirements. Architecture descriptions have to be entire for an implementation specification.

Existing description techniques, like the model-based approach CREATE [6], fulfill these requirements. These approaches define structural and behavioral models for the description of requirements and architectures (see Sec. 4 for an example). Furthermore, these approaches define consistence constraints. But there is actually no appropriate approach available for an automatic consistence maintenance of precise requirements and architectures. An automatic consistence maintenance of structural models is comparatively easy, because consistence can be defined between types. For instance, CREATE defines consistence between types. In contrast, an automatic consistence maintenance between behavioral models of requirements and architectures is challenging. A scenario-based behavior model, e.g., a Message Sequence Chart (MSC), is suitable for the description of requirements and describes interactions between multiple objects [8][6]. A state-based model, e.g., a Petri Net (PN), is suitable for the description of architectures [19][6] and describes the entire behavior of single objects [8]. Hence, an automatic consistence maintenance of requirements and architectures can be achieved by a consistence maintenance of these models. An approach should consider the following requirements:

• The fulfillment of requirements has to be verifiable. Hence, solution details of the architecture have to be separated from requirements. Consequently, requirements and architectures need to have a refinement relation.

• The architecture description is abstract in the important early development phase. Dependencies between parts of the behavior descriptions are not formally defined. Hence, these dependencies cannot be automatically detected. This requires manual decisions during an automatic consistence maintenance.

• Since the architecture needs to be detailed enough for an implementation specification, many model-based approaches are using a Turing-complete architectural behavior model like a higher PN [6][19]. Hence, the approach should be applicable to such expressive models.

In this paper, an approach for an automatic consistence maintenance of requirements and architecture models is presented, which satisfies these requirements. This approach supports the consistence maintenance of structural as well as behavioral models. It is successfully applied in the domain of interactive information systems like web-based systems and might be applicable to other domains.

In Sec. 2, existing approaches are considered for an automatic consistence maintenance of requirements and architectures. Sec. 3 contains a description of the overall approach. In Sec. 4, the description technique of CREATE is introduced. Sec. 5 presents the application of our approach at this example. Sec. 6 contains a discussion and pending points for future work.

2 Related Work

Existing approaches for automatic consistence maintenance of models can be categorized into model transformation approaches and consistence checking approaches. Model transformation approaches can be categorized into unidirectional and bidirectional approaches.

An automatic consistence maintenance of requirements and architectures can be achieved by a consistence maintenance of scenario-based and state-based models (see Sec. 1). Unidirectional approaches for this issue are summarized in [8]. Examples are the approaches described in [11][15]. In [11], state charts are synthesized from MSCs. States are added between the messages of the MSC. Afterwards, the state chart is synthesized by transforming the messages into transitions. In [15], Unified Modeling Language (UML) state diagrams are synthesized by UML collaboration diagrams. Collaboration diagrams are transformed to partial object specifications, which are afterwards merged to complete object specifications. Unidirectional transformation approaches are suitable for many purposes like generation of prototypes. But the refinement relation between requirements and architectures demands that many architecture models are consistent to one requirements model and vice versa. Unidirectional transformation approaches do not consider the target model, and do consequently not retain expensive further developments of requirements and architectures.

Bidirectional model transformation approaches consider the target model at the transformation [12]. Since refactoring methods describe the impact of changes on models, these approaches can be considered as bidirectional transformations. Bidirectional approaches can be categorized into approaches maintaining a complete consistence in the sense of a bijection and approaches considering abstraction. A bidirectional model transformation approach of the first type is described in [10] and [16]. In [10], consistence constraints are defined between UML sequence diagrams and UML activity diagrams. If, for instance, a new model element is added to an activity diagram, a corresponding element has to be added to the sequence diagram. These approaches are suitable to maintain consistence between models with an equal level of detail. Since requirements and architectures have a refinement relation, a consistence maintenance in the sense of a bijection is not applicable. Representative bidirectional approaches considering abstraction are described in [13][14]. At these approaches, lenses are used to transform concrete models into abstract models. Lenses are also suitable in the symmetric
case, in which one model is an abstraction of the other and vice versa at the same time. Unfortunately, dependencies between parts of the architectural behavior descriptions are not formally defined in the important early development phases. Hence, these dependencies cannot be automatically detected. This requires manual decisions during an automatic consistence maintenance. Some approaches allow manual decisions during the transformation [20]. Since the architectural behavior model is Turing-complete, the number of potential transformations increases strongly with the size of the model. A selection of an appropriate transformation out of this set is not practical. In the case of manual decisions, an adequate bidirectional transformation is an automatic consistence check [12].

Consistence checking approaches can be categorized into approaches checking a complete consistency in the sense of a bijection and model checking approaches. Consistence checking approaches of the first type are, for instance, described in the synthesis approach [15]. After transforming UML collaboration diagrams to state diagrams, it is checked whether the scenarios are complete in respect to the synthesized state diagrams. As mentioned before, approaches maintaining consistence in the sense of a bijection are not applicable for the consistence maintenance of requirements and architectures. Model checking approaches are, for instance, described in [9] and [17]. In [9], consistency is checked between universal life sequence charts and I/O automata. In this case, consistency means the I/O automata satisfy the sequence charts specification. In [17], the consistence is checked between action based and reactive systems. Reactive systems perform action sequences in reaction to events. Action based systems perform action sequences and events. The models are consistent, if the reactive system is deadlock free at the events triggered by the action system. Model-checking the consistence between scenario-based models and state-based models means a check whether a sequence of transitions in the state-based model can be fired. This problem can be reduced to the reachability problem of a state. According to the requirements in the introduction the architectural behavior model is Turing-complete. The reachability problem in Turing-complete models is not decidable [18]. Consequently, this is true for the automatic consistence check.

In this paper, an approach is presented, which enables an automatic consistence maintenance of requirements and architectures. This approach supports the consistence maintenance of structural as well as behavioral models.

3 Overall Approach

In our approach, an automatic consistence maintenance of requirements and architectures is realized by an automatic consistence check. The consistence check considers the refinement relation as well as the executable semantics of Turing-complete architectural behavior model and is decidable. The information of the check can be used to maintain the consistence manually. In this way, dependencies between architecture elements can be considered, which are not formally defined in the important early development stages. Consistence can, for instance, be maintained by accepting changes for a next revision not until all inconsistencies are solved.

An automatic consistence check of structural models is comparatively easy, because consistence can be defined between types. Consistence constraints for structural models of the model-based approach CREATE are defined in Sec. 5.1. An automatic consistence check of behavior models is generally difficult, because of the required refinement relation and expressiveness. Automatic consistence maintenance of requirements and architectures can be achieved by a consistence maintenance of scenario-based and state-based models (see Sec. 1). A consistence check of these models considering the refinement relation and all possible states can be reduced to the reachability problem of a state in the architectural behavior model. This problem is not decidable for Turing-complete models [18]. The solution of our approach is to define consistence constraints on syntax level. These constraints consider the executable semantics of the behavior model, but not completely. A check of these constraints helps at the consistence maintenance and is decidable. The constraints can even be efficiently checked.

The scenario-based model MSC \textit{msc1} describes, for instance, an exchange of the messages 1 and 2 between the instances A and B (see Fig. 2 a). The state-based model PN B describes the behavior of the object B (see Fig. 2 b). Only the messages from and to this object are relevant for the consistence check. Hence, the scenario based model can be reduced to a sequence of scenario steps. At these steps, messages are exchanged. The MSC \textit{msc1} can, for instance, be reduced to the sequence of steps 1 and 2 in \textit{Scenario1} (see Fig. 2 c). A state-based model describes the control flow by nodes and edges. State-based models can be described by directed graphs for the consistence check.

Figure 2. General model of the consistence constraints
on syntax level. For example, the syntax of the PN $B$ can be described by the directed graph $DG1$ (see Fig. 2 d). Nodes of the state-based model are realizing messages of the scenario-based model. The message $i$ is, for instance, realized by the transition $I$ (see Fig. 2). Nodes of the directed graphs are typed in order to consider the executable semantics at the consistence check. The node $I$ of $DG1$ is of the type interaction node, since it is realizing a message.

Consistence of our approach is defined by constraints between messages of the scenario-based model and paths in the state-based model. If, for instance, messages are exchanged in sequence in the scenario-based model, a path has to exist between the realizing nodes in the state-based model. Otherwise the control flow could not reach the node, which is realizing the subsequent message. This path can include additional nodes for refinement. For instance, the messages $1$ and $2$ are exchanged in sequence and are realized by the nodes $I$ and $2$ (see Fig. 2 d). Consequently, a path has to exist between these nodes in the directed graph. In the directed graph of the PN $B$, this is true (see Fig. 2 d). The executable semantics can be considered more precise by using the node types on the determined paths. A path in the state-based model between nodes might also imply that realized messages have to be exchanged in sequence in the scenario-based model. This depends on the node types as well.

More than one message can be exchanged in one scenario step due to concurrency. The MSC $msc2$ describes a message sequence. In this sequence, the messages $2$ and $3$ are exchanged concurrently (see Fig. 3 a). This sequence can be reduced to Scenario2 (see Fig. 3 b). At the scenario step 2 of this scenario, the messages $2$ and $3$ are exchanged concurrently. In our approach, the constraints between messages of the scenario-based model and paths in the state-based model consider this. If, for instance, a new message is exchanged at a scenario step, a path has to exist between a realizing node of one message exchanged in the previous scenario step and the realizing node of the new message. For instance, in scenario step 2 of Scenario2 the new messages $2$ and $3$ are exchanged (see Fig. 3 b). Hence, a path has to exist between the nodes $I$ and $2$ as well as between $I$ and $3$ (see Fig. 3 c). In scenario step $3$, a path has to exist between the nodes $2$ and $4$ or $3$ and $4$.

In Sec. 5.2, the consistence constraints are defined at the example of the model-based approach CREATE, which provides a Turing-complete behavioral architecture model. The executable semantics of this model is considered at these consistence constraints by using node types. An automatic consistence check of these constraints helps at the consistence maintenance of iteratively evolved requirements and architectures. Furthermore, the check of these constraints is decidable. A check whether a path exists between two nodes is P-space complete [21]. Hence, the checks can be performed even efficiently.

## 4 Example Description Technique

In this Section, the description technique of the model-based approach CREATE is introduced as an example. The description technique is domain-specific for interacting information systems and is explained at the example of a library system.

**Requirements description.** In CREATE, requirements on the library system are described as follows:

- **Hierarchical Requirements List (HRL):** The HRL enables a text-based description of structural and behavioral requirements. They can be arranged hierarchical for refinement. In the HRL of the library system, the requirement $l$ is refined by the requirement $l.1$ (see Fig. 4 upper left).
- **Domain Structure Diagram (DSD):** The domain structure (e.g., the business structure) can be described by the DSD. It is based on UML composition structure diagrams. The domain structure consists of systems, persons and entities as well as their ability to communicate to each other described by parts and connections. In the case study, employees, books and the library system are derived from the HRL (see gray line in Fig. 4 upper left).
- **Scenario Diagram (SD):** Processes in the domain (e.g., business processes) are described by the SD, which is based on UML communication diagrams. SD describes representative scenarios in the domain, which have to be supported by the system under development. It describes message sequences between instances of the parts introduced in the DSD. The scenario ShowBookStatistic describes an interaction between an employee and the library system (see Fig. 4 upper left).
- **Interaction Mockup Diagram (ID):** ID describes the messages of the SD by interaction mockups for visualization to stakeholders. Interaction mockups are user interface mockups, which visualize interactions of the system in general. In ID, every message of the SD is detailed by exactly one interaction mockup. Scenarios can be visualized by showing the interaction mockups step by step. The interaction mockup Step 1 visualizes, for instance, books stored in the system (see Fig. 4 left).

**Architecture description.** The architecture model of CREATE describes how the software system fulfills given requirements by a complete description of the behavior and the resulting structure of the system.

**System Overview Diagram (OD):** OD describes the most abstract structure and behavior of the system and is based

---

Figure 3. Consistence constraints and concurrency
Figure 4. Requirements models, architecture models and inter-relations of CREATE [6]

on the UML use case diagram. The actors and the system context are derived from the DSD, the use cases (called functions) are derived from SD. In the example, the actor employee is associated with the function ShowBooksStatistics (see Fig. 4 upper right).

Architectural Behavior Diagram (ABD): ABD describes the behavior of the software system and is based on UML activity diagrams including data flow. The ABD describes the entire process of the functions defined in OD. The function ShowBooksStatistics is, for instance, described by the activity ShowBookStatistic (see Fig. 4 right). Within the ABD different action types like InterfaceAction and ServiceAction are used. An InterfaceAction describes an interaction of the system with its environment and is, therefore, associated with an interaction mockup of the ID. The action ShowBooksStatistics is, for instance, associated with an interaction mockup (see Fig. 4 right).

Data Diagram (DD): DD is based on UML class diagrams and describes the data types, which are used by data flows of the ABD. The DD of the library system describes, for instance, the data object Book, which is used in the ABD ShowBookStatistic. The DD describes, for example, the data object Book with the attribute title (see Fig. 4 right).

Architectural Structure Diagram (ASD): ASD is based on UML Component Diagrams 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 internal structure is derived from the actions of the ABD. Hence, each component is associated with an action of an ABD. The component LibrarySystem is refined by a component BookManager, which is associated with the action GetAllBooks in the ABD (see Fig. 4 upper right).

5 Consistence Constraints

In Sec. 3 consistence constraints between requirements and architectures are informally introduced, which can be efficiently checked. In this Section, these constraints are exemplarily and formally defined for the model-based approach CREATE, which defines inter-relations between and within requirements and architecture models. In the following, the former inter-relations are considered, since they are most crucial for the consistence maintenance (see red lines at Fig. 4).

5.1 Structural Models

An automatic consistence maintenance of structural models is comparatively easy, because consistence can be defined between types. The UML class diagram meta model - structural shows exemplarily a tool independent meta
model for structural models DSD and OD of CREATE (see Fig. 5 bottom). The classes DSM and OM describe instances of models defined by an DSD respectively an OD. Subordinate model elements like actors of OD are, for instance, described by the class Actor in the meta model. Consistency constraints are exemplarily described in the Epsilon Validation Language (EVL) (see Fig. 5 bottom). The first constraint defines the consistence of a DSD to an OD. A part of the DSD has a corresponding actor of the OD, if it is connected to the system. A possible change during the co-evolution of requirements and architectures is an addition of a new part manager of the type Manager to the DSD (see Fig. 5 upper left). If a link to a corresponding actor of OD is missing, the constraint is violated. A link to a new actor Manager of the OD resolves this inconsisteny (see Fig. 5 upper right).

The second constraint defines the consistence of the OD to the DSD. Every actor of the OD has a corresponding type in the DSD. Furthermore, a part of this type has to be connected to the system. A possible change is an addition of a new actor Manager (see Fig. 5 upper right). If the part of the Type Manager is not connected to the system, the constraint would be violated.

### 5.2 Behavioral Models

In CREATE, requirements and architectures are described by a scenario-based model and a state-based model respectively. These models have a refinement relation. Activity diagrams including object flow are used for architectures, which provide the expressiveness of a Turing-complete high level PN. Here, consistence constraints on syntax level are defined between these models according to the overall approach (see Sec. 3). These constraints consider the executable semantics of the behavior models, but not completely. A check of these constraints helps at the consistence maintenance and is decidable. The UML class diagram meta model - behavioral shows exemplarily a tool independent meta model for the consistence check on syntax level (see Fig. 6 bottom right). According to the overall approach, a scenario-based model is reduced to a sequence of scenario steps. The class Scenario in the meta model describes instances of models defined by the scenario-based SD and ID of CREATE (see Fig. 6 bottom left). Scenario steps are described by the class Step in the meta model. In CREATE, an interaction mockup of ID is a detailed representation of a message of the SD. An interaction mockup is described by the class Message in the meta model. The state-based model is described by a directed graph for the consistence check on syntax level (see Sec. 3). The activity nodes of the AVD are, for instance, described by the class Node in the meta model. Node types are described by the enumeration NType. The literals iact and dec represent, for instance, interaction nodes and decision nodes respectively. The object flow is not described in the meta model. It is not necessary for the consistence check in this stage of the development. Actions are not implemented yet. Hence, all possible variable states have to be checked, which means a check of all execution paths. Further, a variable state itself has no impact on the result of the consistence check. Consistency constraints between these models are exemplarily described in EVL. The first constraint defines the consistence of the scenario-based model to the state-based model (see Fig. 6 bottom right). If, for instance, a new message...
is exchanged in a scenario step, a path has to exist in the state-based model from a realizing node of one message exchanged in the previous scenario step to the realizing node of the new message. If a previous step exists, the operation `newMsgsOfStep` in this constraint returns all new messages of a scenario step. The operation `pathsToNextInteractionNodes` returns all paths from the current node to a node of the type `iact`. These paths do not contain other nodes of the type `iact`. A possible change at the co-evolution of requirements and architectures is an addition of a new scenario. The manager of the library requires other information about the books than the employees. Because of the new needs of the manager, a scenario is added (see Fig. 6 upper left). In this scenario, the mockup of the book overall view is realized by the action `ShowBookWindow`. After the book overall view a new mockup for the managers book statistics is added. A realizing action `ManagerStats` is added to the AVD (see Fig. 6 upper right). If no path exists from the action `ShowBookWindow` to the action `ManagerStats`, the first condition is violated. Adding a decision node and edges to the action `ManagerStats` solves this inconsistency.

The second constraint defines the consistency of the state-based model to the scenario-based model (see Fig. 6 bottom right). If, for instance, a path exists between two interaction nodes, the realized messages might have to be exchanged in sequence. In this constraint, the operation `isOptional` returns `true`, if the path has not to be traversed by the control flow. If the path, for instance, contains a decision node, it is optional. In the case of concurrency, an exchange of a message might be active at more than one scenario step. The method `lastStep` returns the last step, in which the message exchange is active. A possible change at the co-evolution of requirements and architectures is, for instance, a support of a new function for manager statistics. The architecture is extended by a new action `ManagerStats` and a path from the action `ShowBookWindow` is added (see Fig. 6 upper right). The path is optional, since a decision node is on this path. Consequently a scenario `ShowBookStatisticB` is consistent, if it is continued by a message, which is realized by the action `ManagerStats` or `ShowBooksStatistics`.

Since the computation of paths between two nodes is P-space complete [21], these checks can be performed efficiently.

### 6 Conclusion

An automatic consistence maintenance of requirements and architectures reduces the risk of unfulfilled requirements caused by inconsistencies. In this paper, an approach for an automatic consistence maintenance of requirements and architecture models was presented. This approach supports the consistence maintenance of structural as well as behavioral models. Requirements and architectures have a refinement relation. Furthermore, details of architectures are generally not formally described in the important early development phases. This requires many manual decisions at an automatic consistence maintenance. In this case, an automatic consistence check is an adequate approach.
Many model-based approaches are using a scenario-based requirements model and a state-based as well as Turing-complete architecture model. A consistence check considering all possible states of these models is not decidable. The solution of our approach was the definition of consistence constraints on syntax level. These constraints consider the executable semantics of the behavior models, but not completely. A check of these constraints helps at the consistence maintenance and is decidable. The constraints can even be checked efficiently.

In our approach, constructs like concurrency, loops and variables of the state-based model are considered. Further constructs like hierarchies are actually not supported. In future work, a concept considering these constructs at the consistence check is developed. Furthermore, constraints are defined, which consider the executable semantics of behavior models in more detail. The goal is to detect as much inconsistencies as possible by retaining the decidability and preferably the efficiency of the consistence checks for large-scaled systems.

References


