Linking Design Decisions to Design Models in Model-Based Software Development

Patrick Könemann[1] and Olaf Zimmermann[2]

[1] Informatics and Mathematical Modelling, Technical University of Denmark, 2800 Kgs. Lyngby, Denmark
pk@imm.dtu.dk
[2] IBM Research – Zürich, Säumerstrasse 8, 8803 Rüschlikon, Switzerland
OLZ@zurich.ibm.com

Abstract. Numerous design decisions are made in model-based software development which often are not documented explicitly. Hence, the design knowledge is 'in the designers mind' and communicated orally, if at all, and the rationale behind the decisions is lost. Existing tools tackle this problem for architectural decisions which refer to the higher level architecture of a system. However, these decisions are separate artifacts and not linked to individual design model elements. Hence, there is no automatic check whether the design models comply with made decisions.

This paper presents concepts for explicitly linking design decisions and design model elements. As first class artifacts, design decisions can be used for documentation, consistency checking, and reuse. In case consistency constraints are violated, the user is notified that the design models no longer comply with the decisions made. Reuse is realized by extracting design model changes as reusable patterns for recurring decisions.

1 Introduction

Development of software systems is done in teams today, and models improve the communication within the teams and help to develop such systems. Model-based software development increases the productivity because the level of abstraction rises and models (e.g. in the *Unified Modeling Language*, UML [1]) are first class artifacts: the models are used for documentation, discussion, and to some extent also for code generation [2].

One way of documenting design decisions in such projects is the use of decision management systems. Decisions might either be specific to one particular project or generic, and thus reusable in similar contexts [3]. Reusable decisions, e.g. the use of design patterns [4] to solve a particular design issue, can be stored as best practices and reused in other projects. This makes design decisions valuable artifacts for expressing and sharing design knowledge.

The state-of-the-art decision management systems are only used for documentation, analysis, and for sharing architectural design knowledge [5,11,16], and are isolated from the actual models in model-based software development. All of these tools store the information semi-formally, i.e. structured by decisions.

M. Ali Babar and I. Gorton (Eds.): ECSA 2010, LNCS 6285, pp. 246–262, 2010.

Current modeling tools, on the other hand, have only limited or no capabilities for documenting design decisions. Hence, formal design models and semi-formal design decisions are separated. Our previous work already introduced decision enforcement as a proof-of-concept which is the first step to update design models according to made design decisions [6].

The concepts in this paper introduce an explicit link between design models and design decisions in model-based software development. Our vision is to treat design decisions as first-class artifacts and to exploit them to integrate design models and semi-formal documentation: an explicit link between design model elements and design decisions will allow keeping the design models consistent with the decisions made. Moreover, we propose concepts for automating redundant work on design models with the use of model differences—the latter are used to store reusable design model changes that realize recurring design decisions.

All concepts are tool-independent; to integrate another modeling or decision management tool, the other tool has to realize the interface specified in [7]. In essence, a modeling tool must offer reflection (e.g., as EMOF [8] provides) and design decisions must be mapped to the decision meta model of the interface.

The main benefits of our contributions are reuse of design decisions and the corresponding changes in the design models as well as automated recognition of consistency violations between these artifacts. The goals is to make the development of model-based software faster and less error-prone (because of reuse).

The remainder of the paper is structured as follows. Sect. 2 introduces an example, Sect. 3 states the state of the art as well as our goals, Sect. 4 defines our central concept of a binding and its use, Sect. 5 sketches the prototypic implementation, Sect. 6 discusses related work, and Sect. 7 concludes the paper.

2 Example

In this section, we introduce a running example to illustrate our concepts. It is small on purpose in order to focus on relevant properties. It consists of two decisions, taken from a case study in [7], that describes the development of a web application with respect to made design decisions. Here we assume that the design decisions to make (described by design issues and their solutions, called alternatives) are already known and available in a decision management tool.

The first issue, *Session Awareness*, concerns an existing class *Controller* in the UML design model (which was created due to a previous design decision) and deals with the issue whether or not to introduce session support in the web application. Possible solutions are *Yes* and *No*, as sketched informally in a simplified decision model in Fig. 1. Here we make the decision to pick the alternative *Yes* which induces another issue *Session Management*. Note that although this particular choice does not affect the design models directly, subsequent decisions in fact can have impact on the design models as explained next.

The second issue, *Session Management*, concerns *how* the session management will be realized. The choice will be the *Server Session State* pattern (as defined by Fowler [9]) describing a controller, a session manager, and a session object. Other

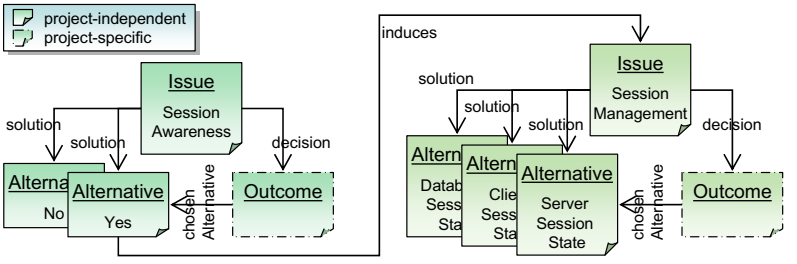

Fig. 1. Design decisions *Session Awareness* and *Session Management*

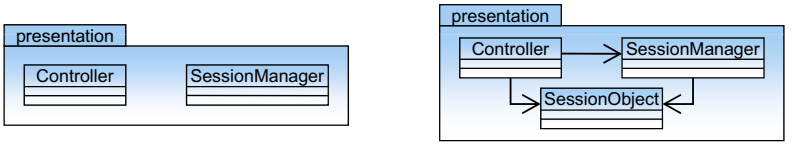

Fig. 2. Parts of the design model before and after the decision *Session Management*

alternatives are *Client Session State* and *Database Session State*. As depicted on the left-hand side of Fig. 2, a session manager already exists – for instance, due to previous work on the design model.

The next step is the realization of the server session state in the design model, i.e. adding design model elements according to the chosen pattern (the result is shown on the right-hand side of Fig. 2). That work is usually tedious and error-prone although it varies with the complexity of the selected solution. There might also be variations of how a particular solution can be realized in the design models. Moreover, the very same solution could have been realized before in another project, and, hence, its realization in design models is recurring work.

We use this example in the next sections to illustrate our approach that adds support for automatic consistency checking (whether the design models comply with made design decisions) and reuse of realizations of design decisions. Thus, design decisions are not lost but captured explicitly.

Although the example is dealing with a web application, all model-based software development processes are supported in which design decisions can be documented and recur in similar projects.

3 Requirements

This section gives an overview of the state-of-the-art of decision management, introduces model differences, and states the goals of our contributions.

3.1 Current Situation

Design knowledge in terms of design decisions consists mostly of informal information (text) structured as follows. A design decision in terms of the system's

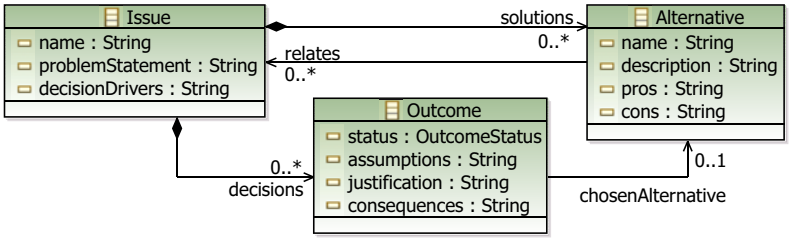

Fig. 3. A typical design decision metamodel in existing work

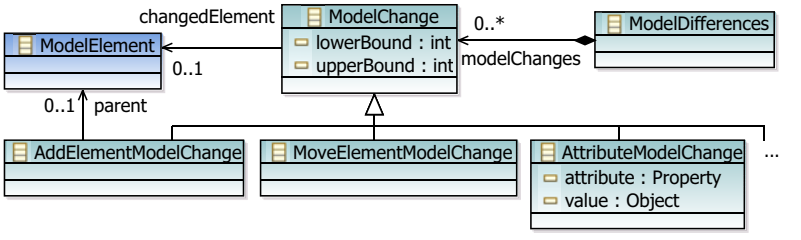

Fig. 4. An excerpt from a typical metamodel for model differences in existing work

architectural design consists of a design *issue* or *problem*, several *constraints* and *assumptions*, one or many *solutions*, and a *rationale*, amongst others [10]. The solutions describe *how* they shall be applied; in case of design patterns it might refer to its definition and/or informally describe its realization in the context of the issue. Moreover, we distinguish between *project-independent* and *project-specific* decisions, at which the former are reusable decisions and the latter are only documented for one particular project [11].

A typical design decision metamodel is shown in Fig. 3 which can be mapped to multiple decision management tools: a design decision addresses a particular design problem (*Issue*), considering one or many solutions (*Alternatives*), and the rationale why a particular alternative was chosen (*Outcome*). Attributes like *problemStatement* and *justification* describe the properties mentioned before. The association *relates* between alternatives and issues allows relating decisions to each other; the reference *induces* in Fig. 1 is an instance of it. This metamodel is based on the one specified in [6], in particular concerning the distinction between project-independent parts (issues and alternatives) and project-specific parts (outcomes). Design model changes are, however, not included in any existing work of decision management we are aware of.

Model differences describe changes in a design model, e.g., adding a class and three associations (cf. decision 2 in the example in Sect. 2). Hence, they can be used for describing design model changes for a particular realization of a solution of a design decision. Here it is sufficient to know that model differences consist of several *ModelChange*s as shown in Fig. 4. Concrete changes are, for instance, addition or movement of elements or change of attributes ([12] discusses

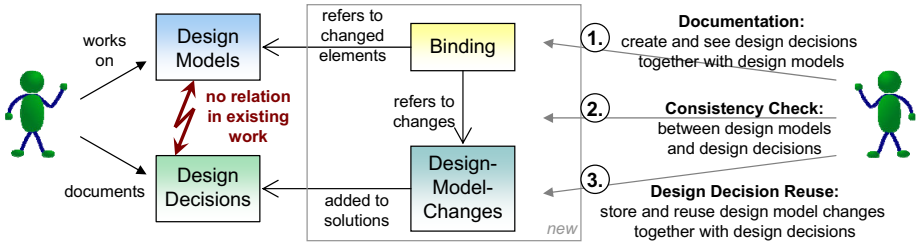

Fig. 5. Binding and design model changes enable our goals

selected differencing approaches in more detail). *lowerBound* and *upperBound* of *ModelChange* are specific for our differencing technology [13] and define how often a particular change may be applied.

We require our solution to be independent of particular tools. That is to say, the metamodels for design models and design decisions shall not be modified. This ensures that the concepts for the binding are tool-independent and are thus applicable to many modeling tools and decision management systems.

3.2 Goals

This section states the goals of our contributions. The left-hand side of Fig. 5 sketches the situation without our extensions: design decisions are isolated from the design model. Adding a *Binding* and *DesignModelChanges*, as shown in the center of the figure, enables our goals to *ease documentation*, to *check consistency*, and to *reuse design model changes of design decisions*.

Goal: Documentation. Almost every change in design models can be seen as a design decision. However, most decisions, even if they are made consciously, are not documented because of lacking tool support and developers lacking discipline. To overcome that problem, our goal is to explicitly link design models to the design knowledge stored in a decision management system. That is, related design decisions can be retrieved for each element in the design model. For instance, if the developer selects the class *SessionObject* (cf. Sect. 2), the tool shall return a list of design decisions containing the decision *SessionManagement*.

Goal: Consistency. Another goal is to validate whether design decisions and their induced changes in the design models are consistent with each other. That is, for the decision *Session Management* in the example, the class *SessionObject* and the three associations between that class, the *Controller*, and the *SessionManager* must prevail in the design model. If any of these classes or associations are removed later on, the design model is not consistent anymore with the result of the decision and the developer shall be notified.

Goal: Design Decision Reuse. The last goal addresses reuse of design decisions in the same or in a similar context, e.g. in another project. Realizing the

same solution multiple times in one or several projects is recurring and error-prone work. Design model changes of a particular design decision, in the form of model differences, can be extracted from one design model and applied to another design model the next time that decision is made. That will not happen fully automatically but the developer has to revise (and, if necessary, refine) the application of design model changes. That said, there might be similar design model changes which realize the same solution, depending on the scenario and context; there are, for instance, multiple realizations of the server session state pattern. Hence, our goal is to support multiple realizations per solution.

4 Concepts: Binding Design Knowledge to Design Models

Next, existing and new components are introduced, the binding is defined, and finally we explain how to use the binding for documentation, consistency checking, and reuse of design decisions.

4.1 Relevant Components

This section lists all relevant existing components before defining a formal link between design decisions and design models. The link has to connect the particular decision, more precisely the *Outcome* of a decision and its chosen *Alternative* (cf. design decision metamodel in Fig. 3), with the design model elements the decision affects. A design model element can be any part of the design model; in case of UML it would be instances of *Element*, that are, for example, classes, associations, attributes, actors for use cases, or messages in a sequence chart [1].

ModelDifferences, which describe *how* a model should be changed when a design decision is made, contain a set of individual *ModelChange*s (cf. metamodel for model differences in Fig. 4).

4.2 Binding between Design Decisions and Design Models

This section defines a new artifact, the *DecisionModelBinding*, to achieve our goals listed in Sect. 3.2. We first sketch it informally with the help of the running example before we define it. Its purpose is to map each change from the model differences to the design model elements the particular design decision affects.

In case of the design decision *Session Management* from the example in Sect. 2, the binding consists of a couple of *ModelElementBinding*s, one for each design model element. Fig. 6 sketches the overall picture for that decision. The design model on the left-hand side and the design decisions on the right-hand side are already known from Sect. 2. The center part shows the project-independent *ModelDifferences* and the project-specific *DecisionModelBinding*. That is, the *ModelDifferences* describe *how* the design model is changed for that specific alternative (this is the reusable realization of the alternative). The *DecisionModelBinding* links these changes to the actual design model elements. We made this separation because the *ModelDifferences* are reusable and, thus, project-independent (required for *Goal: Design Decision Reuse*) whereas the *DecisionModelBinding* is only used for one particular design model and is project-specific.

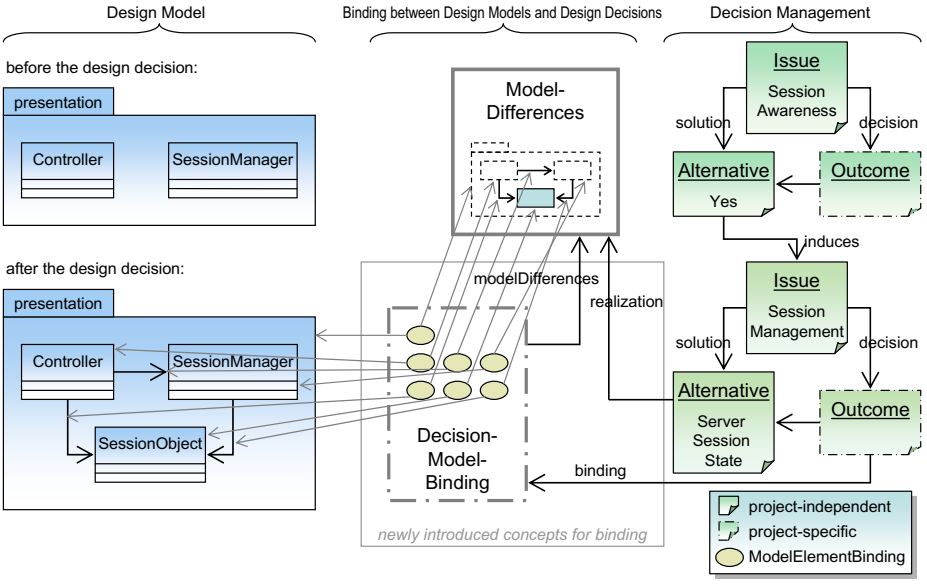

Fig. 6. Example binding between design decisions and design model

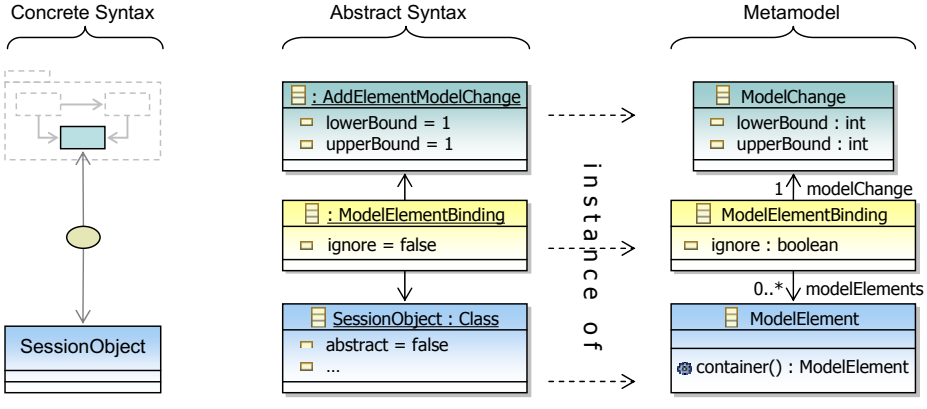

Fig. 7. One *ModelElementBinding* in concrete and abstract syntax and its metamodel

Figure 7 shows one of the binding elements in detail, namely the binding for the added class *SessionObject*. The left-hand side shows the concrete syntax whereas the abstract syntax (UML object diagram) of that binding is shown in the middle. The *ModelElementBinding* contains references to both, the change *AddElementModelChange* in the model differences and to the *SessionObject* in the design model. Furthermore, the figure shows the metamodel elements for this scenario on the right-hand side. The *lowerBound* and *upperBound* define how many design model elements are allowed for a particular binding.

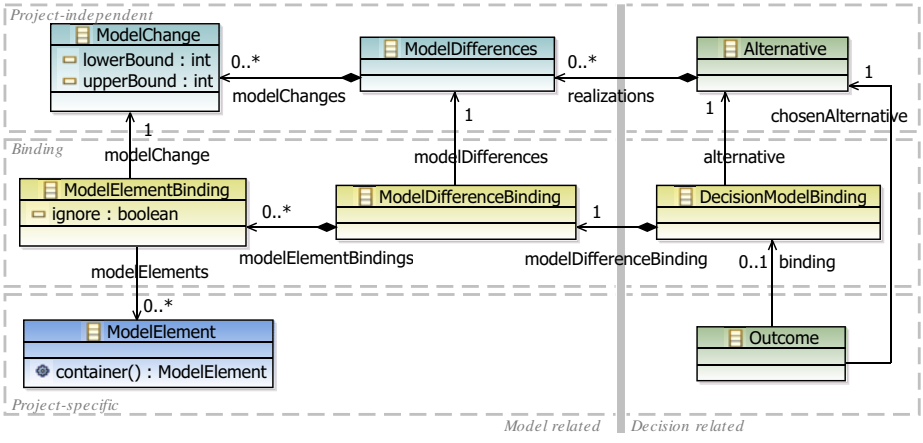

Fig. 8. Metamodel of the binding between design decisions and design model

The definition of the binding is given as a metamodel in Fig. 8. The classes in the middle row define the binding; the *DecisionModelBinding* and *ModelElementBindings* in Fig. 6 are their instances, respectively. The classes are vertically divided into being design model-related and decision-related. The other components in the figure (already introduced in Sect. 4.1) are horizontally divided into being project-independent and project-specific. They are explained next.

- An *Alternative* is a solution in a design decision and may contain several realizations. An *Outcome* is the result of a particular decision and points to the chosen alternative.
- A *ModelChange* defines an individual change in the design model, *ModelDifferences* groups them logically as a realization for an alternative.
- A *ModelElement* is an arbitrary element in the design model.
- A *ModelElementBinding* links a *ModelChange* to the affected *ModelElement* (if *ignore* is true, this binding is not validated); a *ModelDifferenceBinding* groups the binding logically; a *DecisionModelBinding* connects the *ModelDifferenceBinding* to the outcome, that is, to the result of the decision.

Note that the design model does not know about the binding because we do not want to modify the modeling tool. Moreover, *Alternative* and *Outcome* are just wrapper classes for alternatives and outcomes in a decision management system. Thus, the references *Alternative.realizations* to *ModelDifferences* and *Outcome.binding* to *DecisionModelBinding* belong to these wrapper classes. This level of indirection keeps all decision related classes independent of the binding.

Since the explicit binding links design model elements and design decisions, the rationale and other documentation can directly be annotated to the design model (Goal: Documentation).

4.3 Consistency Check

It is easily possible to check the consistency between design models and made design decisions with the binding concepts introduced in Sect. 4.2. We defined a set of constraints for that purpose, for instance that added elements must prevail in the design model. A violated constraint produces either a warning or an error, specified by the constraint's severity. In case of constraint violations, the developer is notified with a description of the constraint and the cause. These constraints apply to design-time only.

Constraint Levels. In order to check that the design model corresponds to a made design decision, two criteria have to be checked. Firstly, all related design model elements must exist. Secondly, all design model changes defined by the model differences must prevail. Starting with these two criteria, we identified three levels with increasing granularity.

1. *Element level: all design model elements linked to the binding must exist.*
   This level is independent of design decisions and concrete changes in the design model but concerns only the relation between the binding and the existence and cardinality of design model elements.
   Example: the class *SessionObject* is referenced by a *ModelElementBinding* and, thus, must exist in the design model (if *ignore* is false).
2. *Change level: all changes must prevail in the design model.*
   This level is specific for changes which are made due to a design decision.
   Example: if a class is changed to being abstract, that change must prevail in the design model.
3. *Decision level: additional custom constraints for a particular decision.*
   Constraints in this level are specific for decisions and do not necessarily relate to model differences. They are specified manually by the developer during design-time.
   Example: the classes *Controller* and *SessionObject* must be located in the same package in the design model.

Constraints for the first two levels are static—we defined them once and for all. Custom constraints (decision level), on the other hand, can be specified by the developer and concern individual decision-related properties in the design model. Two examples from the element and the change level follow. We use the *Object Constraint Language* (OCL) [14] to define them as invariants.

Constraints Excerpts. The element level contains exactly the three invariants shown in Listing 1. They apply to all *ModelElementBinding*s (**context**) and ensure that the correct number of design model elements is bound. The cardinality is defined in the attributes *lowerBound* and *upperBound* of the class *ModelChange* (cf. binding definition in Fig. 8) and is checked in lines 4–5. The third invariant (line 8) checks that all referenced design model elements are defined which includes the check that added elements prevail in the design model.

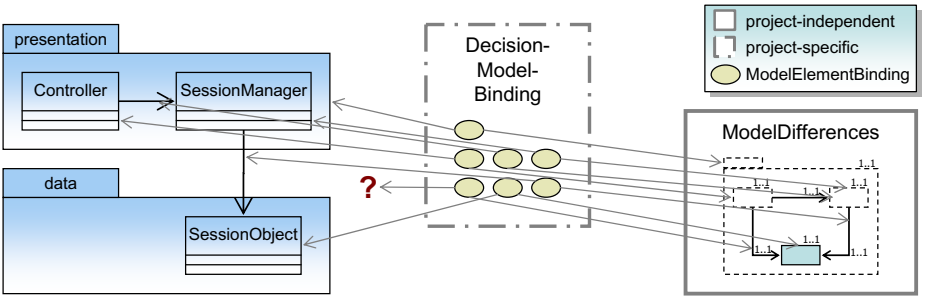

Fig. 9. Example scenario with constraint violations

```
1  context ModelElementBinding
2
3  -- the binding contains links to the correct number of model elements
4  inv lowerBound: modelChange.lowerBound <= modelElements->size()
5  inv upperBound: modelChange.upperBound >= modelElements->size()
6
7  -- all model elements exist and are defined
8  inv modelElements: modelElements->forAll(e | not e.oclIsUndefined())
```

Listing 1. Three constraints for the element level (severity: error)

In contrast to the element level, change level constraints check design model-specific properties, for instance whether a class is abstract or not. Thus, the constraints have to access properties which are design model specific. The constraint in Listing 2 checks whether added elements are contained in their expected containers. It uses the reflective call container()[1] (line 6) to retrieve the actual container and compares it with the expected value *AddElementModelChange.parent* (cf. metamodel in Fig. 4) in line 7. The invariant is only relevant for *AddedElementModelChange*s, hence the implication in line 5.

```
1  context ModelElementBinding
2
3  -- all added elements are contained in the expected parent
4  inv addedElementContainedInExpectedParent:
5    modelChange.oclIsTypeOf(diff::AddElementModelChange) implies
6      modelElements->forAll(e | e.container() =
7        modelChange.oclAsType(diff::AddElementModelChange).parent)
```

Listing 2. A constraint for the change level (severity: warning)

Example. The following example illustrates the consistency check. The left-hand side of Fig. 9 shows a modified design model: *SessionObject* was moved to another package and the association between *Controller* and *SessionObject* was removed. Consequently, not all design model changes induced by the design decision *Session Management* prevail. Using the constraints, we can automatically detect these violations: the upper and lower bounds of each *ModelChange* (denoted with *1..1* for model changes in Fig. 9) match the number of referenced

[1] This constraint requires an EMOF-compliant [8] metamodel because EMOF provides facilities for reflection like the operation container() : ModelElement.

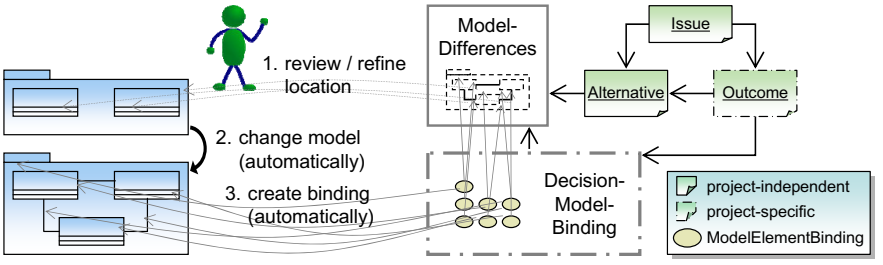

Fig. 10. Reusing Design Decisions by applying model differences

design model elements, so we are safe here. However, one *ModelElementBinding* has a dangling reference, i.e. it points to a design model element that does not exist anymore (invariant in line 8 of Listing 1). This violation is presented to the developer with the severity *error*. Moreover, the parent for the added element *SessionObject* differs from the one defined in the model change (invariant in Listing 2). This violation, in contrast, is presented with the severity *warning* because the added element still exists.

These constraints ensure that for each design decision all relevant design model elements exist (element level) and that all changes prevail in the design model (change level). Hence, it is now automatically possible to verify that design decisions are realized in the design model (Goal: Consistency).

4.4 Reusing Design Decisions

Up to now, we defined the binding and explained its use for consistency checking. The question is how to create these bindings. Similar to having the design knowledge already predefined in some decision management system, we assume that the model differences representing design model changes have been created and attached to an alternative in advance. At this point, one can think of these model differences as a design template extracted from a sample model, a previous project, or a pattern repository.

Next, we explain how a binding is created as a result of a made design decision (sketched in Fig. 10). Once the decision *Session Management* is made, i.e. the developer selects a particular alternative, she/he chooses one of the attached realizations (in form of model differences):

1. The developer has to review/refine the location for applying the changes to the design model. In the example, the package *presentation* and the two classes *Controller* and *SessionManager* must be selected.
2. The design model is (automatically) changed according to the model differences. As for any automatic step, it is recommended to review all changes.
3. Then the binding is (automatically) created and contains one *ModelElementBinding* for each changed design model element.

Overall, the only manual work for realizing a design decision in the design model is to define the correct location where to apply the design model changes and

to review them afterwards (instead of manually changing the design model). The binding can then be used for the goals *Documentation* and *Consistency Checking*.

5 Realization

This section gives some insight into the realization (architecture and some parts of the GUI) of our prototype. All concepts presented in Sect. 4 are implemented.

Architecture. Here we briefly outline the architecture of the prototype. We have chosen Eclipse as the base platform because many technologies already exist for reuse and because it is easily extendable. Figure 11 informally sketches the dependencies between used and new components. Their purpose in the prototype is explained on the website `http://imm.dtu.dk/~pk/decisions`.

One can easily see that the component setup conforms to the binding definition in Fig. 8. We decided to keep the *Difference Binding* and the *Decision Binding* separate because the *Difference Binding* is independent of any design decision—it can be stored after applying model differences to a model and can also be exploited for other things, for instance, model synchronization.

In order to create the binding, we extended the algorithm for difference application in the component *Model Differencing*. The extension is straightforward: every time a change is made to the design model, e.g. an element was added or moved, corresponding *ElementBinding*s (cf. Fig. 8) are created.

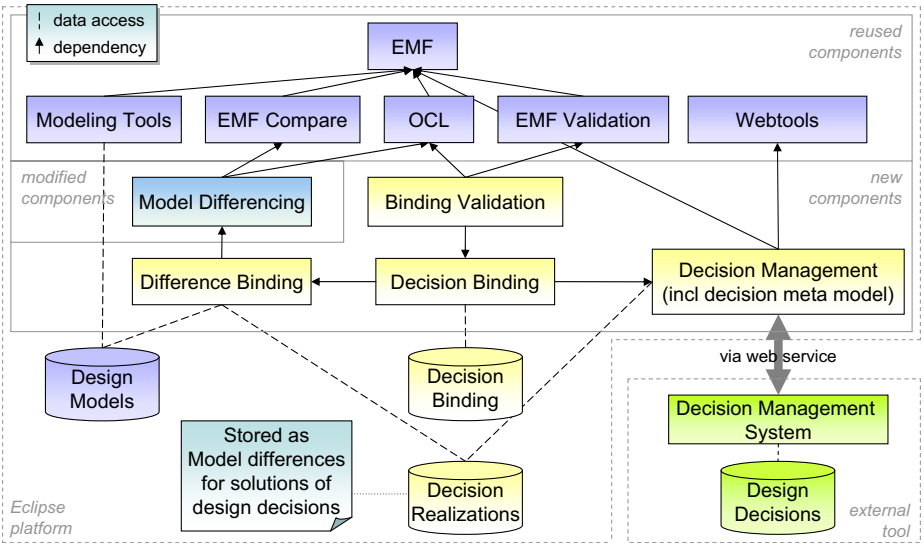

Fig. 11. The architecture of the prototype

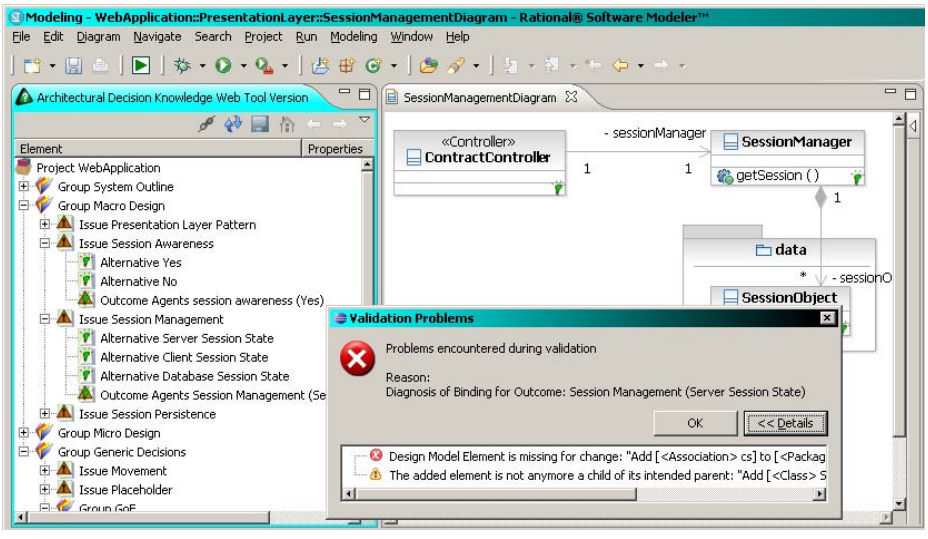

Fig. 12. Design Decision view and validation results

User Interface. This section sketches the realization of the user interface for the presentation of design decisions within Eclipse and for consistency checking.

The left-hand side of Fig. 12 shows all design decisions of the current project in the design decision view. Actions are available for browsing through design decisions, creating new, or modifying existing decisions.

The right-hand side of Fig. 12 shows a dialog as the result of a consistency check of the bindings between the design decisions shown on the left and the design model on the right. The dialog shows the same two violations from the example in Sect. 4.3 with their severity (error and warning) and a description. The affected design model elements are also marked in the graphical editor.

6 Related Work

There are many tools for documenting decisions and capturing architectural knowledge. None of the existing research prototypes and commercial tools provides the integration between design decisions and design models we motivated and specified in a previous publication [15] and in previous sections of this paper. Hence, our documentation goal has only been partially met so far; the consistency and reuse goals have not been addressed sufficiently yet.

Documentation Goal. There are several systems and approaches which support developers in capturing and making decisions during a software development process. ADDSS [5], for instance, is a web-based tool to collect and store architectural knowledge including, but not limited to, architectural decisions.

ADDSS supports after-the-fact decision capturing; the captured information can be studied retrospectively, for instance on a subsequent project phase or different project. However, ADDSS does not support a tight, use case-driven design model integration such as the one we introduced in the previous sections. For instance, it is not possible to create outcome instances via the modeling tool to record the rationale behind a design model change while or immediately after performing the change. To do so, it is required to switch to the decision management tool.

AREL [16] is another system for the documentation of architectural decisions based on their rationale. It specifies a UML profile for modeling architectural design decision rationale and traces them back to the architectural elements; a single tool can be used to work with UML design models and with design decisions. However, AREL does not allow the user to capture and reuse changes in the design models, and to synchronize this information with decision decisions on the fly; these two artifacts merely coexist in the tool.

The Architectural Decision Knowledge Web Tool (formerly known as Architectural Decision Knowledge Wiki) [11], which we extended in our prototype, allows architects to capture, store, and share design rationale. Its base version supports the user in making and reusing decisions but does not integrate design decisions with design models. This support is provided by the prototype described in Sect. 5.

Other tools [17,18] have similar characteristics as the ones discussed so far.

Consistency Goal. The consistency goal is not met by any of the existing research prototypes; ensuring consistency remains a manual task. In practice, informal, human-centric techniques such as coaching, architectural templates, and code reviews dominate. For instance, software engineering processes like RUP [19] advise architects to enforce decisions by refining the design in small and therefore actionable increments. The agile community emphasizes the importance of face-to-face communication and team empowerment [20]. Maturity models such as the Capability Maturity Model Integration[2] recommend rigid approaches to ensure that decision outcome materializes, e.g., formal reviews. Applying these techniques takes time and their success depends on the architects' coding and leadership skills.

We are not aware of any model-based software development tools that respect design decisions. OpenArchitectureWare[3] is a framework for model-driven development allowing the developer to define and use model transformations. However, architectural decisions are not a genuine modeling concept in OpenArchitectureWare. Modeling tools like the IBM Rational Software Modeler[4] and Borland Together[5] provide pattern authoring capabilities which are similar to the intention of the realizations of design decisions. However, a metamodel for expressing relations between them as well as tool supported guidance, i.e. proposing

[2] Available at: http://www.sei.cmu.edu/cmmi/
[3] Available at: http://www.openarchitectureware.org
[4] Available at: http://www.ibm.com/software/awdtools/modeler/swmodeler/
[5] Available at: http://www.borland.com/us/products/together/

subsequent patterns, is missing. Other commercial modeling tools allow the user to make simple decisions, for instance regarding model element naming, but use fixed defaults for architectural concerns, e.g. system transaction management boundaries [21]. Consequently, development resources have to be invested to change the defaults to the settings required in a particular application design and implementation.

Design Decision Reuse Goal. In the past, the design decision rationale and architectural knowledge communities have focused on documenting decisions that have already been made (following a retrospective, after-the-fact decision capturing approach). As a consequence, there is no notion of reusing knowledge about decisions required (i.e., issues and alternatives); few concepts exist for bringing required decisions into the original design process or into the model-driven development transformation chain. For instance, ADDSS and AREL do not support a reuse strategy which automatically updates the design models according to a decision made. In our previous work, we have developed a framework for architectural decision modeling with reuse which includes an explicit decision enforcement step [6]. The integration concepts introduced in this paper provide an advanced, partially automated form of decision enforcement for the framework.

7 Conclusion and Future Work

In this paper, we presented concepts for connecting design models in model-based software development with semi-formal design knowledge (design decisions) to automate tedious and error-prone, recurring work. The proposed concepts make use of existing technologies (decision management systems and model differences) and introduce a formal binding between design models, design decisions, and model differences. We defined three goals for our contributions: easier documentation is achieved by exploiting the binding and showing the information in an additional view; consistency checking is achieved by validating formal constraints on bindings; reuse of design decisions is partially automated by attaching design model changes to solutions of design issues.

The concepts are implemented in a prototype[6] and its technical feasibility is proven with a case study [7]. Decision reuse has been validated in our previous work [6,21]. Moreover, we evaluated reuse of model differences with all 23 design patterns from [4] and 25 refactorings from [22] (the other refactorings are not applicable to UML models): 8 design patterns and 14 refactorings are generically applicable right away. Although the other 15 design patterns are also applicable, they rather produce a draft which must be adjusted. The other 11 refactorings are not applicable generically because the current prototype only allows to reuse precisely those realizations which have been made before. In other words, if the design model does not contain the context specified in the model differences,

[6] Information about the prototype is available at `http://imm.dtu.dk/~pk/decisions`

that particular realization cannot be used. Work in progress is a generalization of model differences which aims to overcome this problem. Moreover, we demonstrated the prototype to leading software architects and developers of a commercial modeling platform. An evaluation on a real project is in preparation.

Future work includes to improve the presentation of the consistency check results and to exploit causal relations between design decisions to propose subsequent decisions—e.g. via the relation *induces* in Fig. 1.

References

1. Object Management Group: UML Superstructure, V2.2 (November 2007)
2. Object Management Group: MDA Guide V1.0.1 (June 2003)
3. Nowak, M., Pautasso, C., Zimmermann, O.: Architectural Decision Modeling with Reuse: Challenges and Opportunities. In: 5th SHARK, South Africa (May 2010)
4. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (January 1995)
5. Capilla, R., Nava, F., Duenas, J.C.: Modeling and Documenting the Evolution of Architectural Design Decisions. In: 2nd SHARK-ADI, Minneapolis, USA, pp. 9–15. IEEE Computer Society, Los Alamitos (May 2007)
6. Zimmermann, O.: An Architectural Decision Modeling Framework for Service-Oriented Architecture Design. Dissertation, University of Stuttgart (2009)
7. Könemann, P.: Integrating a Design Decision Management System with a UML Modeling Tool. IMM-Technical Report-2009-07, Technical University of Denmark (April 2009)
8. Object Management Group: MOF Core Specification, Version 2.0 (January 2006)
9. Fowler, M.: Patterns of Enterprise Application Architecture. Addison Wesley, Reading (November 2002)
10. Shahin, M., Liang, P., Khayyambashi, M.R.: Architectural Design Decision: Existing Models and Tools. In: WICSA/ECSA Working Session. IEEE Computer Society, Los Alamitos (September 2009)
11. Zimmermann, O., Gschwind, T., Küster, J.M., Leymann, F., Schuster, N.: Reusable Architectural Decision Models for Enterprise Application Development. In: Overhage, S., Szyperski, C., Reussner, R., Stafford, J.A. (eds.) QoSA 2007. LNCS, vol. 4880, pp. 15–32. Springer, Heidelberg (2008)
12. Förtsch, S., Westfechtel, B.: Differencing and Merging of Software Diagrams–State of the Art and Challenges. In: ICSOFT, Setubal, Portugal, pp. 90–99 (July 2007)
13. Könemann, P.: Model-independent Differences. In: ICSE Workshop on Comparison and Versioning of Software Models, pp. 37–42. IEEE Computer Society, Los Alamitos (May 2009)
14. Object Management Group: OCL Specification, Version 2.0 (May 2006)
15. Könemann, P.: Integrating Decision Management with UML Modeling Concepts and Tools. In: WICSA/ECSA Working Session. IEEE Computer Society, Los Alamitos (September 2009)
16. Tang, A., Jin, Y., Han, J.: A Rationale-based Architecture Model for Design Traceability and Reasoning. Journal of Systems and Software 80(6), 918–934 (2007)
17. Bachmann, F., Merson, P.: Experience Using the Web-Based Tool Wiki for Architecture Documentation. Technical Report CMU/SEI-2005-TN-041, Carnegie Mellon University, Software Engineering Institute (September 2005)

18. Liang, P., Jansen, A., Avgeriou, P.: Knowledge Architect: A Tool Suite for Managing Software Architecture Knowledge. Technical Report RUG-SEARCH-09-L01, University of Groningen (February 2009)
19. Kruchten, P.: The Rational Unified Process: An Introduction. Addison-Wesley, Reading (2003)
20. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley, Reading (1999)
21. Zimmermann, O., Grundler, J., Tai, S., Leymann, F.: Architectural Decisions and Patterns for Transactional Workflows in SOA. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 81–93. Springer, Heidelberg (2007)
22. Fowler, M.: Refactoring: Improving the Design of Existing Code. In: Object Technology Series. Addison-Wesley, Reading (June 1999)