Automatic Enforcement of Constraints in Real-time Collaborative Architectural Decision Making

Patrick Gaubatz*, Ioanna Lytra, Uwe Zdun

Faculty of Computer Science, University of Vienna, Austria

Abstract

Making and documenting architectural design decisions becomes increasingly important in the process of software architecting. However, the remoteness of different decision stakeholders, ranging from local distribution in an office environment to globally distributed teams, as well as the different domain knowledge, expertise and responsibilities of the stakeholders hinder effective and efficient collaboration. Existing tools and methods for collaborative architectural decision making focus mainly on sharing and reusing of knowledge, making trade-offs, and achieving consensus, but do not consider the various stakeholders' decision making constraints due to their roles in the development process. To address this problem, we propose a meta-model for a set of decision making constraints, with precisely defined semantics, as well as a collaborative architectural decision making approach based on this meta-model. We also present tool support, called CoCoADvISE, which automatically enforces the constraints at runtime. The evaluation of this tool in a controlled experiment with 48 participants shows that our approach, besides preventing constraint violations, significantly increases both the time and effort related efficiency, as well as the effectiveness of users in collaborative decision making.

Keywords: Decision Making Constraint, Constraint Enforcement, Collaborative Decision Making, Architectural Decision Making, Reusable Architectural Decision Model, Controlled Experiment

1. Introduction

The trend of globally distributed projects in software and IT industries makes collaboration and coordination within dispersed teams challenging [11]. Unlike co-located project teams, geographically distributed partners need to overcome collaboration problems caused by the distance, different concerns of

*Corresponding author: Patrick Gaubatz, Faculty of Computer Science, University of Vienna, Währingerstraße 29, 1090 Vienna, Austria; Phone, +43-1-4277-785 20

Email addresses: patrick.gaubatz@univie.ac.at (Patrick Gaubatz), ioanna.lytra@univie.ac.at (Ioanna Lytra), uwe.zdun@univie.ac.at (Uwe Zdun)

stakeholders, and different development processes. While these challenges are particularly problematic in distributed project settings, even in a locally distributed environment, like offices on multiple different floors, efficient and effective collaboration is an issue. Software architecture can be seen as a tool for coordination in distributed software development, as a common understanding and agreement on issues at the architectural abstraction level can prevent coordination problems in later phases [34]. As architectural decisions have become a primary means for describing software architecture in recent years, the collaboration in architectural decision making and documenting should be supported. Farenhorst et al. point out the need of explicitly supporting collaboration between architects with appropriate tools and consider this aspect to be one of the five most important characteristics of software architecting [7].

Only a few of the existing tools for architectural decision management address collaboration in architectural decision making and documentation. Approaches that target collaborative architectural decision support (see, e.g., [31, 39, 53, 41, 20, 2, 4]) mainly focus on team building, achieving consensus, making of trade-offs, and sharing of architectural knowledge. However, none of these approaches considers the various stakeholders' decision making constraints due to their roles in the development process. This is despite the fact that such roles (see, e.g., [30, 45, 6]), their potentially diverging rights and duties and their potentially conflicting objectives (see, e.g., [36, 29, 34]) within a possibly constrained (see, e.g., [3, 18, 14]) decision making process actually do exist. As an exemplary decision making constraint, consider that a stakeholder with the role *Integration Architect* must be in agreement with a stakeholder with the role *Application Architect* before an architectural decision that concerns technical development aspects can be finalized.

In this paper, we therefore present a novel approach for augmenting reusable architectural decision models with such decision making constraints. Reusable decision models provide (similar to design patterns [10]) proven solutions – both application-generic and technology-specific – to various design issues along with their forces, consequences, and alternative solutions (see, e.g., [66, 67, 68, 65, 21]). We show how CoCoADvISE (Constrainable Collaborative Architectural Design Decision Support Framework), our prototype implementation, automatically enforces these decision making constraints at runtime. As an example, we describe how our approach can be applied in an industrial context, i.e., in the context of service-based platform integration.

So far there are only a few empirical studies on architectural decision making (see, e.g., [42, 59, 58]) in general or on the specific aspect of group decision making (see, e.g., [31, 37, 38]). As our work mainly deals with propositions about the efficiency and effectiveness of supporting automatic enforcement of constraints for humans, we decided to evaluate it using a controlled experiment with 48 participants. Our experiment provides evidence that automatic enforcement of decision making constraints significantly increases both the time and effort related efficiency and effectiveness of users of decision making tools.

The contributions of this paper are as follows:

- We propose automatic enforcement of decision making constraints, a new aspect to be considered in collaborative architectural decision making.
- We present CoCoADvISE, a constrainable collaborative decision making tool supporting automatic enforcement of constraints.
- We precisely specify a set of decision making constraints using first-order logic based on a formal meta-model (described in the appendix).
- We provide empirical evidence of the time and effort related efficiency and effectiveness of supporting automatic enforcement of constraints.

The remainder of the paper is structured as follows. We give an overview of the existing architectural decision management tools and discuss collaborative aspects and their challenges in Section 2. Section 3 motivates our work and provides a motivating example. In Section 4 we present our collaborative architectural decision making tool CoCoADvISE. Section 5 exemplifies the application of our approach in the context of service-based platform integration, which was investigated in the context of the EU research project INDENICA. The following Section 6 and 7 describe our experimental setting and the analysis of the results of the controlled experiment respectively. Section 8 discusses our findings, implications, as well as threats to validity, and Section 9 concludes.

2. Related Work

2.1. Architectural Design Decisions

Software architecture is seen more and more as a set of architectural decisions [13]. Capturing architectural design decisions is important for analyzing, understanding, and sharing the rationale and implications of these decisions and reducing the problem of architectural knowledge vaporization [10].

Approaches for capturing architectural decisions, using either templates [56], ontologies [19], or meta-models [66], concentrate on the reasoning on software architectures, capturing and reusing of architectural knowledge, as well as sharing and communicating of design decisions between stakeholders. In addition, patterns are regarded as proven knowledge for capturing architectural decisions and their rationale [10] and are considered often in the aforementioned approaches.

A substantial amount of work has been done in the direction of documenting architectural knowledge using architectural decision modeling (refer to [40] for a comparison of existing architectural decision models and tools). For instance, Jansen and Bosch propose a meta-model to capture decisions that consist of problems, solutions and their attributes [13]. Zimmermann et al.'s meta-model for capturing architectural decisions [69] consists of *Architectural Decisions (AD)* related to one or more *ADTopics* organized in *ADLevels*, entailing *ADAlternatives*, the selection of which leads to an *ADOutcome*. The advantage of such decision models is that they are reusable and can be used as guidance for architectural decision making activities, whenever recurring design issues emerge. Various reusable architectural decision models have been

documented in the literature, covering Service-oriented Architecture related solutions [67, 68], service-based platform integration [21], the design of domain specific languages [64], and model and metadata repositories [26]. To motivate and demonstrate our proposal we use such reusable architectural decision models as basis for decision making that involves different stakeholders.

2.2. Collaboration in Architectural Decision Making

Whereas several tools have been developed to ease capturing, managing and sharing of architectural decisions [40], only a few target explicitly the collaboration needs of distributed teams, i.e., when making architectural decisions in a group. The sharing of architectural knowledge is one of the main concerns of the architectural decision management tools AD_{kwik} [39], Knowledge Architect [20], and PAKME [2]. The collaboration in all cases is achieved by providing central repositories containing design pattern catalogs, documented architectural decisions, use case scenarios, and so on, accessible to all co-located or distributed software team members, without any automated support regarding the required collaborative work. In addition, Compendium [41] supports a visual environment for documenting and visualizing design rationale behind architectural design decisions for multiple users. In some cases, also, Wiki-based tools are proposed [4] to assist architectural knowledge management performed at geographically separate locations. However, collaborative work is not the main focus of these tools, and thus, the challenges of making and documenting architectural decisions collaboratively are not studied in aforementioned works.

Other recent proposals, such as Software Architecture Warehouse [31] and GADGeT [53], target the consensus making, the communication, and the progress tracking for group architectural decisions. Proposing, ranking, and voting for alternatives are main concepts that are integrated in the aforementioned tools. However, decision making constraints caused by different stakeholder roles, company policies or processes which may cause inconsistencies and additional effort are not considered. In our proposal, we extend the concepts of collaborative architectural decision making by introducing the automatic enforcement of such constraints during group decision making. Automatic support is thus an advantage of CoCoADvISE in comparison to the aforementioned tools, which do not target any automation in collaborative decision making.

Recent literature surveys and reviews that compare approaches and tools for architectural decision making and documentation (e.g., [40, 55, 38]) consider collaboration support as an important feature of these tools. In addition, according to a survey with 43 architects from industry conducted by Tofan et al., most of the architectural decisions are group decisions (86%) [54]. This finding is also validated in a different study by Miesbauer and Weinreich [27]. However, little empirical evidence – especially quantitative results – exists with focus on collaborative decision making by practitioners. Nowak and Pautasso have collected feedback from more than 50 focus groups of students regarding the usability and situational awareness support of their tool Software Architecture Warehouse [31]. Rekha et al. performed an exploratory study to investigate how

architectural design decisions are made in a group, what information is documented, and which tools are used by practitioners in the industry [37]. In our work, we target collaborative architectural decision making involving different stakeholder roles and various constraints.

2.3. Constraint Enforcement in the Context of Security and Access Control

Although constraint enforcement has not been considered in the context of architectural decision making yet, it is actually a quite well-studied topic in other contexts, such as security and access control. Especially access control in the context of business processes and workflows (see, e.g., [62, 15, 49]) introduces the notion of assigning (stakeholder) roles to each task in a process, which is similar to our notion of assigning a certain role to be responsible for a specific architectural decision. At runtime, each task/decision can only be performed/made by a user that owns the required role.

Task-based entailment constraints (see, e.g., [48, 50, 63]), which also originate from the context of workflows can, for instance, be used to enforce the four-eyes principle or other separation of duties constraints. In principle, such separation of duties constraints are similar in definition, checking, and enforcement to decision making constraints that require several stakeholders or roles to unanimously agree on a concrete solution for a specific architectural decision (which are, for instance defined in our approach). Unfortunately, to the best of our knowledge, there are no works that provide empirical evidence about the positive effects of constraint enforcement in these contexts.

2.4. Constraint Enforcement in the Context of Collaboration

In recent years, there has been a movement to embrace and facilitate collaboration in various software engineering tools. For example, numerous real-time collaborative Web-based Integrated Development Environments, such as Cloud9[1], Koding[2], Adinda [57] or Collabode [9] have been proposed. In addition, real-time collaborative Web-based modeling tools, such as Creatly[3] or Lucidchart[4], have also been proposed. Finally, more specialized software engineering tools, such as the collaborative Web-based software architecture evaluation tool presented by Maheshwari et al. [24], have emerged.

Similarly to these tools, our CoCoADvISE tool does not require its users to install or configure any software locally on their computers, which is one of the main benefits of Web applications. However, the aforementioned tools – unlike CoCoADvISE – consider different stakeholder roles, permissions and constraints. As the adoption of such tools in industrial contexts is likely to rise, this situation will probably change in the future [60]. Also, to the best of our knowledge, no comparable empirical studies on these tools and their underlying concepts have been performed.

[1] `http://c9.io`
[2] `http://koding.com`
[3] `http://creately.com`
[4] `http://lucidchart.com`

3. Motivation

3.1. Constrained Architectural Decision Making

Our approach provides tool support for architectural decision making constraints. Such constraints and the relationships and responsibilities of stakeholders that are formally expressed in those constraints are frequently discussed both in academic and industrial contexts. To motivate our approach, we summarize in this section some evidence from the literature.

Nord et al. present a structured approach for reviewing architecture documentation [30]. They provide an illustrative list of common stakeholder roles and their concerns in the decision making process. In particular, they also document the existence of potentially diverging rights and duties withing the decision making process, such as: "*For example, in safety-critical systems, the safety analyst is one of the most important stakeholders, as this person can often halt development on the spot if he or she believes the design has let a fault slip through.*". A study by Smolander et al. reveals and analyzes more than 20 stakeholder roles participating in architecture design in three different software companies, and concludes that further research on practices and tools for effective communication and collaboration among the varying stakeholders of the architecture design process are needed [45]. Eckstein documents experiences of a lead software architect on global agile projects [6]. This lead software architect propagates orchestration of the architect roles, i.e., having one lead architect and several subordinate architects, partitioned according to boundaries of subsystems, problem domains and sites. In the context of Enterprise Architecture (EA), an orthogonal study by van der Raadt et al. concludes that stakeholders pursue different, potentially conflicting, objectives, related to their specific role within the organization [36]. They also note that efficient collaboration between stakeholders is one of the main critical success factors for EA. Nakakawa et al. conducted a survey on effective execution of collaborative tasks during EA creation and highlighted – among others – the following challenges, reported by 70 enterprise architects [29]: (1) it is hard to reach consensus due to conflicting stakeholders' interests, (2) organization politics result in fuzzy decision making, (3) stakeholders are not accountable for their decisions, (4) lack of a clear decision making unit, (5) lack of a governance process that can ensure architecture compliance, (6) lack of supporting tools and techniques for executing collaborative tasks. Similar results have been reported by Päivi Ovaska et al. In [34] they conducted a case study in an international ICT company, analyzing coordination challenges in multi-site environment with geographically dispersed development teams. Most notably, they observed problems in coordination, such as lack of overall responsibility (i.e., unclear decision making) in architecture design.

In recent years, software development governance has been recognized as a key component for steering software development projects. In general, governance is about deciding the "*who, what, when, why, and how*" of (collaborative) decision making (see, e.g, [3, 5, 16]). Kofman et al. frame the ideal software development governance environment [18]: "*Every member of a team would know at any given point of time, what needs to be done, who is responsible for which*

task, and how to perform the tasks for which he or she is responsible". In the course of developing the IBM Software Development Governor prototype they also observed and documented the need for decision making constraints. That is, they mention decision making policies, such as framing the boundaries of the decision (i.e., who should participate, and when), or specifying how the decision is to be made (i.e., consensus or voting). Jensen et al. noted that "*there are many issues critical to governing software development, including decision rights, responsibilities, roles, accountability, policies and guidelines, and processes*" [14]. Interestingly, by studying NetBeans, an open source software project, they could actually identify similarities to these aforementioned governance issues. In summary, architectural decisions are key decisions, that can – in certain contexts – be subject to software development governance including the entailing decision making policies.

3.2. Motivating Example

To illustrate the concept of collaborative architectural decision making and to motivate the need for automatic enforcement of decision making constraints, let us consider a fictive online retailer company that wants to provide third-party online stores with a Web service for purchasing books and electronic gadgets. Based on functional requirements, as well as non-functional requirements, including requirements concerning secure and encrypted transactions, the company has to decide on what kind of Web service they are eventually going to deploy. For decisions that have such far-reaching implications throughout the system, company policies require that the involved software development teams, which are spread across several, geographically distributed offices, collaboratively participate in the decision making process.

In particular, the following decision making constraints can be derived from the company policies: First, all decision stakeholders with the role *Integration Architect* shall *unanimously* decide on the type of Web service to be exposed. *Application Architects* shall decide on technical details like a concrete transport protocol, and *Integration Architects* have to confirm and approve these decisions eventually. *Security Experts* shall propose a solution for the security and encryption related decisions. Finally, representatives from selected third-party companies shall be allowed to participate in the decision making process. However, their votes can be overruled at any time by internal stakeholders of the retailer company.

The company might now rely on guidelines that tell each stakeholder to exactly know their roles and duties within this decision making process and to stay compliant to these constraints in any decision process they participate in. However, with a growing number of stakeholders, stakeholder roles, responsibilities, duties and other forms of constraints, it becomes nearly impossible for a single participant to not (unintentionally) violate some of these policies (as will be shown in our controlled experiment in Section 6). Hence, we propose (in Section 4) a collaborative architectural decision making tool which automatically enforces such constraints.

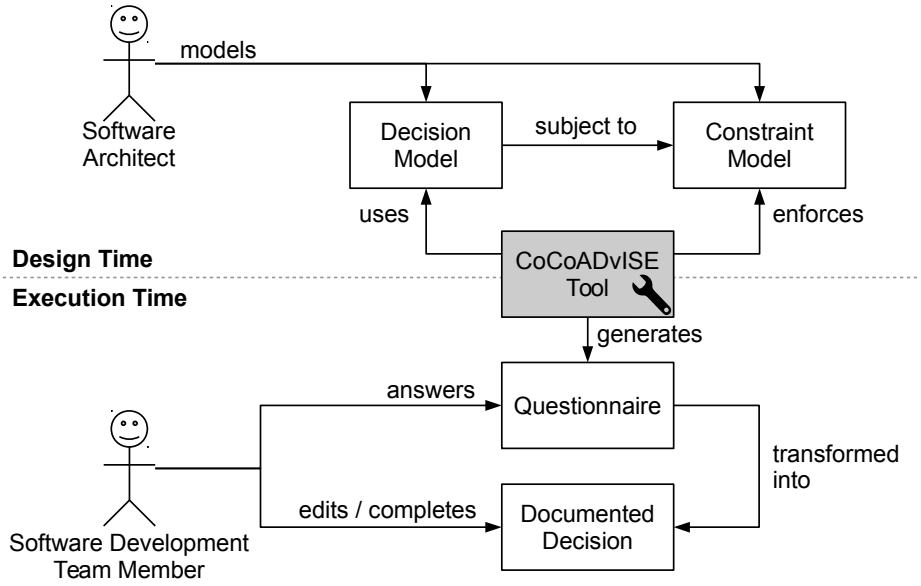


Figure 1: Overview of CoCoADvISE

4. CoCoADvISE Approach

CoCoADvISE[5] is a Web-based, collaborative tool that provides automated support for architectural decision making and documentation based on reusable decision models, as well as automatic enforcement of decision making constraints. It is based on previously presented work. More precisely, the tool is built upon the foundations of ADvISE (Architectural Design Decision Support Framework) [22] and CoCoForm (Constrainable Collaborative Form) [8], a real-time collaborative Web application framework. The main concepts of our approach are shown in Figure 1.

CoCoADvISE follows the reusable decision model approach (discussed already in Section 2.1) in which the documented reusable decisions can be instantiated as concrete decisions and thus used as guidance for architectural decision making activities, whenever recurring design issues emerge. The advantage of this approach is that the decisions must be created only once for a recurring design situation. In similar application contexts, a single decision model is reused multiple times for making multiple concrete decisions. A decision model can be (re-)used by transforming it into interactive questionnaires using model-driven techniques (see Section 4.1). Based on the outcomes of the questionnaires answered by Software Development Team Members, CoCoADvISE can automatically resolve potential constraints and dependencies (e.g., reveal follow-

[5]A demo of CoCoADvISE is available at: `https://piri.swa.univie.ac.at/cocoadvise`. Use the following user names to log in: experiment1 or experiment2 (no password required).

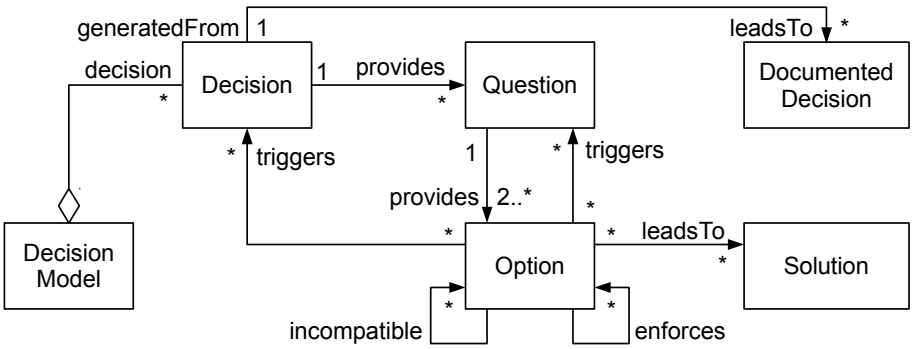


Figure 2: Conceptual Overview of CoCoADvISE's Decision Meta-model

on questions and decisions, deactivate options), recommend best-fitting design solutions, and eventually generate semi-complete architectural design decision (ADD) documentations (see Section 4.1).

The real-time collaboration features of CoCoADvISE enable multiple, possibly geographically dispersed software architects and stakeholders (i.e., Software Development Team Members) to participate in the group decision making and documentation process. In order to be applicable in industrial contexts, such as intra- and cross-organizational businesses, CoCoADvISE provides means for making this decision making process subject to constraints. It employs a model-driven approach in which reusable decision models are made subject to constraints, as can be seen in Figure 1 (see Section 4.2 and 4.3 for details).

The CoCoADvISE approach requires Software Architects to define abstract and reusable decision models at design time. If the decision making process has to be made subject to constraints, the Software Architects are supposed to formalize these requirements by defining constraint models and relating them to the corresponding decision models. Note that reusable decision models and constraints can only be defined and created using the ADvISE tool [22] while CoCoADvISE provides means for actually using these models.

4.1. Supporting Decision Making and Documentation

CoCoADvISE relies on reusable architectural decision models based on Questions, Options, and Criteria [23] that can be reused many times by the software architects in form of questionnaires, in order to guide architectural decision making. Such reusable architectural decision models are defined for recurring design situations, both domain and technology dependent and independent.

In particular, software architects define architectural decision models by creating instances of CoCoADvISE's decision meta-model at design time. While a detailed and formal representation of this meta-model can be found in Appendix A, Figure 2 provides a conceptual and graphical overview that may be more suitable for quickly grasping the core concepts and abstractions. Note that Figure 2, as well as the following Figure 3, 6, 7 and 8 represent UML2.2

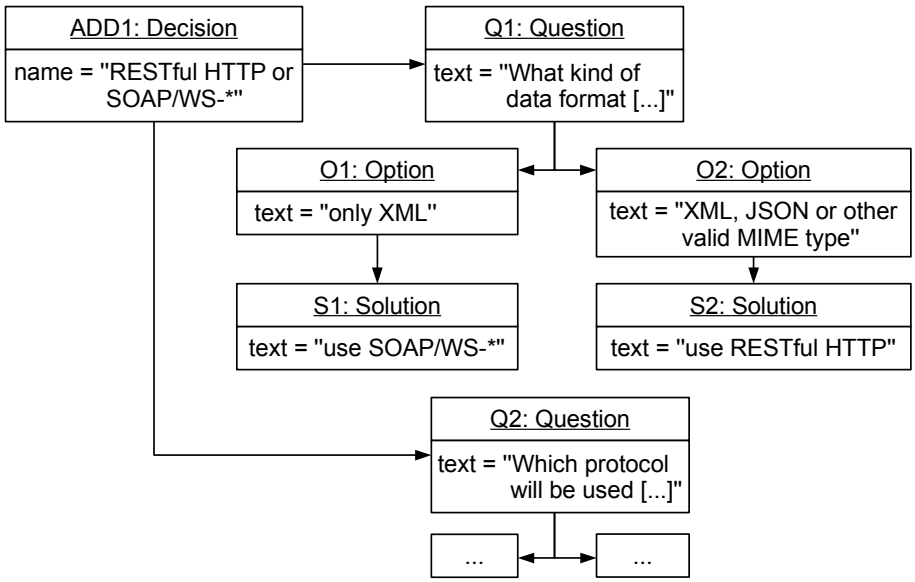


Figure 3: Exemplary Decision Model

Class and Object Diagrams with a minor syntactical deviation from the corresponding standard [33]. More precisely, we symbolize a relation's navigability with a filled arrowhead (i.e., ➙) instead of an open arrowhead.

As we can see, an architectural Decision Model consists of Decisions, Questions, Options, Solutions and relationships among them. Note that the usage of sans serif font indicates a reference to a class, e.g., Decision Model in Figure 2. These relationships allow for expressing consistency constraints and prescribing control flows like:

- If users choose Option x they must also choose Option y (enforces ➙).
- If users choose Option x they must not choose Option y (incompatible ➙).
- If users choose Option x they must also answer Question y (triggers ➙).
- If users choose Option x they must also decide on Decision y (triggers ➙).

The object diagram in Figure 3 depicts an excerpt of a very simplistic decision model that might be helpful in a situation where software architects have to decide on the kind of Web service to be deployed (as has been suggested in the motivating example from Section 3.2). This exemplary model consists of a single decision (i.e., ADD1) which has exactly one question (i.e., Q1). The question deals with the Web service's payload data format and can be answered by one of two options (i.e., O1 and O2). Finally, we can see, that each option leads to different solutions (i.e., S1 for O1 and S2 for O2). In general, Decisions

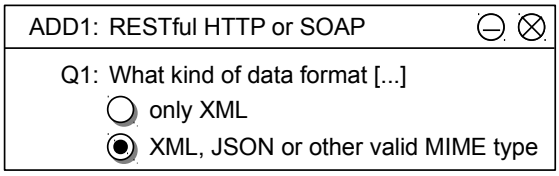


Figure 4: Exemplary Decision Questionnaire

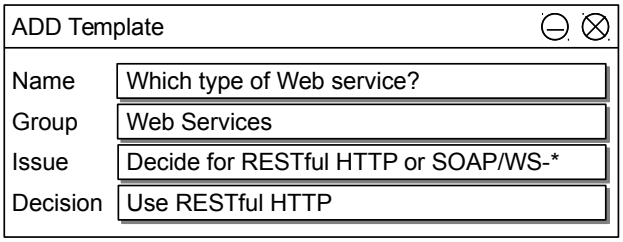
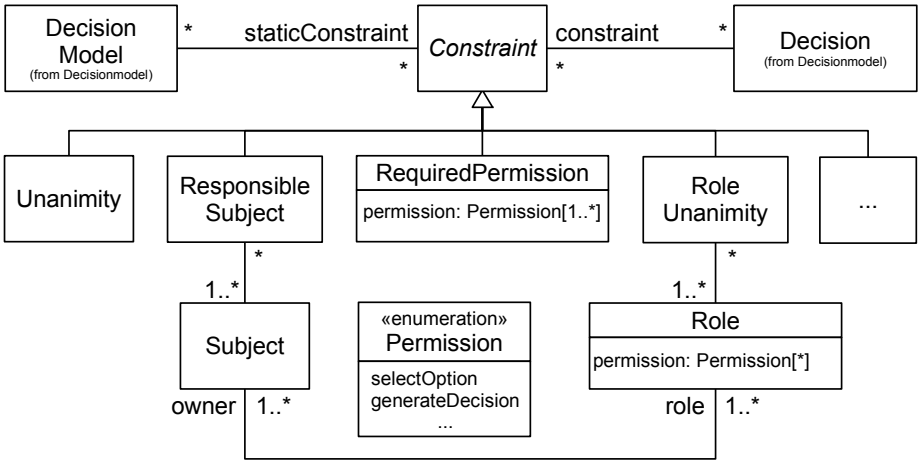


Figure 5: Exemplary Decision Template

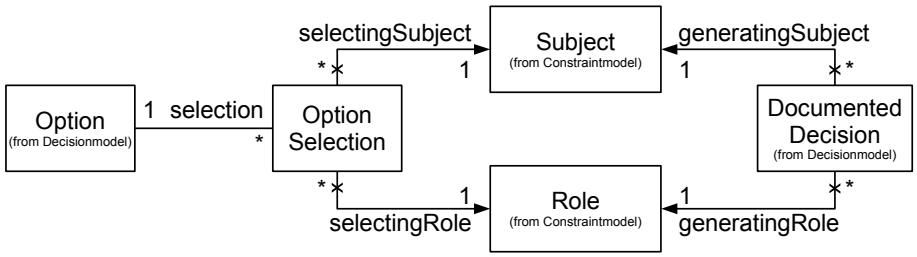
capture the essence of reusable architectural design decisions. Questions always belong to a particular superordinate Decision and are supposed to guide the software architect towards finding Solutions for the respective Decision. Finding Solutions involves choosing Options, thereby answering Questions.

In similar application contexts, software architects can reuse suitable architectural decision models. More precisely, reusing a decision model means instantiating the decision model and generating questionnaires which are used at execution time of the collaborative Web application by software development team members. As indicated in Figure 4 a questionnaire consists of questions and each question has to be answered by choosing exactly one option from a set of possible options. Thus, by answering these questionnaires, the best-fitting design solutions are recommended to the software development team members. CoCoADvISE takes the responsibility of verifying and guaranteeing the consistency of the decision model instances by automatically revealing follow-on questions and decisions, hiding, showing, enabling and/or disabling specific parts of the questionnaires in a way that users simply can not leave the questionnaire in an inconsistent state. For instance, whenever a user selects a specific option, the system automatically disables all other incompatible options.

Eventually, the selected options and gathered solutions can automatically be transformed into semi-completed architectural decision documentation templates similar to the one proposed by Tyree and Akerman [56]. For instance, if we consider the exemplary questionnaire depicted in Figure 4 and assume that a user has selected Option 2 (i.e., "XML, JSON or other valid MIME type"), we can transform this particular questionnaire into a decision documentation template (i.e., Documented Decision in Figure 1) like the one depicted in Figure 5. As we can see, many fields of these templates, such as "Name", "Group", "Issue"



(a) Constraint Model Extension



(b) Runtime Model Extension

Figure 6: Conceptual Overview of CoCoADvISE's Constrainable Decision Meta-model

and "Decision" can be completed (semi-)automatically by combining information that is encoded in the decision model (see, e.g., Figure 3) with user-provided input gathered via questionnaires (see, e.g., Figure 4).

4.2. Definition and Enforcement of Decision Making Constraints

In CoCoADvISE, reusable decision models are augmented with additional constraint elements. This section gives an overview of the generic meta-model for the specification of decision making constraints for reusable architectural decision models and details how these constraints are enforced at runtime.

The class diagrams depicted in Figure 6 provide a conceptual overview of the essential concepts of a Constrainable Decision Meta-model including Subjects, Roles, Permissions, and various types of Constraints. Technically, the Constrainable Decision Meta-model extends the previously presented Decision Meta-model (see Figure 2 and Section 4.1) with additional elements, relevant at design time (i.e., Figure 6(a)) and execution time (i.e., Figure 6(b)).

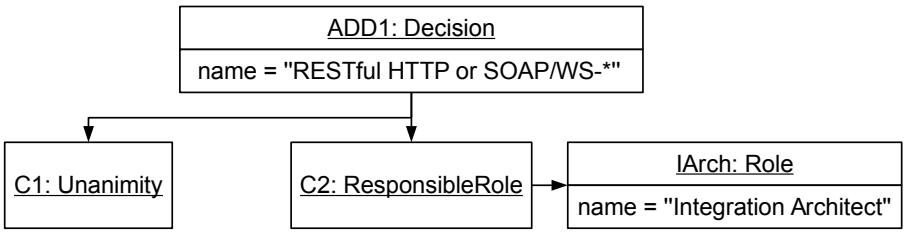


Figure 7: Exemplary Constraint Model

In our approach, Roles are used to model different job positions and scopes of duty within an organization. These Roles are equipped with the Permissions to perform certain tasks. Human users (i.e., Subjects) are assigned to Roles according to their work profile (see, e.g., [47]). This relation is a many-to-many relation, which means that a Subject can be assigned to numerous Roles and each Role can be owned by numerous Subjects.

CoCoADvISE provides two separate mechanisms for defining decision making constraints. First of all, we can make particular reusable architectural decisions subject to decision making constraints by assigning Constraints to Decisions. In addition, all reusable architectural decisions of a decision model can be made subject to decision making constraints at once by assigning the corresponding Constraints to the Decision Model, instead of a single decision. These two mechanisms are inherited by all subclasses of the abstract Constraint class (e.g., Unanimity or Responsible Subject).

As an example, let us revisit the motivating example from Section 3.2. Figure 7 depicts how the first set of decision making constraints can be implemented in CoCoADvISE. More precisely, we augment the previously defined exemplary decision model (see Figure 3) by attaching decision making constraints. By defining both a Responsible Role and a Unanimity constraint we can formalize the requirement that stakeholders with the role *Integration Architect* have to *unanimously* decide on the type of Web service. Note that we have to parametrize the Responsible Role constraint by assigning the Role object, which represents the *Integration Architect* role, to it. On the contrary, a Unanimity constraint can and does not need any further parametrization. According to Figure 6(a) the three additional classes that are used for parametrization of some constraints are: Subject, Role, and Permission. A Subject represents a person, i.e., a user/stakeholder of the system (see, e.g., [49]). Roles are assigned to subjects, and through their roles the subjects are granted Permissions. A Permission models a right to perform a certain action within the system. In CoCoADvISE we mainly focus on constraining two fundamental actions, i.e., selecting options and generating decisions. Figure 6 reflects this circumstance by explicitly listing the corresponding permissions selectOption and generateDecision. Other permissions could be added to the tool as extensions.

Figure 6(b) shows how we extend our original decision meta-model of Figure 2 with additional runtime-specific elements and relations. For the purpose of

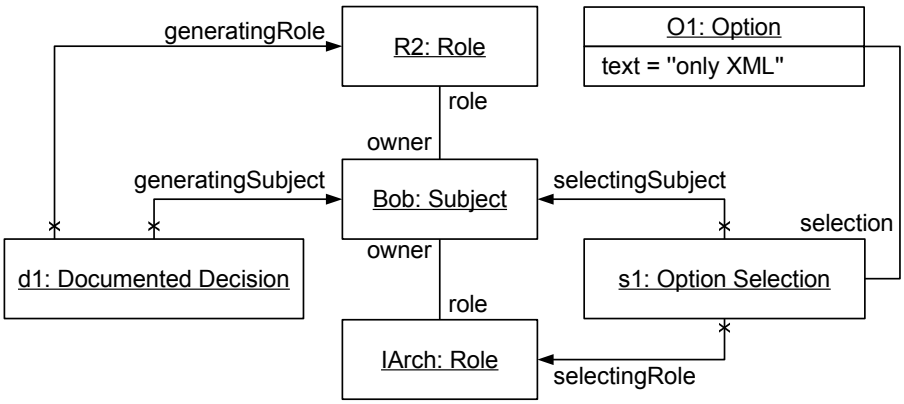


Figure 8: Exemplary Constraint Runtime Model

enforcing certain decision making constraints we have to introduce – among others – the concept of an Option Selection. At execution time, the existence of an Option Selection object means that the corresponding Option has been selected by a user (i.e., executing a selectOption action). More precisely, the relations selectingSubject and selectingRole of this Option Selection object are supposed to point to the user's corresponding Subject respective Role objects. Analogously, the generatingSubject and generatingRole relations of a Documented Decision object are supposed to refer to the Subject and Role objects of the user who has generated a decision (i.e., executing a generateDecision action). Figure 8 exemplifies these runtime-specific elements and relations. In this particular example, there is one Subject (i.e., Bob), that owns two different Roles (i.e., IArch and R2). Assuming, that Bob selects Option O1 at runtime, Option Selection s1 is newly instantiated. In addition, the selectingSubject and selectingRole relations of s1 are set to Bob and IArch (hereby assuming that IArch is the Role that Bob is currently using). In summary, s1 captures the fact that Bob has selected O1 using the Role IArch. Later, Bob generates a Documented Decision d1 using another Role, i.e., R2. As a result, the generatingSubject and generatingRole relations of d1 are set to Bob and R2 to capture the Subject and Role responsible for this event.

4.3. Logical Specification of Decision Making Constraints

In this section, we present the precise definition of the decision making constraints using first-order logic as a formalism that is abstract and technology-independent but can still easily be mapped to different existing constraint languages used in modeling and software development, such as OCL[6], the Check

[6] `http://www.omg.org/spec/OCL/`

language of the Eclipse M2T project[7], or Frag's FCL constraints[8]. In order to enable this precise definition, we needed to map the meta-models from Figures 2 and 6 to first-order logic as well. For the sake of completeness we provide this formalization in Appendix A. The reader can refer to Appendix A for a complete reference of the semantics of the constraint meta-model but we note that this is not necessary for understanding the constraint types and their application. In particular, Definition A.1 provides a list of elements and their relations, Definition A.2 presents crucial model invariants to be considered at design time, and Definition A.3 lists model invariants relevant at execution time.

The following list of Constraint Type (CT) Definitions constitutes our proposed set of decision making constraints. Each Constraint Type Definition consists of a narrative description, a formal definition of model invariants, expressed using first-order logic (see, e.g., [46]), as well as an exemplary application of the respective constraint type. A constrainable decision model that fulfills the invariants of a Constraint Type Definition CT_n is said to be compliant with CT_n. Note that this list is *not* an exhaustive list of all possible constraint types. Instead, we selected a set of constraint types that we believe to be representative and common in real-life scenarios.

The following three Constraint Types (CT_{1-3}) belong to the family of **Unanimity** constraints. Such constraints can be used to enforce consensus finding among a particular set of stakeholders. As has been mentioned in Section 4.2, CoCoADvISE focuses on constraining the process of selecting options and generating decisions. In connection with this, Unanimity constraints concern the option selection process.

CT_1 **(General) Unanimity:** Each question of a decision that is subject to a (general) unanimity constraint has to be answered unanimously. More precisely, a question is said to be answered unanimously if all subjects have agreed on the concrete set of options to select. This is the case when all subjects (i.e., "users") have selected the same set of options.

In order to be able to express this model invariant, we need to define the following required mappings first:

$cd(c) = \{d_1, ..., d_n\}$	Set of decisions $\{d_1, ..., d_n\}$ that is constrained by constraint c (see Definition A.1.19)
$qda(d) = \{q_1, ..., q_n\}$	Set of questions $\{q_1, ..., q_n\}$ that belong to decision d (see Definition A.1.2)
$oqa(q) = \{o_1, ..., o_n\}$	Set of options $\{o_1, ..., o_n\}$ that belong to question q (see Definition A.1.3)
$soa(o) = \{os_1, ..., os_n\}$	Set of (option) selections $\{os_1, ..., os_n\}$ that belong to option o (see Definition A.1.9)

In addition, the following set needs to be defined (see Definition A.1):

[7] `https://www.eclipse.org/modeling/m2t/`
[8] `http://frag.sourceforge.net/`

C_U	An element of C_U is called *Unanimity Constraint*
D	An element of D is called *Decision*
Q	An element of Q is called *Question*
O	An element of O is called *Option*
O_S	An element of O_S is called *Option Selection*

Therefore:

For each unanimity constrained question (q), if one option (o_x) has been selected ($soa(o_x) \neq \emptyset$), all other options ($o_y \neq o_x$) must not be selected ($soa(o_y) = \emptyset$).

$$\forall c \in C_U(\forall d \in cd(c)(\forall q \in qda(d)(\forall o_x \in oqa(q)(\forall o_y \in oqa(q)(\\ o_x \neq o_y \wedge soa(o_x) \neq \emptyset \Rightarrow soa(o_y) = \emptyset)))))$$

Example: Suppose there is a decision ($d_1 \in D$), a question ($q_1 \in Q$), two options ($\{o_1, o_2\} \subseteq O$), a unanimity constraint ($c_1 \in C_U$) and two option selections ($\{os_1, os_2\} \subseteq O_S$). The question belongs to the decision ($qda(d_1) = \{q_1\}$) and the options belong to the question ($oqa(q_1) = \{o_1, o_2\}$). Finally, the decision is made subject to the unanimity constraint ($cda(d_1) = \{c_1\}$). If we assume that option o_1 has been selected twice by different "users" ($soa(o_1) = \{os_1, os_2\}$), then o_2 could not have been selected ($soa(o_2) = \emptyset$) and decision d_1 is said to be compliant regarding the unanimity constraint c_1. Hereby, an option selection such as os_1 (and os_2 respectively) maps to exactly one subject and one role (see Definition A.1.10, Definition A.1.11 and Figure 6(b)). In other words, an option selection links a particular option with a subject/role combination (i.e., a "user"). Conversely, if both o_1 and o_2 have been selected once ($soa(o_1) = \{os_1\}$ and $soa(o_2) = \{os_2\}$), the invariant would not be fulfilled and the unanimity constraint c_1 is said to be violated.

CT_2 **Role Unanimity:** Contrary to an ordinary unanimity constraint, the role unanimity constraint allows for precisely specifying the set of subjects that are supposed to unanimously agree on a decision. More specifically, it allows for providing a specific set of roles. All subjects that own at least one of these roles have to unanimously agree on the same set of options for the questions of a decision.

A role unanimity constraint may be used to enforce the exemplary requirement "stakeholders with the role *Integration Architect* shall *unanimously* decide on the type of Web service" from the motivating example in Section 3.2.

Required mappings:

$sr(os) = r$	Role r that has been used to perform selection os (see Definition A.1.11)
$rruca(c) = \{r_1, ..., r_n\}$	Set of roles $\{r_1, ..., r_n\}$ that must unanimously agree on decisions that are constrained by constraint c (see Definition A.1.14)

Required sets:

C_{U_R}	An element of C_{U_R} is called *Role Unanimity Constraint*
R	An element of R is called *Role*

Therefore:

For each role unanimity constrained question (q), if one option (o_x) has been selected ($soa(o_x) \neq \emptyset$) by one of the specified roles ($sr(o_x) \in rruca(c)$), all other options must not be selected ($soa(o_y) = \emptyset$).

$$\forall c \in C_{U_R}(\forall d \in cd(c)(\forall q \in qda(d)(\forall o_x \in oqa(q)(\forall o_y \in oqa(q)(\\ o_x \neq o_y \wedge soa(o_x) \neq \emptyset \Rightarrow soa(o_y) = \emptyset \wedge \forall os_x \in soa(o_x)(\\ sr(os_x) \in rruca(c)))))$$

Example: Based on the example for CT_1 (i.e., d_1, q_1, o_1, o_2, os_1 and os_2), we assume that the decision (d_1) is made subject to a role unanimity constraint ($c_1 \in C_{U_R}$ and $cda(d_1) = \{c_1\}$). The constraint requires that subjects with a particular role ($r_1 \in R$ and $r_1 \in rruca(c_1)$) have to unanimously agree on a particular option. If we assume that option o_1 has been selected twice by different "users" ($soa(o_1) = \{os_1, os_2\}$) with that particular role ($sr(os_1) = sr(os_2) = r_1$), then o_2 could not have been selected ($soa(o_2) = \emptyset$) and decision d_1 is said to be compliant regarding the role unanimity constraint c_1. Conversely, if both o_1 and o_2 have been selected once ($soa(o_1) = \{os_1\}$ and $soa(o_2) = \{os_2\}$), the invariant would not be fulfilled and the role unanimity constraint c_1 is said to be violated. The constraint is also violated if either option has been selected by a role other than the required one ($sr(os_1) \notin \{r_1\}$ or $sr(os_2) \notin \{r_1\}$).

CT_3 **Subject Unanimity:** The subject unanimity constraint is similar to the role unanimity constraint, but instead of specifying a set of roles (supposed to unanimously agree on a decision), it allows for directly specifying a set of subjects.

Required mappings:

$ss(os) = s$	Subject s that has performed selection os (see Definition A.1.10)
$ssuca(c) = \{s_1, ..., s_n\}$	Set of subjects $\{s_1, ..., s_n\}$ that must unanimously agree on decisions that are constrained by constraint c (see Definition A.1.15)

Required sets:

C_{U_S}	An element of C_{U_S} is called *Subject Unanimity Constraint*
S	An element of S is called *Subject*

Therefore:

For each subject unanimity constrained question (q), if one option (o_x) has been selected ($soa(o_x) \neq \emptyset$) by one of the specified subjects ($ss(o_x) \in$

ssuca(c)), all other options must not be selected ($soa(o_y) = \emptyset$).

$$\forall c \in C_{U_S}(\forall d \in cd(c)(\forall q \in qda(d)(\forall o_x \in oqa(q)(\forall o_y \in oqa(q)(\\ o_x \neq o_y \wedge soa(o_x) \neq \emptyset \Rightarrow soa(o_y) = \emptyset \wedge \forall os_x \in soa(o_x)(\\ ss(os_x) \in ssuca(c)))))$$

Example: Based on the example for CT_1 (i.e., d_1, q_1, o_1, o_2, os_1 and os_2), we assume that the decision (d_1) is made subject to a subject unanimity constraint ($c_1 \in C_{U_S}$ and $cda(d_1) = \{c_1\}$). The constraint requires that particular subjects ($\{s_1, s_2\} \subseteq S$ and $\{s_1, s_2\} \subseteq ssuca(c_1)$) have to unanimously agree on a particular option. If we assume that option o_1 has been selected twice ($soa(o_1) = \{os_1, os_2\}$) by that particular subjects ($ss(os_1) \in \{s_1, s_2\}$ and $ss(os_2) \in \{s_1, s_2\}$), then o_2 could not have been selected ($soa(o_2) = \emptyset$) and decision d_1 is said to be compliant regarding the subject unanimity constraint c_1. Conversely, if both o_1 and o_2 have been selected once ($soa(o_1) = \{os_1\}$ and $soa(o_2) = \{os_2\}$), the invariant would not be fulfilled and the subject unanimity constraint c_1 is said to be violated. The constraint is also violated if either option has been selected by a subject other than the required ones ($ss(os_1) \notin \{s_1, s_2\}$ or $ss(os_2) \notin \{s_1, s_2\}$).

The following two Constraint Types (CT_{4-5}) belong to the family of **Responsibility** constraints. Such Responsibility constraints concern the decision generation process and can be used to precisely specify which particular set of stakeholders shall be entitled to eventually make (i.e., generate) a decision.

CT_4 **Responsible Role:** Each decision that is subject to a responsible role constraint shall be transformable into documented decisions *only* by subjects that own at least one of given set of roles. In other words, this constraint allows for preventing subjects that do not own specific roles from making (i.e., generating) certain decisions.

A responsible role constraint may be used to enforce the exemplary requirement "*Security Experts* shall propose a solution for the security and encryption related decisions" from the motivating example in Section 3.2.

Required mappings:

$dda(d) = \{dd_1, ..., dd_n\}$	Set of documented decisions $\{dd_1, ..., dd_n\}$ that have been generated from decision d (see Definition A.1.6)
$gr(dd) = r$	Role r that has been used to generate documented decision dd (see Definition A.1.8)
$rrrca(c) = \{r_1, ..., r_n\}$	Set of roles $\{r_1, ..., r_n\}$ that are responsible for generating decisions that are constrained by constraint c (see Definition A.1.16)

Required sets:

C_{R_R}	An element of C_{R_R} is called *Responsible Role Constraint*

Therefore:

If documented decisions (dd) have been generated from a responsible role constrained decision (d), these decisions have to be generated by a subject using one of the specified (responsible) roles ($gr(dd) \in rrrca(c)$).

$$\forall c \in C_{R_R}(\forall d \in cd(c)(\forall dd \in dda(d)(gr(dd) \in rrrca(c))))$$

Example: Based on the example for CT_1 (i.e., d_1, q_1, o_1, o_2, os_1 and os_2), we assume that the decision (d_1) is made subject to a responsible role constraint ($c_1 \in C_{R_R}$ and $cda(d_1) = \{c_1\}$). The constraint requires that subjects with a particular role ($r_1 \in R$ and $r_1 \in rrrca(c_1)$) have to generate decisions that are based on d_1. If we assume that decision dd_1 has been generated from d_1 ($dd_1 \in dda(d_1)$) by a subject with that particular role ($gr(dd_1) = r_1$), then decision d_1 is said to be compliant regarding the responsible role constraint c_1. Conversely, if dd_1 had been generated by a role other than the required one ($gr(dd_1) \notin \{r_1\}$), the invariant would not be fulfilled and the responsible role constraint c_1 is said to be violated.

CT_5 **Responsible Subject:** Similarly to the responsible role constraint, the responsible subject constraint allows for precisely specifying a set of subjects that are supposed to make (i.e., generate) certain decisions. Conversely, all other subjects shall not be allowed to do so.

Required mappings:

$gs(dd) = s$	Subject s that has generated documented decision dd (see Definition A.1.7)
$srsca(c) = \{s_1, ..., s_n\}$	Set of subjects $\{s_1, ..., s_n\}$ that are responsible for generating decisions that are constrained by constraint c (see Definition A.1.17)

Required sets:

C_{R_S}	An element of C_{R_S} is called *Responsible Subject Constraint*

Therefore:

If documented decisions (dd) have been generated from a responsible subject constrained decision (d), these decisions have to be generated by one of the specified (responsible) subjects ($gs(dd) \in srsca(c)$).

$$\forall c \in C_{R_S}(\forall d \in cd(c)(\forall dd \in dda(d)(gs(dd) \in srsca(c))))$$

Example: Based on the example for CT_1 (i.e., d_1, q_1, o_1, o_2, os_1 and os_2), we assume that the decision (d_1) is made subject to a responsible subject constraint ($c_1 \in C_{R_S}$ and $cda(d_1) = \{c_1\}$). The constraint requires that a particular subject ($s_1 \in S$ and $s_1 \in srsca(c_1)$) has to generate decisions that are based on d_1. If we assume that decision dd_1 has been

generated from d_1 ($dd_1 \in dda(d_1)$) by that particular subject ($gs(dd_1) = s_1$), then decision d_1 is said to be compliant regarding the responsible subject constraint c_1. Conversely, if dd_1 had been generated by a subject other than the required one ($gs(dd_1) \notin \{s_1\}$), the invariant would not be fulfilled and the responsible subject constraint c_1 is said to be violated.

The following Constraint Type (CT_6) can be considered a generic **Access Control** constraint. In the context of CoCoADvISE it concerns constraining the set of stakeholders that shall be *authorized* to select options or generate decisions. Whereas CT_6 concerns authorization (i.e., who shall potentially be entitled to do what), CT_{2-5} concerns *obligation* (i.e., who must do what).

CT_6 **Required Permission:** A required permission constrained decision respective decision model enforces that subjects that do not own certain permissions (i.e., via their roles) within the system must not perform the corresponding actions.

Required mappings:

$prpca(c) = \{p_1, ..., p_n\}$	Set of permissions $\{p_1, ..., p_n\}$ that are required for performing actions concerning decisions that are constrained by constraint c (see Definition A.1.18)
$pra^{-1}(r) = \{p_1, ..., p_n\}$	Set of permissions $\{p_1, ..., p_n\}$ that are owned by role r (see Definition A.1.5)

Required sets:

C_P	An element of C_P is called *Required Permission Constraint*

Therefore:

If documented decisions (dd) have been generated from a decision (d) that is subject to a required permission constraint (c), the generating role ($gr(dd)$) must own the corresponding permission ($generateDecision \in pra^{-1}(gr(dd))$) to generate decisions.

$$\forall c \in C_P(generateDecision \in prpca(c) \Rightarrow \forall d \in cd(c)(\forall dd \in dda(d)(generateDecision \in pra^{-1}(gr(dd)))))$$

If options of a decision (d) that is subject to a required permission constraint (c) have been selected (os), the selecting role ($sr(os)$) must own the permission to select options ($selectOption \in pra^{-1}(sr(os))$).

$$\forall c \in C_P(selectOption \in prpca(c) \Rightarrow \forall d \in cd(c)(\forall q \in qda(d)(\forall o \in oqa(q)(\forall os \in soa(o)(selectOption \in pra^{-1}(sr(os)))))))$$

Note that similar invariants for other actions, such as deleting questionnaires or decisions, have been omitted for brevity.
Example: Based on the example for CT_1 (i.e., d_1, q_1, o_1, o_2, os_1 and os_2),

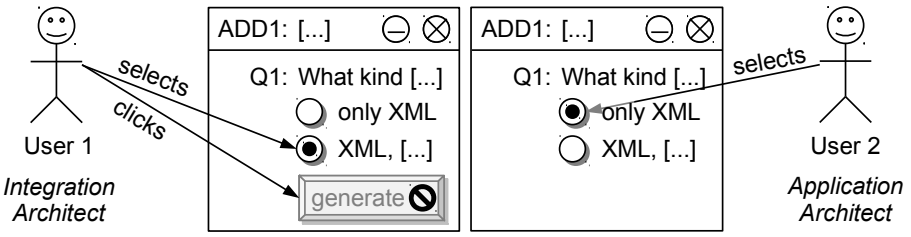


Figure 9: Enforcing Exemplary Constraints at Execution Time

we assume that the decision model is made subject to a required permission constraint ($c_1 \in C_P$). The constraint mandates – among other things – that all option selections have to be performed by subjects owning the required *selectOption* role. If we assume that option selection os_1 has been performed by a subject using a role ($r_1 \in R$) that owns that particular permission ($selectOption \in pra^{-1}(r_1)$), then the decision model is said to be compliant regarding the required permission constraint c_1. Conversely, if r_1 did not own that particular permission ($selectOption \notin pra^{-1}(r_1)$), the invariant would not be fulfilled and the required permission constraint c_1 is said to be violated.

At execution time, CoCoADvISE interprets constrainable decision models and constantly checks the system's compliance to the defined constraints while the software development team members are making and documenting decisions. The actual constraint enforcement logic is mainly embedded in the user interface of the collaborative Web application. The constraints are automatically enforced in the same way, the system also guarantees the consistency of the decision model instances, i.e., by automatically hiding, showing, disabling, etc. specific parts of the questionnaires and the user interfaces in a way that users simply can not violate any defined constraints at all. Figure 9 visualizes a possible runtime situation of the previously defined decision (see Figure 3) and constraint model (see Figure 7). We can see that the specified responsible role constraint is enforced by showing the "generate" button only to those users that own the required role (i.e., User 1, which owns the role *Integration Architect*), while hiding it for all other users (i.e., User 2). Given that User 1 chooses option 2 and User 2 chooses another option 1, the system correctly enforces the unanimity constraint by disabling the "generate" button. As soon as all users have unanimously agreed on the same set of options, the system will eventually enable the button again, allowing User 1 to generate a documented decision.

4.4. Implementation Details

CoCoADvISE is a real-time collaborative Single-page Web application that is founded on Google's Web application framework AngularJS[9]. Thus, it is exe-

[9] `http://angularjs.org`

cuted mostly client-side (i.e., in the user's Web browser). The model invariants for each constraint type (i.e., CT_{1-6}) have been transformed manually into client-side executable code. A key component of real-time collaborative Web applications is a real-time model synchronization engine that allows for synchronizing the shared application state with all clients. CoCoADvISE leverages Racer[10] for synchronizing the decision making process with all clients. Racer consists of both client-side and server-side executed code. All server-side code is executed in Node.js[11], an asynchronous event driven framework and runtime environment based on Google's V8[12] JavaScript engine. The back-end of this Thin Server Architecture persists the application state using MongoDB[13], a document-oriented (i.e., NoSQL) database. According to the tool CLOC[14] (Count Lines of Code), the application consists of nearly 2100 lines of client-side executed JavaScript code, roughly 1000 lines of HTML code and 150 lines of server-side executed Javascript code.

In addition to collaborative editing of questionnaires and architectural design decisions, CoCoADvISE also provides a simple chat, which allows all stakeholders to participate in discussions concerning the decision making process.

4.5. Motivating Example Resolved

In the course of revisiting the motivating example from Section 3.2, Figure 10 shows screenshots of CoCoADvISE, currently displaying the mentioned decision model excerpt from an *Integration Architect's* point of view. ① indicates that another *Software Architect* with the name "experiment1" has currently selected a different option than "we". Due to the Unanimity constraint, the system automatically disables the corresponding "Generate decision" button ②. By selecting the same option as "experiment1" ③ "we" can successfully resolve this constraint violation and eventually the system allows for generating the corresponding documented decision ④. Finally, ⑤ displays an excerpt of this generated decision.

5. Application of Constrainable Collaboration in Service-based Platform Integration

In practice, more than two roles with various intertwining responsibilities, rights, and permissions are involved in decision making on architectural-related issues. In this section, we present the application of constrainable collaborative architectural decision making in the domain of service-based platform integration. In particular, we discuss the implementation of our approach to support architecture governance in tailoring of heterogeneous domain-specific platforms,

[10] `http://github.com/codeparty/racer`
[11] `http://nodejs.org`
[12] `http://code.google.com/p/v8`
[13] `http://mongodb.org`
[14] `http://cloc.sourceforge.net`

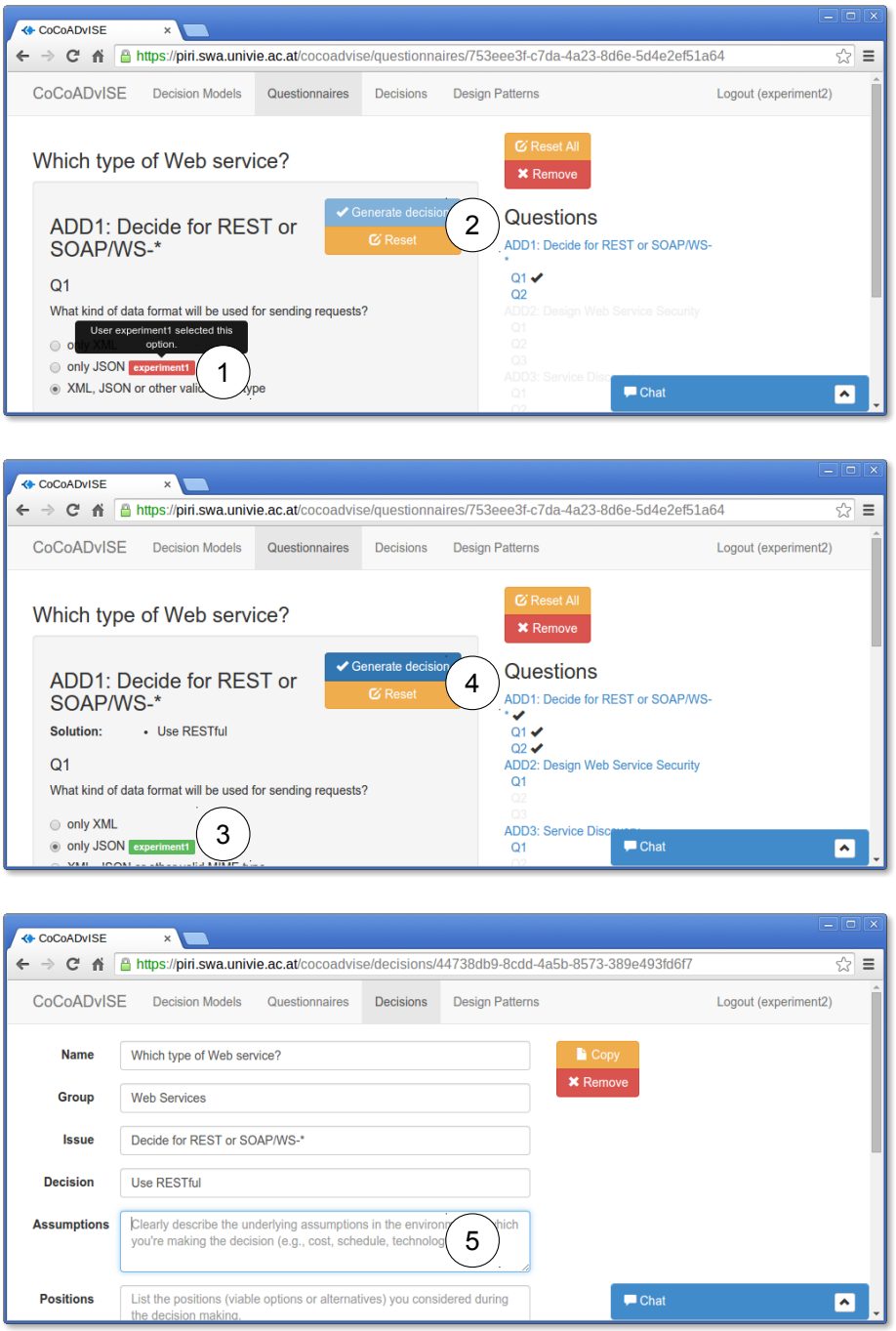


Figure 10: Screenshots of CoCoADvISE

which was investigated in the context of the EU research project INDENICA[15], which included project partners from the industry. INDENICA and in particular one of its technical reports [44] on role-based governance originally inspired us to devising the approach presented in this paper.

Building domain-specific service platforms is necessary for fulfilling specific requirements of various domains – sometimes incompatible to each other – in order to ease the development of services and applications within the domain. Such domain-specific service platforms may form a family of platforms, in which member platforms share assets. In this way, the INDENICA approach tailors the platforms towards the application domains and provides methods and tools for designing and implementing a Virtual Domain-Specific Service Platform (VSP). As described in the technical report [44], this approach requires the implementation and application environment to be considered from an organizational and human behavior point of view. This can be addressed by introducing different types of governance, such as Corporate Governance, Business Process Management Governance (BPM Governance), Information Technology Governance (IT Governance), Enterprise Architecture Governance (EA Governance), SOA Governance, and Architecture Governance in the different software processes. We will focus on Architecture Governance which concentrates on system and platform architecture related aspects.

Architecture Governance is defined as *"the practice and orientation by which enterprise architectures and other architectures are managed and controlled at an enterprise-wide level"* [52]. To develop a consistent architecture and ensure the evolution, adaptation, and modification of integrating domain-specific service platforms in INDENICA, architecture governance is of key importance. That is, without such governance the risk of e.g., wrong usage of services, project failure, over-complex applications, and design erosion increases. During design time, the following key participating roles have been defined:

Platform Provider is a technology expert and describes the current variability and the variability binding process of the existing platform he owns.

Platform Variant Creator is responsible for binding unresolved variability in base platform(s) and for creating an executable platform variant.

Platform Architect is responsible for VSP requirements, variability within VSP, baseline architecture and adaptation behavior of VSP.

Platform Integrator generates the VSP instance.

In Table 1, we list 16 decision categories related to service-based platform integration design that have been described in [44]. These decision categories include decision points that need to be considered, discussed, and eventually, resolved by the various stakeholders in collaboration. For instance, for Decision

[15] `http://www.indenica.eu`

Decision Category	Constraints	
	C_{R_R}	C_{U_R}
01. Decide variability modeling (describing the current variability and the variability binding process of the base platform).	PP	PP
02. Implement base platform relevant requests.	PP	
03. Bind unresolved variability in base platform.	PVC	
04. Create an executable platform variant optional.	PVC	
05. Decide additional functionality not covered by the base platform.	PVC	PP, PVC
06. Decide variability modeling of additional functionality.	PVC	PP, PVC
07. Implement domain platform relevant change requests.	PVC	PP, PVC
08. Design the VSP Capabilities (requirements management).	PA	
09. Decide the variability within VSP.	PA	
10. Decide the VSP constraints.	PA	
11. Decide the VSP orchestration.	PA	
12. Create the baseline architecture.	PA	PA, PI
13. Create the baseline adaptation behavior of VSP.	PA	PA, PI
14. Decide/Implement monitoring/adaptation rules.	PI	PA, PI
15. Decide integration of appropriate platforms.	PI	PA, PI
16. Generate the integration of the domain platforms to the VSP.	PI	

PP: Platform Provider PVC: Platform Variant Creator PA: Platform Architect
PI: Platform Integrator

Table 1: Role Constraints for Service-based Platform Integration

Category 15 (i.e., "Decide integration of appropriate platforms") we list exemplary architectural decisions regarding the integration of heterogeneous domain platforms to the VSP:

- Decide on the type of component for integrating the platform service into the VSP.
- Decide on the connection of heterogeneous systems (in terms of synchronization and queuing behavior).
- Decide on the protocol(s) for accessing the VSP from the integrating platforms.
- Decide on how to accommodate diverse protocol requirements of integrating platforms.

For each Decision Category, the report [44] also provides governance rules, i.e., a detailed description of the rights and duties of each involved stakeholder role. As these stakeholders and stakeholder roles belong to different organizations and domains, collaboratively deciding and making ADDs while complying to all governance rules is very complicated and error-prone. Using our previously described approach for supporting decision making and documentation (see Section 4), we can precisely formalize the described governance rules in the form of decision making constraints (see Section 4.2) and rely on CoCoADvISE's automatic constraint enforcement capabilities in order to stay compliant to these governance rules. For instance, the governance rules for Decision Category 1 from Table 1 can be formalized and enforced by defining a Responsible Role constraint (i.e., C_{R_R}), with a (responsible) stakeholder role of "Platform Provider" (i.e., PP) and a corresponding Role Unanimity constraint (i.e., C_{U_R}). At runtime, these two constraints will enforce, that only a stakeholder with the role PP will be responsible for deciding on variability modeling (i.e., Responsible Role) and that the other participating stakeholders with the role PP have to decide unanimously (i.e., Role Unanimity) on this matter. In a similar way, our approach can be used to enforce the governance rules of the remaining Decision Categories 2–16 too (as can be seen in Table 1).

6. Empirical Evaluation

In order to collect empirical evidence about the effectiveness and efficiency of our proposed concepts, we conducted a controlled experiment with computer science students. We designed and executed our controlled experiment following the guidelines of Kitchenham [17] and analyzed and evaluated the results according to Wohlin et al.'s advice [61]. The following subsections discuss the goals and hypotheses of the controlled experiment, as well as its design and execution in detail.

6.1. Goals and Hypotheses

The goal of the experiment is twofold. On the one hand, we want to study and quantify the benefits of automatically enforcing constraints in a collaborative architectural decision making tool. On the other hand, we want to analyze and quantify the adverse effects of constraint violations in detail. Consequently, we postulate the following hypotheses:

Automatic enforcement of constraints in CoCoADvISE...

H_{01} has no effect or decreases the *effectiveness* of its users.

H_1 increases the *effectiveness* of its users.

H_{02} has no effect or decreases the *time related efficiency* of its users.

H_2 increases the *time related efficiency* of its users.

H_{03} has no effect or decreases the *effort related efficiency* of its users.

H_3 increases the *effort related efficiency* of its users.

We expect that the corresponding null hypotheses can be rejected. That is, we expect that automatic enforcement of constraints in CoCoADvISE increases both the *effectiveness* and the *time and effort related efficiency* of its users. In particular, we expect that users will manage to achieve more of the imposed work tasks than users that can not rely on automatic enforcement of constraints. We also expect that the former will have to invest both less effort (i.e., by performing less work steps/actions) and less time in order to achieve the same results.

6.2. Parameters and Values

During the experiment several dependent and independent variables have been observed. Table 2 provides a detailed description, including the type, scale type, unit and range of those variables.

Dependent Variables. All dependent variables have been extracted automatically from CoCoADvISE's database. In particular, we instrumented its source code in such a way that we could precisely record all user activities within the system. The variable *time* indicates a single user's total time spent logged in (i.e., the sum of all session durations). Similarly, the variable *actions* counts a user's total number of essential actions or work steps within the system. In particular, we consider the following actions in CoCoADvISE to be essential: create/remove a questionnaire, generate/remove/copy a decision, and select an option to a question. The variable *violations* indicates how many violations of decision making constraints a single user has caused. Finally, the variable *work* represents how much of the required work tasks in percent could actually be achieved. For instance, users that completed 3 out of 6 work tasks got a value of 50 for their *work* variable. The concrete number of work tasks depends on the role that is randomly assigned to each user. That is, the role *Software Architect* had to perform 7 tasks and the *Application Developer* 6, respectively.

Derived Variables. To allow for a meaningful comparison of *time* and *actions*, we decided to introduce two additional derived variables: $timeNorm$ and $actionsNorm$. In particular, we normalize the *time* and *actions* variables by dividing them by *work*. As a result, $timeNorm$ can be interpreted as the total time a user would have needed to finish all work tasks. Comparing, for instance, $timeNorm$ instead of *time*, rules out the possibility that the participants of one treatment group needed less time only because they "worked less" (i.e., in terms of *work*) than the participants of the other group.

Independent Variables. The independent variables *group*, *exp* and $commExp$ can potentially influence the dependent variables. In particular, *group* contains a participant's treatment group, and *exp* and $commExp$ concern their programming experience and commercial programming experience, respectively.

Type	**Name**	**Description**	**Scale Type**	**Unit**	**Range**
Dependent	*time*	Overall time needed to make and document decisions	Ratio	Minutes	Positive natural numbers including 0
	actions	Number of actions performed	Ratio	–	Positive natural numbers including 0
	violations	Number of constraint violations caused	Ratio	–	Positive natural numbers including 0
	work	Percentage of work that a single user is supposed to perform	Ratio	–	0 (lowest) to 100 (highest)
Derived	$timeNorm$ $(= \frac{time}{work})$	Time that would be needed to perform 100% of *work*	Ratio	Minutes	Positive natural numbers including 0
	$actionsNorm$ $(= \frac{actions}{work})$	Number of actions that would need to be performed to accomplish 100% of *work*	Ratio	–	Positive natural numbers including 0
Independent	*group*	Treatment group	Nominal	–	Either "experiment" or "control"
	exp	Programming experience	Ordinal	Years	4 classes: 0-1, 1-3, 3-6, >6
	commExp	Commercial programming experience in industry	Ordinal	Years	4 classes: 0-1, 1-3, 3-6, >6

Table 2: Observed and Derived Variables

6.3. Experiment Design

The controlled experiment was conducted in the context of Information System Technologies lecture at the Faculty of Computer Science, University of Vienna, Austria, in January 2014.

Participants. From the 48 students of the lecture, 26 participated in the control group and 22 in the experiment group, in teams of two students. The experiment was part of a practical exercise on architectural decisions for service-based software systems. The practical exercises took place in four separate groups (in different rooms) to which the students were randomly assigned. All students had background in Java programming, Web services, and design patterns.

Objects. As the basis for making and documenting pattern-based architectural decisions collaboratively and remotely, a list of documented architectural design patterns, as well as a set of reusable architectural decision models, were provided in the CoCoADvISE tool. The design patterns and architectural decision models were selected based on the lecture materials known to the students and the students' experiences from the previous practical exercises.

Instrumentation. In the preparation phase, all participants were given an introduction to CoCoADvISE and were asked to study the catalog of architectural design patterns and related technologies. Before starting with the experiment tasks, all participants had to fill in a short questionnaire regarding their programming experiences. Afterwards, all participants were provided with a description and requirements of the system to be designed ("An Online Retailer for Selling Books and Gadgets"), a description of the different stakeholder roles, their responsibilities, as well as a description of additional constraints regarding the collaborative decision making process. In Table 3 we give an example of the software system's requirements, and in Table 4 we summarize the two stakeholder roles along with an excerpt of their privileges and responsibilities. Note that in reality a larger number of roles would be needed, but in order to reach a controlled environment we had to simplify the roles used in the experiment. In total, the students had to consider three groups of requirements (*Expose Web Services to a Web Shop*, *Customer Login*, and *Manage Different Formats for Inventories*), given in descriptive form. Additionally, some hints were provided with information about the concrete decisions that were expected. Each requirement had to be covered by one or more documented architectural decisions.

Eventually, all participants were given access to the CoCoADvISE tool. For the needs of the controlled experiment a detailed list of related architectural design patterns and three reusable architectural decision models with use instructions were provided in the tool. The functionality of CoCoADvISE is described in Section 4 and the setting provided to the students can be viewed at `https://piri.swa.univie.ac.at/cocoadvise`[16]. The participants needed to

[16] Use the following user names (no password required): experiment1, experiment2, control1 or control2.

Name	Description
Customer Login	A customer needs to login in order to purchase books and gadgets online and is responsible for saving his/her session in order to keep the state of his/her orders (stateful remote objects). If the session is inactive for a predefined time period the session should expire and the customer gets automatically logged out. *Decide* how to implement the creation and lifecycle management of the sessions. **Hint:** Create a questionnaire based on the Resource and Lifecycle Management decision model.

Table 3: Online Retailer Requirement Example

Software Architect	The Software Architect is responsible for high-level decisions rather than for implementation details.
Application Developer	The Application Developer is responsible for decisions that refer to low-level design and implementation details.
Privileges	Only the Software Architect should be able to create the questionnaires giving a name already agreed/edited by both the Software Architect and the Application Developer. Also, only the creator of questionnaires and decisions is able to remove them, i.e., you are not allowed to delete questionnaires or decisions of your partner. [...] The Software Architect will make the final decision (generate decision) about the type of Web Service that will be exposed by the Online Retailer (as well as the transport protocol) but has to agree with the Application Developer on this before he/she makes the final decision. The same applies for the architectural decision regarding the service discovery. Once the decision about the type of web service has been made, the Application Developer can proceed with deciding the security and encryption of the web service. Only the Application Developer is responsible for deciding on security issues and the Software Architect should not interfere in this issue. [...]

Table 4: Decision Making Roles with their Privileges

reuse three architectural decision models (*Resource and Lifecycle Management*, *Message Transformations*, and *Web Services*) in teams, in order to make and document the architectural decisions related to the given requirements.

The crucial difference between the experiment and the control group was that we completely disabled the automatic constraint enforcement functionality for users belonging to the control group. In other words, the control group's members were completely responsible on their own for working in a way that no constraints are violated. In particular, they had to detect violations on their own and they also had to resolve them "manually". In contrast, these tasks have been automated for the experiment group.

Note that for the purpose of this experiment only the following constraint types have been considered: Unanimity, ResponsibleRole and RequiredPermission.

6.4. Execution

As described in the previous section, the experiment was executed in the context of the Information System Technologies lecture at the Faculty of Computer Science, University of Vienna in the Winter Semester 2013/2014.

The participants were randomly divided into groups of two persons and also randomly assigned to experiment and control group. As two of the four course groups with different numbers of students took place simultaneously at a time and the collaborating students were not allowed to work in the same room, we had to divide the participants in unequal groups of 22 (experiment group) and 26 (control group) participants. Three participants of the experiment group were excluded, as they did not hand in any results (that is, they did not edit any questionnaires or decisions).

As we can see in Figure 11 the programming experience, as well as the industry programming experience of the participants are comparable in both treatment groups with the control group having slightly longer programming experience and most of the students having more than 2.5 years of programming experience but very few having experience in industrial projects (0-1 years).

The same materials were handed out to all participants in the beginning of the exercise. The experiment group used the different version of CoCoADvISE where the constraints (such as the ones in Table 4) were integrated and automatically enforced (i.e., they could not be violated in any way).

The experiment was executed in two sessions of 90 minutes. In this time period, the students had to read, understand and execute the exercise. They were allowed to finish with the tasks earlier. Access to the tool was given only during these sessions to avoid offline work or discussion among students.

The collection of the participants' data has been performed automatically during the experiment. In particular, all relevant information, such as created questionnaires, selected options, and exchanged messages, as well as all relevant events, such as deletions or modifications of architectural decisions and changes of selected options were saved in a database.

No deviations from the initial study design occurred and no situations in which participants behaved unexpectedly.

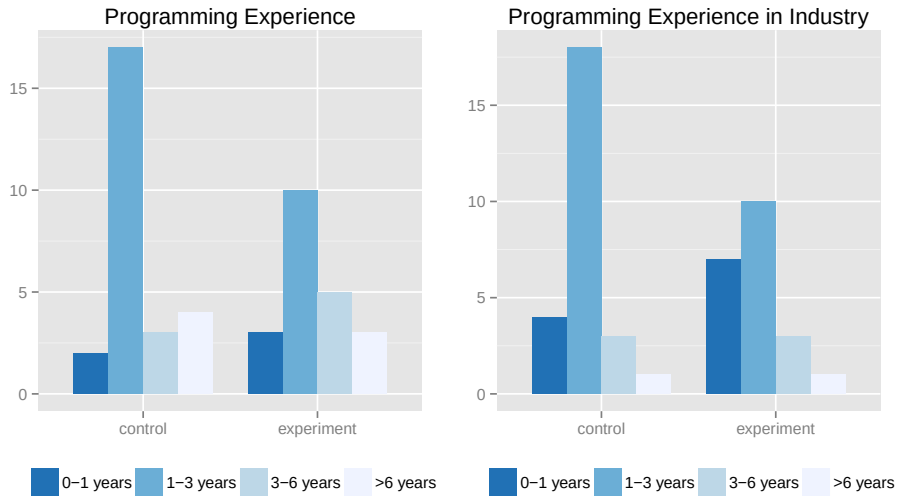


Figure 11: Participants' Programming Experience

7. Analysis of Results

The following statistical analysis has been carried out using R language and environment for statistical computing [51]. Note that the raw data as well as the corresponding R script for calculating these results are available online at `https://piri.swa.univie.ac.at/cocoadvise-experiment`.

7.1. Descriptive Statistics

As a first step in our analysis, we use descriptive statistics to compare observed variables related to the efficiency and the effectiveness of making and documenting architectural decisions. That is, Table 5 and Figure 12 display the mean and median values for the number of actions the participants of each treatment group needed to perform in order to complete the exercise (*actions*), the total time they needed (*time*), the percentage of tasks they completed (*work*), and the number of constraint violations they caused (*violations*).

It is clearly noticeable that the experiment group spent less time working on the exercise and had to perform less actions than the control group. A user of the control group caused 5.23 constraint violations on average, which are automatically prevented for the experiment group by our tool. The experiment group could finish more work tasks than the control group (i.e., 85.03% vs. 71.79% on average). Given these results, it makes sense to take a closer look at our derived variables (i.e., $timeNorm$ and $actionsNorm$). We can see that the gap between both treatment groups gets wider when we look at these derived variables. That is, the experiment group would need roughly 41% less time and nearly 44% less actions in order to completely finish all required work tasks.

Variable	Means		Medians	
	control	**experiment**	**control**	**experiment**
time (min)	163.72	120.53	148.33	120.21
timeNorm (min)	244.25	146.72	221.41	142.57
actions	95.15	67.05	88.50	57.00
actionsNorm	140.81	80.08	127.20	60.00
violations	5.23	–	3.50	–
work (%)	71.79	85.03	71.43	85.71

Table 5: Means and Medians of Observed Variables

Constraint Type	Means	Medians
Unanimity	0.27	0.00
ResponsibleRole	3.38	2.00
RequiredPermission	1.85	0.00

Table 6: Observed Constraint Violations per User in the Control Group

Finally, Table 6 provides the mean and median values of the observed constraint violations per type and per user in the control group. We notice that, on average, ResponsibleRole constraints were violated 3.38 times per user, followed by RequiredPermission constraints, which were violated 1.85 times per user.

7.2. Data Set Reduction

Studying the deviations from the means for each of the four variables that we observed we noticed a few outliers, i.e., points that are either much higher or much lower than the mean values. As these potential candidate data points for exclusion correspond to different participants (for instance, a student delivered more required work in less time) these single outlier points do not necessarily make the participant an outlier. Thus, we decided to exclude only participants who did not perform any action and delivered 0% of the required work tasks, and who therefore would make the study results vulnerable. This was done, however, before the data analysis (see explanation in Section 6.4); at this stage, we did not perform any further data set reduction.

7.3. Hypotheses Testing

Testing for Normal Distribution. In order to see whether we can apply parametric tests like the t-test that assume the normal distribution of the analyzed data, we tested the normality of the data by applying the Shapiro-Wilk test [43]. The null hypothesis of the Shapiro-Wilk test states that the input data is normally distributed. It is tested at the significance level of $\alpha = 0.05$ (i.e., the level of confidence is 95%). That is, if the calculated p-value is lower than 0.05 the null hypothesis is rejected and the input data is not normally distributed. If the

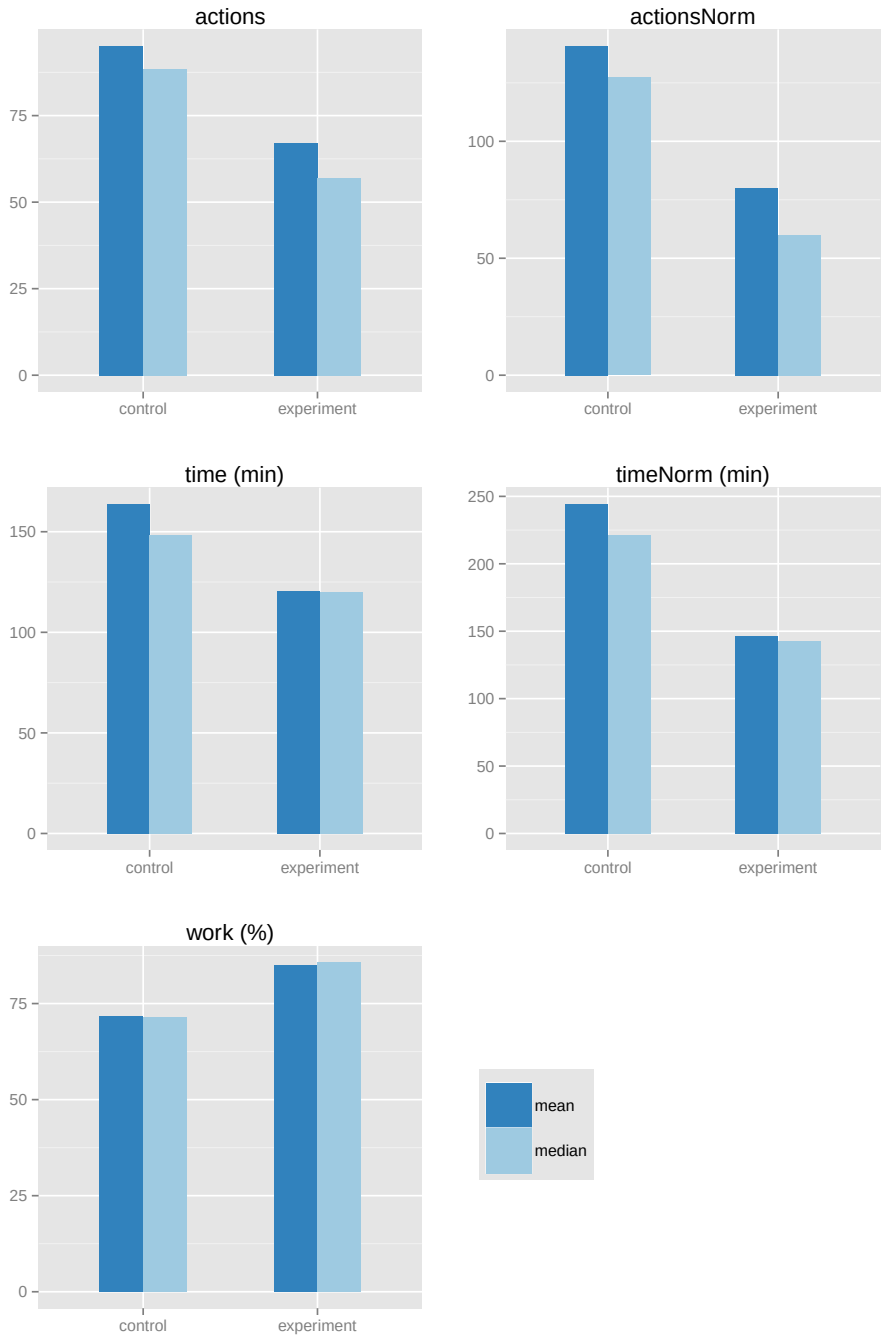


Figure 12: Means and Medians of Observed Variables

Variable	p-Value	
	control	**experiment**
time	0.0571	0.8451
actions	0.0092	0.0017
violations	0.0045	–
work	0.0850	0.0047

Table 7: Shapiro-Wilk Normality Test

p-value is higher than 0.05, we can not reject the null hypothesis that the data is normally distributed.

Table 7 lists the p-values of the Shapiro-Wilk normality test for each observed variable and treatment group. We can see that only *time* exhibits a very weak tendency of being normally distributed, while for all other variables it can not be concluded that they are normally distributed. As a result, we decided to pursue non-parametric statistical tests with our data.

Comparing the Means of Variables. To compare the means of variables, we applied the Wilcoxon rank-sum test [25]. The one-tailed Wilcoxon rank-sum test is a non-parametric test for assessing whether one of two data samples of independent observations is stochastically greater than the other. Its null hypothesis, which is appropriate for the hypotheses in our experiment, is that the means of the first variable's distribution is less than or equal to the means of the second variable's distribution, so that we can write $H_0 : A \leq B$. The Wilcoxon rank-sum test tries to find a location shift in the distributions, i.e., the difference in means of two distributions. The corresponding alternative hypothesis H_A could be written as $H_A : A > B$. If a p-value for the test is smaller than 0.05 (i.e., the level of confidence is 95%), the null hypothesis is rejected and the distributions are shifted. If a p-value is larger than 0.05, the null hypothesis can not be rejected, and we can not claim that there is a shift between the two distributions.

Table 8 contains the p-values of five Wilcoxon rank-sum tests that were performed to test our hypotheses (see Section 6.1). It also contains the corresponding null hypotheses (e.g., $\mathbf{H_{01}}$ is the null hypothesis of $\mathbf{H_1}$) and their assumptions regarding the means of a specific variable. Based on the obtained p-values, we can assess that all distributions show a statistically significant shift between each other and that all null hypotheses can be rejected.

Testing Hypothesis $\mathbf{H_1}$. In our experiment, we observed that the experiment group was able to perform more work tasks than the control group, i.e., their participants were *more effective* than the participants of the other group. With a p-value of 0.0093 we can *reject* the null hypothesis $\mathbf{H_{01}}$ (i.e., automatic enforcement of constraints in CoCoADvISE has no effect or decreases the *effectiveness* of its users). Hence, we can *accept* $\mathbf{H_1}$.

Hypothesis	Assumption	Variable (μ)	p-Value
$\mathbf{H_{01}}$	$\mu_{exp} \leq \mu_{control}$	$work$	0.0093
$\mathbf{H_{02}}$	$\mu_{exp} \geq \mu_{control}$	$time$	0.0040
		$timeNorm$	0.0003
$\mathbf{H_{03}}$	$\mu_{exp} \geq \mu_{control}$	$actions$	0.0018
		$actionsNorm$	0.0001

Table 8: Hypothesis Testing Results

That is, there is evidence that the automatic enforcement of constraints increases the *effectiveness* of its users.

Testing Hypothesis $\mathbf{H_2}$. We also found that the experiment group needed less time than the control group, i.e., its members were *more efficient* in terms of *time invested* than the members of the other group. This observation holds for both the observed variable *time* and the derived variable $timeNorm$. Hence, we tested the null hypothesis $\mathbf{H_{02}}$ (i.e., automatic enforcement of constraints in CoCoADvISE has no effect or decreases the *time related efficiency* of its users) for both variables. As both p-values were below 0.05 (i.e., 0.0040 in the case of *time* and 0.0003 in the case of $timeNorm$) we can *reject* $\mathbf{H_{02}}$ and *accept* $\mathbf{H_2}$.

That is, there is evidence that automatic enforcement of constraints increases the *time related efficiency* of its users.

Testing Hypothesis $\mathbf{H_3}$. Finally, we discovered that the experiment group performed less actions than the control group, i.e., its participants were *more efficient* in terms of *effort invested* than the participants of the other group. As this observation holds for both *actions* and $actionsNorm$ we tested the null hypothesis $\mathbf{H_{02}}$ (i.e., automatic enforcement of constraints in CoCoADvISE has no effect or decreases the *effort related efficiency* of its users) for both variables. The p-values of 0.0018 and 0.0001 led us to *reject* $\mathbf{H_{03}}$ and *accept* $\mathbf{H_3}$.

Hence, we conclude that there is evidence that automatic enforcement of constraints also increases the *effort related efficiency* of its users.

7.4. Regression Analysis

In order to better understand the adverse effects of constraint violations, this section presents a linear regression analysis. Table 9 depicts three different linear regression models that can be used to quantify the effect of constraint violations (*violations*) on a user's effectiveness ($work$) and efficiency (*actions* and *time*). In particular, it shows the explained variable, the value of the coefficient (i.e., *violations*) and the p-value of the corresponding (two-tailed) t-test, the regression's R^2 and the p-value of the corresponding F-test as well as the number of outliers that had to be excluded from the regression.

The null hypothesis of the t-test assumes that the corresponding coefficient has a value of 0. At the significance level of $\alpha = 0.05$, a p-value lower than 0.05 provides evidence that the coefficient is significantly different from 0. R^2, the

Variable	Coefficient (*violations*)		R^2	p-Value	# Outliers
	Value	p-Value			
time	5.6230	0.0039	0.1671	0.0039	0
actions	4.1728	0.0000006	0.4596	0.0000006	5
work	−1.2032	0.0468	0.0868	0.0468	2

Table 9: Linear Regression Models

coefficient of determination, is used as an indicator of how well a linear regression fits a set of data. The statistical test of significance for R^2 is the F-test. Linear regressions require the following four crucial assumptions to hold: Linearity, Homoscedasticity, Uncorrelatedness and Normality. Pena et al. proposed a procedure for testing these assumptions [35]. We used the corresponding `gvlma` R package for assuring (at a significance level of $\alpha = 0.05$) that the four assumptions hold for our linear regressions. Note that we had to iteratively increase the number of outliers to be excluded from the regressions until the `gvlma` package confirmed that all assumptions are justified. Eventually we had to exclude 5 participants from the regression that explains *actions* and 2 for the regression explaining *work*.

As we can see, the p-values of all t-tests and F-tests are below 0.05. Hence, we consider all coefficient values and the R^2 for each regression to be statistically significant at the significance level of $\alpha = 0.05$. Each regression model can be used to quantify the effect of a single constraint violation on each of the explained variables. For instance, regarding *time* our model predicts that a single constraint violation increases the overall time needed to make and document decisions by nearly 6 minutes. Similarly, a violation increases *actions*, i.e., the number of actions that a user performs, by roughly 4. With an R^2 of 0.4596 it can also be noted, that the coefficient *violations* "can explain" nearly 46% of *actions* variability. Finally, there is a negative relation between *violations* and *work*. According to our model, a single constraint violation reduces the percentage of work tasks that a user manages to accomplish by roughly 1.2%.

In summary, these regression models fortify and complement our main findings. While our hypotheses dealt with finding evidence that automatic enforcement of constraints is beneficiary in terms of effectivity and efficiency in general, the regression models provide further insights into (1) what exactly are the adverse effects of constraint violations and (2) to which extent they influence the effectivity and efficiency of users.

8. Discussion

The following subsections discuss our main findings and their implications as well as their threats to validity.

8.1. Evaluation of Results and Implications

Increased Effectiveness. Hypothesis $\mathbf{H_1}$ and the corresponding null hypothesis $\mathbf{H_{01}}$ concern the effectiveness of users of collaborative and constrained architectural decision making tools. In Section 7.3 we could provide evidence that the null hypothesis $\mathbf{H_{01}}$ can be rejected. Thus, automatic enforcement of constraints increases the effectiveness of its users.

We interpret this finding as follows. The concrete set of work tasks a specific user has to complete stems from the role that has been assigned to the user. In our experiment, the duties of each role have been described textually, as can be seen in Table 4. For instance, a concrete work task of users with the role *Software Architect* is that they are supposed to generate the decision about the type of Web service to be deployed. If the other user generates the decision instead, we do not increment the number of successfully accomplished work tasks of the *Software Architect*. In Table 6 we can see that ResponsibleRole constraints were violated 3.38 times per user (on average). Thus, it seems that many users were unsure or confused about who is supposed to do what in the decision making process. In fact, similar issues have been documented in [28] and [1].

As many users performed tasks that were supposed to be performed by another user and the fact that our approach for calculating the percentage of accomplished work penalizes these "deviations from the prescribed regulations", we conclude that our experiment provides evidence that automatic enforcement of constraints increases the effectiveness of its users.

Increased Efficiency. In Section 7.3 we could provide evidence that both null hypotheses $\mathbf{H_{02}}$ and $\mathbf{H_{03}}$ can be rejected. The corresponding alternative hypotheses $\mathbf{H_2}$ and $\mathbf{H_3}$ concern the efficiency of users of collaborative and constrained architectural decision making tools. Thus, automatic enforcement of constraints increases the efficiency of its users. To be exact, it increases both the time and effort related efficiency.

We have the following explanation for these findings. In order to be able to "manually" detect and prevent constraint violations, users have to read and understand the description and meaning of each defined constraint type first. Then, during working on their work tasks and performing actions, they have to be careful not to (unintentionally) cause constraint violations. In case a violation happens anyway, there are two possibilities. If the violation gets detected, the users have to resolve the violation, e.g., by revoking and redoing already performed tasks. In general, detecting and resolving constraints requires an additional investment of both time and effort. Violations which are not detected decrease – among other things – the effectiveness of users. To this end, especially our linear regression model (see Section 7.4) can be interpreted as a good estimator for predicting the effort reduction to be expected when introducing automatic enforcement of constraints. For instance, if we would expect an average of 5.23 constraint violations per user (which is the actual observed mean for *violations* in our experiment), our model predicts that we can anticipate our users to require 21.82 additional work steps needed to resolve these viola-

tions again. Analogously, these additional work steps are predicted to cost 34.64 additional minutes.

In summary, we can conclude that our experiment provides evidence that automatic enforcement of constraints increases the efficiency of users, because it takes away the burden of detecting, preventing and resolving constraint violations "manually" from the user.

Initial Development and Modeling Effort. A possible limitation of our approach is that automatic enforcement of decision making constraints, as proposed in this paper, is only possible if the required amount of time and effort gets invested into modeling decision and constraint specifications before the tool is used. In addition, in the rare case that a new constraint type is introduced, developers have to augment the tool with additional constraint checking and enforcement logic. As our approach is based on *reusable* architectural decision models that are supposed to be instantiated more than once, the modeling effort is only required once per reusable decision model and constraint type. Nowak et al. envision the idea of collaboration and knowledge exchange between different architectural knowledge repositories (i.e., repositories containing reusable architectural decision models) [32]. When applied to our context, this means a further reduction of initial modeling effort. Thus, models are shared, reused and adapted instead of built from scratch. Hence, except for rarely used decisions or constraint types this limitation should be negligible.

8.2. Threats to Validity

To ensure the validity of our results, we consider the categorization of validity threats of Wohlin [61] and discuss each of them separately in the context of our controlled experiment.

Conclusion Validity. The conclusion validity focuses on the relationship between the treatment we used in the experiment and the actual outcome, i.e., on the existence of a significant statistical relationship.

The way we measured the working time of the students automatically from the database entries may pose a threat to conclusion validity, as users might have spent some observed working time idle or with other tasks, or they might have worked offline without the system noticing the working time. In addition, to measure the actual time spent on working with the CoCoADvISE tool is very difficult, if not impossible, as the participants may have spent some time reading the tasks or familiarizing with the tool. However, we think that idle working times, times spent on other tasks, or offline work can largely be excluded due to the limited experiment time of 180 minutes in which the participants needed to work in a concentrated manner in order to get the work done.

The number of participants (48 students) may also affect the statistical validity of the results.

Internal Validity. The internal validity refers to the extent to which treatment or independent variables caused the effects seen on the dependent variables.

In order to reduce this kind of validity threats, we made sure that the participants of both groups had at least medium experience in programming and design – with slight differences – and that they were aware of the architectural design patterns they had to use for making and documenting architectural decisions (they had also implemented some of them during the practical course before the experiment).

Also, the experiment was carried out in a controlled environment and the group members were in different rooms and did not know the identity of their partner. An observer in the room ensured that no interactions between the participants of the same room occurred. The students did not know the goals of the experiment or the group they belong to, nor could they realize that the control and experiment groups were working with different versions of CoCoADvISE.

Our inability, to effectively prevent the participants from using external Web sites (i.e., search engines, social networks, chats, wikis, etc.) during the experiment, might also pose a potential – but arguably negligible – threat to validity. We believe, that it is very unlikely, that any of these external Web sites might have been advantageous in actually preventing constraint violations.

A threat to validity was introduced by the execution of the experiment in two different sessions. To limit this threat, we did not allow any access to CoCoADvISE and the accompanying materials outside these two sessions.

Construct Validity. The construct validity focuses on the suitability of the experiment design for the theory behind the experiment and the observations.

The students worked in their task assignment only on a single software system with an excerpt of the full list of requirements. However, it is likely that this did not (heavily) affect the validity of the results because the constraints in the collaboration were our focus. We regard the amount and type of the constraints introduced in the experiment to be grounded in real-life collaborative architectural decision making scenarios.

The variables that have been observed in the experiment are regarded as accurate and objective as they are related to the actual use of tools and were automatically extracted from the database.

Also, for calculating the completed work per participant, we first extracted a list of required tasks from the exercise description which was afterwards used to calculate the completion of work automatically from the database entries.

External Validity. The external validity is concerned with whether the results are generalizable outside the scope of our study.

The subjects of the experiment have medium programming experience and were familiar with the architectural design decisions they were asked to make. However, only few students have experience in the industry. We hence consider the group under study to be representative for novice software developers or architects and plan to test the same hypotheses with other target groups as well.

Kitchenham et al. regard students close to practitioners, as they are considered to be the next generation of software professionals [17].

As mentioned before, the measurements were extracted from the tool database, avoiding any bias by the experimenters.

The system under study and the corresponding architectural decision models and patterns are representative for Web and enterprise information systems, and hence it is likely that the findings can be generalized to similar system domains and decision models. It would require additional experiments to determine if they can be generalized to vastly different system domains, such as software systems operating close to the hardware.

Finally, in our experiment we observed group decision making with groups of only two members. In our point of view, it is highly likely that the results are similar for slightly larger groups (e.g., of 3 or 4 members). However, it is unclear, if the results can be generalized to larger groups of decision makers.

8.3. Inferences

In principle, our observations are coherent with the findings of similar studies in slightly different contexts. For instance, Herbsleb et al. present a theory that models collaborative software engineering as a distributed constraint satisfaction problem [12]. They also found that backtracking, as a result of constraint violations, increases both the time and effort to be invested. This is a further indication that our findings should be generalizable. In particular, we believe that virtually any kind of collaborative process that concerns different stakeholder roles and demands to be restricted by various domain and context specific constraints will benefit from automatic constraint enforcement in a similar way to CoCoADvISE.

9. Conclusions and Future Work

The approach presented in this paper is the first one to consider the precise definition and automatic enforcement of constraints in real-time collaborative architectural decision making. CoCoADvISE ensures that stakeholders with different roles make and document collaborative architectural design decisions consistently. We demonstrate the applicability of our approach in an industrial context and with the help of a controlled experiment we are also able to report strong evidence that the automatic enforcement of constraints leads to increased time and effort related efficiency and effectiveness of the users while making and documenting architectural decisions.

We consider our approach and accompanying tool to be relevant and useful for other collaborative software engineering tools as well, which involve various stakeholder roles and distributed teams. Therefore, we plan to extend CoCoADvISE to cover other constrainable collaborative activities with focus on software architecture processes.

In our future work, we will also collect more empirical evidence about the supportive effect of automatic enforcement of constraints in collaborative architectural decision making tools on the efficiency and effectiveness of users, by

conducting similar controlled experiments. Our main goal is to test our assumptions with practitioners, receive feedback regarding the usability of our tool, and test our approach with different group sizes, in different system domains, and with different decision models.

Acknowledgments

We would like to thank all students of the Information System Technologies lecture in the Winter Semester 2013/2014 for participating in the experiment.

Appendix A. Generic Meta-model for Decision Making Constraints

This appendix provides the complete formal definition of the Constrainable Decision Meta-model introduced in Section 4.3. In particular, Definition A.1 provides a list of elements and their relations, Definition A.2 presents crucial model invariants to be considered at design time, and Definition A.3 lists model invariants relevant at execution time.

To provide a self-contained view in this paper, the following formal meta-model repeats the core definitions regarding the concepts of subjects, roles and permissions from [49], which form the basis for our approach.

Definition A.1 (Constrainable Decision Meta-model)
A Constrainable Decision Model $CDM = (E, M)$ where $E = DM \cup D \cup Q \cup O \cup S \cup R \cup P \cup DD \cup O_S \cup C$ refers to pairwise disjoint sets of the meta-model and $M = dma \cup qda \cup oqa \cup rsa \cup pra \cup dda \cup gs \cup gr \cup soa \cup ss \cup sr \cup cda \cup cma \cup rruca \cup ssuca \cup rrrca \cup srsca \cup prpca \cup cd$ to mappings that establish relationships, such that:

- For the sets of the meta-model:
  - An element of DM is called *Decision Model*. $DM \neq \emptyset$.
  - An element of D is called *Decision*. $D \neq \emptyset$.
  - An element of Q is called *Question*. $Q \neq \emptyset$.
  - An element of O is called *Option*. $O \neq \emptyset$.
  - An element of S is called *Subject*. $S \neq \emptyset$.
  - An element of R is called *Role*. $R \neq \emptyset$.
  - An element of P is called *Permission*. $P \supseteq \{selectOption, generateDecision\}$.
  - An element of DD is called *Documented Decision*. $DD \neq \emptyset$.
  - An element of O_S is called *Option Selection*.
  - An element of C is called *Constraint*. $C = C_U \cup C_{U_R} \cup C_{U_S} \cup C_{R_R} \cup C_{R_S} \cup C_P$
  - An element of C_U is called *Unanimity Constraint*.

- An element of C_{U_R} is called *Role Unanimity Constraint.*
- An element of C_{U_S} is called *Subject Unanimity Constraint.*
- An element of C_{R_R} is called *Responsible Role Constraint.*
- An element of C_{R_S} is called *Responsible Subject Constraint.*
- An element of C_P is called *Required Permission Constraint.*

In the list below, we iteratively define the partial mappings of the Decision Making Constraint Model and provide corresponding formalizations ($\mathcal{P}$ refers to the power set):

1. A decision model consists of many decisions and each decision belongs to exactly one decision model.
   Formally: The *injective* mapping $dma : DM \mapsto \mathcal{P}(D)$ is called **decision-to-decision-model assignment**. For $dma(dm) = D_{dm}$ we call $dm \in DM$ *decision model* and $D_{dm} \subseteq D$ is called the set of *decisions assigned to dm*. The mapping $dma^{-1} : D \mapsto DM$ returns the decision model a decision is assigned to.
2. A decision consists of many questions and each question belongs to exactly one decision.
   Formally: The *injective* mapping $qda : D \mapsto \mathcal{P}(Q)$ is called **questions-to-decision assignment**. For $qda(d) = Q_d$ we call $d \in D$ *decision* and $Q_d \subseteq Q$ is called the set of *questions assigned to d*. The mapping $qda^{-1} : Q \mapsto D$ returns the decision a question is assigned to.
3. A question provides many options and each option belongs to exactly one question.
   Formally: The *injective* mapping $oqa : Q \mapsto \mathcal{P}(O)$ is called **option-to-question assignment**. For $oqa(q) = O_q$ we call $q \in Q$ *question* and $O_q \subseteq O$ is called the set of *options assigned to q*. The mapping $oqa^{-1} : O \mapsto Q$ returns the question an option is assigned to.
4. Roles are assigned to subjects (i.e., human users), and through their roles the subjects acquire the rights to perform certain tasks (see [49]). The role-to-subject assignment relation is a many-to-many relation, so that each subject may own several roles and each role can be assigned to different subjects. For example, in case the "Software Architect" role is assigned to two subjects called Alice and Bob, both can perform all tasks assigned to the "Software Architect" role.
   Formally: The *injective* mapping $rsa : S \mapsto \mathcal{P}(R)$ is called **role-to-subject assignment**. For $rsa(s) = R_s$ we call $s \in S$ *subject* and $R_s \subseteq R$ the set of *roles assigned to this subject* (the set of roles owned by s). The mapping $rsa^{-1} : R \mapsto \mathcal{P}(S)$ returns all subjects assigned to a role (the set of subjects owning a role).
5. Permissions are assigned to roles. The permission-to-role assignment relation is a many-to-many relation, so that each role may own several permissions and each permission can be assigned to different roles.

Formally: The *injective* mapping $pra : R \mapsto \mathcal{P}(P)$ is called **permission-to-role assignment**. For $pra(r) = P_r$ we call $r \in R$ *role* and $P_r \subseteq P$ the set of *permissions assigned to this role* (the set of permissions owned by r). The mapping $pra^{-1} : P \mapsto \mathcal{P}(R)$ returns all roles assigned to a permission (the set of roles owning a permission).

6. At runtime, users can generate documented decisions which are based on a reusable decision. Thus, when a user generates a documented decision, a new documented decision is assigned to the corresponding (reusable) decision (see Figure 2).
   Formally: The mapping $dda : D \mapsto \mathcal{P}(DD)$ is called **documented-decision-to-decision assignment**. For $dda(d) = DD_d$ we call $d \in D$ *decision* and $DD_d \subseteq DD$ is called the set of *documented decisions assigned to d*.
7. As defined in Definition A.1.6, documented decisions are assigned to decisions whenever users generate documented decisions. The generating-subject mapping is used to hold the exact subject that generated a documented decision.
   Formally: The mapping $gs : DD \mapsto S$ is called **generating-subject mapping**. For $gs(dd) = s$ we call $s \in S$ the *generating subject* and $dd \subseteq DD_S$ is called the *documented decision*.
8. Similarly to Definition A.1.7, we define the role that is used to generate a documented decision to be called the generating-role of the corresponding documented decision.
   Formally: The mapping $gr : DD \mapsto R$ is called **generating-role mapping**. For $gr(dd) = r$ we call $r \in R$ the *generating role* and $dd \subseteq DD_S$ is called the *documented decision*.
9. At runtime, users of a decision model can select options. Thus, when a user selects an option, a new option selection is assigned to the corresponding option (see Figure 6(b)).
   Formally: The mapping $soa : O \mapsto \mathcal{P}(O_S)$ is called **selection-to-option assignment**. For $soa(o) = O_{S_o}$ we call $o \in O$ *option* and $O_{S_o} \subseteq O_S$ is called the set of *option selections assigned to o*.
10. As defined in Definition A.1.9, option selections are assigned to options whenever users select options. The purpose of an option selection is to hold the subject and role that is used to select a certain option. For example, if subject Alice selects the option "only JSON", the corresponding option selection holds a reference to the subject "Alice".
    Formally: The mapping $ss : O_S \mapsto S$ is called **selecting-subject mapping**. For $ss(os) = s$ we call $s \in S$ the *selecting subject* and $os \subseteq O_S$ is called the *option selection*.
11. Similarly to Definition A.1.10, we define the role that is used to select a certain option to be called the selecting-role of the corresponding option selection.
    Formally: The mapping $sr : O_S \mapsto R$ is called **selecting-role mapping**. For $sr(os) = r$ we call $r \in R$ the *selecting role* and $os \subseteq O_S$ is called the *option selection*.

12. Particular reusable architectural decisions can be made subject to decision making constraints. For example, the decision "RESTful HTTP vs. SOAP/WS-*" (see Figure 3), which is required to be decided unanimously by all stakeholders, may be made subject to a unanimity constraint. More precisely, decisions are made subject to constraints by assigning them to constraints.
    Formally: The *injective* mapping $cda : D \mapsto \mathcal{P}(C)$ is called **constraint-to-decision assignment**. For $cda(d) = C_d$ we call $d \in D$ *decision* and $C_d \subseteq C$ is called the set of *constraints assigned to d*. The mapping $cda^{-1} : C \mapsto \mathcal{P}(D)$ returns the set of decisions a constraint is assigned to.
13. All reusable architectural decisions of a decision model can be made subject to decision making constraints at once. For instance, all decisions of a decision model can statically be made subject to a Required Permission constraint by assigning the corresponding constraint to the decision model.
    Formally: The *injective* mapping $cma : DM \mapsto \mathcal{P}(C)$ is called **constraint-to-decision-model assignment**. For $cma(dm) = C_{dm}$ we call $dm \in DM$ *decision model* and $C_{dm} \subseteq C$ is called the set of *constraints assigned to dm*. The mapping $cma^{-1} : C \mapsto \mathcal{P}(DM)$ returns the set of decision models a constraint is assigned to.
14. A Role Unanimity Constraint enforces that all subjects that own at least one of a specific set of roles have to unanimously agree on the same set of options, at runtime. Thus, these roles are assigned to role unanimity constraints. The role-to-role-unanimity-constraint assignment relation is a many-to-many relation, so that each role may be assigned to several role unanimity constraints and each role unanimity constraint can be assigned to different roles.
    Formally: The *injective* mapping $rruca : C_{U_R} \mapsto \mathcal{P}(R)$ is called **role-to-role-unanimity-constraint assignment**. For $rruca(ruc) = R_{ruc}$ we call $ruc \in C_{U_R}$ *role unanimity constraint* and $R_{ruc} \subseteq R$ the set of *roles assigned to this role unanimity constraint*. The mapping $rruca^{-1} : R \mapsto \mathcal{P}(C_{U_R})$ returns all role unanimity constraints assigned to a role.
15. A Subject Unanimity Constraint enforces that a specific set of subjects have to unanimously agree on the same set of options, at runtime. Thus, these subjects are assigned to subject unanimity constraints. The subject-to-subject-unanimity-constraint assignment relation is a many-to-many relation, so that each subject may be assigned to several subject unanimity constraints and each subject unanimity constraint can be assigned to different subjects.
    Formally: The *injective* mapping $ssuca : C_{U_S} \mapsto \mathcal{P}(S)$ is called **subject-to-subject-unanimity-constraint assignment**. For $ssuca(suc) = S_{suc}$ we call $suc \in C_{U_S}$ *subject unanimity constraint* and $S_{suc} \subseteq S$ the set of *subjects assigned to this subject unanimity constraint*. The mapping $ssuca^{-1} : S \mapsto \mathcal{P}(C_{U_S})$ returns all subject unanimity constraints assigned to a subject.
16. A Responsible Role Constraint enforces that only subjects that own at

least one of a specific set of roles shall be allowed to make and generate a specific decision. Thus, these roles are assigned to responsible role constraints. The role-to-responsible-role-constraint assignment relation is a many-to-many relation, so that each role may be assigned to several responsible role constraints and each responsible role constraint can be assigned to different roles.

Formally: The *injective* mapping $rrrca : C_{R_R} \mapsto \mathcal{P}(R)$ is called **role-to-responsible-role-constraint assignment**. For $rrrca(rrc) = R_{rrc}$ we call $rrc \in C_{R_R}$ *responsible role constraint* and $R_{rrc} \subseteq R$ the set of *roles assigned to this responsible role constraint.* The mapping $rrrca^{-1} : R \mapsto \mathcal{P}(C_{R_R})$ returns all responsible role constraints assigned to a role.

17. A Responsible Subject Constraint enforces that only a specific set of subjects shall be allowed to make and generate a specific decision. Thus, these subjects are assigned to responsible subject constraints. The subject-to-responsible-subject-constraint assignment relation is a many-to-many relation, so that each subject may be assigned to several responsible subject constraints and each responsible subject constraint can be assigned to different subjects.

    Formally: The *injective* mapping $srsca : C_{R_S} \mapsto \mathcal{P}(S)$ is called **subject-to-responsible-subject-constraint assignment**. For $srsca(rsc) = S_{rsc}$ we call $rsc \in C_{R_S}$ *responsible subject constraint* and $S_{rsc} \subseteq S$ the set of *subjects assigned to this responsible subject constraint.* The mapping $srsca^{-1} : S \mapsto \mathcal{P}(C_{R_S})$ returns all responsible subject constraints assigned to a subject.
18. A Required Permission Constraint enforces that only subjects that own certain permissions (i.e., via their roles) within the system are allowed to perform the corresponding activities. For example, consider permission "Generate Decision" is only assigned to role "Software Architect", then only subjects that own the role "Software Architect" shall be allowed to generate decisions. The permissions to be enforced have to be assigned to required permission constraints. The permission-to-required-permission-constraint assignment relation is a many-to-many relation, so that each permission may be assigned to several required permission constraints and each required permission constraint can be assigned to different permissions.

    Formally: The *injective* mapping $prpca : C_P \mapsto \mathcal{P}(P)$ is called **permission-to-required-permission-constraint assignment**.

    For $prpca(rpc) = P_{rpc}$ we call $rpc \in C_P$ *required permission constraint* and $P_{rpc} \subseteq P$ the set of *permissions assigned to this required permission constraint.* The mapping $prpca^{-1} : P \mapsto \mathcal{P}(C_P)$ returns all required permission constraints assigned to a permission.
19. A decision can effectively be constrained either by a constraint that is directly assigned to the decision (i.e., using a constraint-to-decision assignment, see, Definition A.1.12), or by a constraint that is assigned to the corresponding decision model (i.e., using a constraint-to-decision-model assignment, see, Definition A.1.13). The constrained-decision mapping

aggregates the set of decisions that are constrained by a constraint. Formally: The mapping $cd : C \mapsto \mathcal{P}(D)$ is called **constrained-decision mapping**, such that for each constraint $c \in C$ the set of decisions that are effectively constrained by c are returned, i.e., $cd(c) = dma(cma^{-1}(c)) \cup cda^{-1}(c)$.

Definition A.2 (Design Time *CDM* Meta-model Invariants)
Let $CDM = (E, M)$ be a Constrainable Decision Model. CDM is said to be statically correct if the following requirements hold:

1. A constraint either constrains a single decision or a complete decision model. It is either assigned to a decision or a decision model. Therefore:

$$\forall c \in C(cda^{-1}(c) = \emptyset \oplus cma^{-1}(c) = \emptyset)$$

   Note that the $\oplus$ symbol represents the XOR (i.e., exclusive or) operation.

2. All roles that are assigned to a responsible role constraint must own the permission "generate decision". Therefore:

$$\forall r \in R(rrrca^{-1}(r) \neq \emptyset \Rightarrow generateDecision \in pra(r))$$

3. All subjects that are assigned to a responsible subject constraint must own a role that owns the permission "generate decision". Therefore:

$$\forall s \in S(srsca^{-1}(s) \neq \emptyset \Rightarrow \exists r \in rsa(s)(generateDecision \in pra(r)))$$

4. Each permission that is assigned to a required permission constraint must be owned by at least one role. Therefore:

$$\forall c \in C_P, p \in prpca(c)(pra^{-1}(p) \neq \emptyset)$$

5. When an unanimity constraint is used there must be at least one subject that has the permission "select option". Therefore:

$$\forall c \in C_U \Rightarrow \exists r \in R(selectOption \in pra(r))$$

6. All roles that are assigned to role unanimity constraints must have the permission "select option". Therefore:

$$\forall r \in R(rruca^{-1}(r) \neq \emptyset \Rightarrow selectOption \in pra(r))$$

7. All subjects that are assigned to subject unanimity constraints must have the permission "select option". Therefore:

$$\forall s \in S(ssuca^{-1}(s) \neq \emptyset \Rightarrow \exists r \in rsa(s)(selectOption \in pra(r)))$$

8. Subject unanimity constraints and role unanimity constraints may potentially conflict. More precisely, if both constraint types are used simultaneously, it is required that each subject assigned to the subject unanimity constraint owns each role assigned to the role unanimity constraint. Therefore:

$$\begin{aligned}&\forall dm \in DM(\forall d \in dma(dm)(\forall c_r \in cda(d) \cup cma(dm)(\\&\forall c_s \in cda(d) \cup cma(dm)(c_r \in C_{U_R} \wedge c_s \in C_{U_S}\\&\Rightarrow \forall s \in ssuca(c_s)(rsa(s) \supseteq rruca(c_r))))))\end{aligned}$$

Definition A.3 (Execution Time *CDM* Meta-model Invariants) Let $CDM = (E, M)$ be a Constrainable Decision Model. CDM is said to be dynamically correct if the following requirements hold:

1. For each option selection, the selecting subject must own the corresponding selecting role. Therefore:

$$\forall os \in O_S(sr(os) \in rsa(ss(os)))$$

2. A subject may only select a single option for each question. Therefore:

$$\begin{aligned}&\forall q \in Q(\forall o \in oqa(q)(\forall os_x \in soa(o)(\forall os_y \in soa(o)(\\&os_x \neq os_y \wedge \exists ss(os_x) \Rightarrow ss(os_x) \neq ss(os_y)))))\end{aligned}$$

3. All subjects must choose an option for each question. Therefore:

$$\forall q \in Q(\forall s \in S(\exists o \in oqa(q)(\exists os \in soa(o)(ss(os) = s))))$$

4. All decision making constraint type definitions (i.e., CT_1 through CT_6, see Section 4.3) must be met. Therefore:

$$\forall dm \in DM(CT_1 \wedge CT_2 \wedge CT_3 \wedge CT_4 \wedge CT_5 \wedge CT_6)$$

References

[1] N. D. Anh and D. S. Cruzes. Coordination of software development teams across organizational boundary – an exploratory study. In *8th IEEE International Conference on Global Software Engineering (ICGSE)*, pages 216–225, Aug 2013.

[2] M. A. Babar and I. Gorton. A Tool for Managing Software Architecture Knowledge. In *Proceedings of the Second Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent*, SHARK-ADI'07, Washington, DC, USA, 2007. IEEE Computer Society.

[3] M. Cantor and J. D. Sanders. Operational IT governance. Technical report, IBM developerWorks, 2007.

[4] V. Clerc, E. de Vries, and P. Lago. Using wikis to support architectural knowledge management in global software development. In *Proceedings of the 2010 ICSE Workshop on Sharing and Reusing Architectural Knowledge*, pages 37–43. ACM, 2010.

[5] Y. Dubinsky, A. Yaeli, and A. Kofman. Effective Management of Roles and Responsibilities: Driving Accountability in Software Development Teams. *IBM J. Res. Dev.*, 54(2):173–183, Mar. 2010.

[6] J. Eckstein. *Agile Software Development with Distributed Teams: Staying Agile in a Global World*. Dorset House, 2010.

[7] R. Farenhorst, P. Lago, and H. Van Vliet. Effective Tool Support for Architectural Knowledge Sharing. In *Proceedings of the First European Conference on Software Architecture*, ECSA'07, pages 123–138, Berlin, Heidelberg, 2007. Springer-Verlag.

[8] P. Gaubatz and U. Zdun. Supporting Entailment Constraints in the Context of Collaborative Web Applications. In *28th Symposium On Applied Computing*, pages 736–741, USA, March 2013. ACM.

[9] M. Goldman, G. Little, and R. C. Miller. Real-time collaborative coding in a web ide. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST'11, pages 155–164, New York, NY, USA, 2011. ACM.

[10] N. B. Harrison, P. Avgeriou, and U. Zdun. Using Patterns to Capture Architectural Decisions. *IEEE Software*, 24(4):38–45, 2007.

[11] J. D. Herbsleb and R. E. Grinter. Architectures, Coordination, and Distance: Conway's Law and Beyond. *IEEE Software*, 16(5):63–70, Sept. 1999.

[12] J. D. Herbsleb, A. Mockus, and J. A. Roberts. Collaboration in software engineering projects: A theory of coordination. In *Proceedings of the International Conference on Information Systems (ICIS)*, page 38, 2006.

[13] A. Jansen and J. Bosch. Software Architecture as a Set of Architectural Design Decisions. In *5th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 109–120. IEEE Computer Society, 2005.

[14] C. Jensen and W. Scacchi. Governance in Open Source Software Development Projects: A Comparative Multi-level Analysis. In *Open Source Software: New Horizons*, volume 319 of *IFIP Advances in Information and Communication Technology*, pages 130–142. Springer Berlin Heidelberg, 2010.

[15] M. Jensen and S. Feja. A security modeling approach for web-service-based business processes. In *16th Annual IEEE International Conference on the Engineering of Computer Based Systems (ECBS'09)*, pages 340–347, 2009.

[16] M. Kalumbilo. Effective Specification of Decision Rights and Accountabilities for Better Performing Software Engineering Projects. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE'12, pages 1503–1506, Piscataway, NJ, USA, 2012. IEEE Press.

[17] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. E. Emam, and J. Rosenberg. Preliminary Guidelines for Empirical Research in Software Engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, Aug. 2002.

[18] A. Kofman, A. Yaeli, T. Klinger, and P. Tarr. Roles, Rights, and Responsibilities: Better Governance Through Decision Rights Automation. In *Proceedings of the 2009 ICSE Workshop on Software Development Governance*, SDG'09, pages 9–14, Washington, DC, USA, 2009. IEEE Computer Society.

[19] P. Kruchten. An Ontology of Architectural Design Decisions. In *Proceedings of 2nd Workshop on Software Variability Management*, pages 54–61, 2004.

[20] P. Liang, A. Jansen, and P. Avgeriou. Knowledge Architect: A Tool Suite for Managing Software Architecture Knowledge. Technical report, University of Groningen, 2009.

[21] I. Lytra, S. Sobernig, and U. Zdun. Architectural Decision Making for Service-Based Platform Integration: A Qualitative Multi-Method Study. In *Joint 10th Working IEEE/IFIP Conference on Software Architecture & 6th European Conference on Software Architecture (WICSA/ECSA), Helsinki, Finland*, pages 111–120. IEEE Computer Society, 2012.

[22] I. Lytra, H. Tran, and U. Zdun. Supporting Consistency Between Architectural Design Decisions and Component Models Through Reusable Architectural Knowledge Transformations. In *Proceedings of the 7th European Conference on Software Architecture (ECSA)*, ECSA'13, pages 224–239, Berlin, Heidelberg, 2013. Springer-Verlag.

[23] A. MacLean, R. Young, V. Bellotti, and T. Moran. Questions, Options, and Criteria: Elements of Design Space Analysis. *Human-Computer Interaction*, 6:201–250, 1991.

[24] P. Maheshwari and A. Teoh. Supporting ATAM with a collaborative Web-based software architecture evaluation tool. *Science of Computer Programming*, 57(1):109–128, 2005.

[25] H. B. Mann and W. D. R. On a Test of Whether One of Two Random Variables is Stochastically Larger than the Other. *Annals of Mathematical Statistics*, 18(1):50–60, 1947.

[26] C. Mayr, U. Zdun, and S. Dustdar. Reusable Architectural Decision Model for Model and Metadata Repositories. In F. de Boer, M. Bonsangue, and

E. Madelaine, editors, *Formal Methods for Components and Objects*, volume 5751 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 2009.

[27] C. Miesbauer and R. Weinreich. Classification of Design Decisions: An Expert Survey in Practice. In *Proceedings of the 7th European Conference on Software Architecture*, ECSA'13, pages 130–145, Berlin, Heidelberg, 2013. Springer-Verlag.

[28] A. Nakakawa, P. v. Bommel, and H. A. Proper. Challenges of involving stakeholders when creating enterprise architecture. In B. v. Dongen and H. Reijers, editors, *Proceedings of the 5th SIKS/BENAIS Conference on Enterprise Information Systems (EIS-2010), Eindhoven, The Netherlands*, pages 43–55, November 2010.

[29] A. Nakakawa, P. van Bommel, and H. A. Proper. Supplementing Enterprise Architecture Approaches with Support for Executing Collaborative Tasks - a Case of TOGAF ADM. *International Journal of Cooperative Information Systems*, 22(2), 2013.

[30] R. Nord, P. C. Clements, D. Emery, and R. Hilliard. A Structured Approach for Reviewing Architecture Documentation (CMU/SEI-2009-TN-030). Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 2009.

[31] M. Nowak and C. Pautasso. Team Situational Awareness and Architectural Decision Making with the Software Architecture Warehouse. In *7th European Conference on Software Architecture*, ECSA'13, pages 146–161, Berlin, Heidelberg, 2013. Springer-Verlag.

[32] M. Nowak, C. Pautasso, and O. Zimmermann. Architectural Decision Modeling with Reuse: Challenges and Opportunities. In *Proceedings of the 2010 ICSE Workshop on Sharing and Reusing Architectural Knowledge*, SHARK'10, pages 13–20, New York, NY, USA, 2010. ACM.

[33] Object Management Group. OMG Unified Modeling Language (OMG UML): Superstructure (Version 2.2). URL: `http://www.omg.org/spec/UML/2.2/`, February 2009.

[34] P. Ovaska, M. Rossi, and P. Marttiin. Architecture as a Coordination Tool in Multi-site Software Development. *Software Process: Improvement and Practice*, 8(4):233–247, Oct./Dec. 2003. Special Issue: Global Software Development: Growing Opportunities, Ongoing Challenges.

[35] E. A. Peña and E. H. Slate. Global Validation of Linear Model Assumptions. *Journal of the American Statistical Association*, 101(473):341–354, 2006.

[36] B. Raadt, S. Schouten, and H. Vliet. Stakeholder Perception of Enterprise Architecture. In *Proceedings of the 2nd European Conference on Software Architecture*, ECSA'08, pages 19–34, Berlin, Heidelberg, 2008. Springer.

[37] V. S. Rekha and H. Muccini. A Study on Group Decision-Making in Software Architecture. In *IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 185–194, 2014.

[38] S. Rekha V. and H. Muccini. Suitability of Software Architecture Decision Making Methods for Group Decisions. In *Software Architecture*, volume 8627 of *Lecture Notes in Computer Science*, pages 17–32. Springer International Publishing, 2014.

[39] N. Schuster, O. Zimmermann, and C. Pautasso. ADkwik: Web 2.0 Collaboration System for Architectural Decision Engineering. In *SEKE*, pages 255–260. Knowledge Systems Institute Graduate School, 2007.

[40] M. Shahin, P. Liang, and M. Khayyambashi. Architectural design decision: Existing models and tools. In *Software Architecture, 2009 European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*, pages 293–296, Sept 2009.

[41] M. Shahin, P. Liang, and M. R. Khayyambashi. Improving Understandability of Architecture Design Through Visualization of Architectural Design Decision. In *Proceedings of the 2010 ICSE Workshop on Sharing and Reusing Architectural Knowledge*, SHARK'10, pages 88–95, New York, NY, USA, 2010. ACM.

[42] M. Shahin, P. Liang, and Z. Li. Architectural Design Decision Visualization for Architecture Design: Preliminary Results of A Controlled Experiment. In *Proceedings of the 1st Workshop on Traceability, Dependencies and Software Architecture (TDSA)*, pages 5–12. ACM, 2011.

[43] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 3(52), 1965.

[44] Siemens AG, Politecnico di Milano, Telcordia, and TU Vienna. D3.2 Architecture for Role-Based Governance of Virtual Service Platforms. Technical report, INDENICA Project, February 2012.

[45] K. Smolander and T. Päivärinta. Describing and Communicating Software Architecture in Practice: Observations on Stakeholders and Rationale. In *Proceedings of the 14th International Conference on Advanced Information Systems Engineering*, CAiSE'02, pages 117–133, London, UK, 2002. Springer.

[46] R. M. Smullyan. *First-order logic*, volume 21968. Springer, 1968.

[47] M. Strembeck. Scenario-driven Role Engineering. *IEEE Security & Privacy*, 8(1), January/February 2010.

[48] M. Strembeck and J. Mendling. Generic algorithms for consistency checking of mutual-exclusion and binding constraints in a business process context. In *Proceedings of the 18th International Conference on Cooperative Information Systems (CoopIS)*, pages 204–221, 2010.

[49] M. Strembeck and J. Mendling. Modeling Process-related RBAC Models with Extended UML Activity Models. *Information and Software Technology*, 53(5):456–483, May 2011.

[50] K. Tan, J. Crampton, and C. A. Gunter. The consistency of task-based authorization constraints in workflow systems. In *17th IEEE workshop on Computer Security Foundations (WCSF)*, pages 155–169, 2004.

[51] R. C. Team et al. *R: A language and environment for statistical computing*, 2005.

[52] The Open Group. TOGAF 9 - The Open Group Architecture Framework Version 9, 2009.

[53] D. Tofan and M. Galster. Capturing and making architectural decisions: An open source online tool. In *Proceedings of the 2014 European Conference on Software Architecture Workshops*, ECSAW'14, pages 33:1–33:4, New York, NY, USA, 2014. ACM.

[54] D. Tofan, M. Galster, and P. Avgeriou. Difficulty of Architectural Decisions – A Survey with Professional Architects. In *7th European Conference on Software Architecture*, ECSA'13, pages 192–199, 2013.

[55] D. Tofan, M. Galster, P. Avgeriou, and W. Schuitema. Past and future of software architectural decisions – A systematic mapping study. *Information and Software Technology*, 56(8):850–872, 2014.

[56] J. Tyree and A. Akerman. Architecture Decisions: Demystifying Architecture. *IEEE Software*, 22(2):19–27, 2005.

[57] A. van Deursen, A. Mesbah, B. Cornelissen, A. Zaidman, M. Pinzger, and A. Guzzi. Adinda: A knowledgeable, browser-based ide. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE'10, pages 203–206, New York, NY, USA, 2010. ACM.

[58] U. van Heesch, P. Avgeriou, and A. Tang. Does decision documentation help junior designers rationalize their decisions? A comparative multiple-case study. *Journal of Systems and Software*, 86(6):1545–1565, 2013.

[59] U. van Heesch, P. Avgeriou, U. Zdun, and N. Harrison. The supportive effect of patterns in architecture decision recovery – A controlled experiment. *Science of Computer Programming*, 77(5):551–576, 2012.

[60] J. Whitehead. Collaboration in software engineering: A roadmap. In *2007 Future of Software Engineering*, FOSE'07, pages 214–225, Washington, DC, USA, 2007. IEEE Computer Society.

[61] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[62] C. Wolter, M. Menzel, A. Schaad, P. Miseldine, and C. Meinel. Model-driven business process security requirement specification. *Journal of Systems Architecture*, 55:211–223, 2009.

[63] C. Wolter, A. Schaad, and C. Meinel. Task-based entailment constraints for basic workflow patterns. In *13th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 51–60. ACM, June 2008.

[64] U. Zdun and M. Strembeck. Reusable Architectural Decisions for DSL Design: Foundational Decisions in DSL Development. In *Proceedings of 14th European Conference on Pattern Languages of Programs (EuroPLoP)*, pages 1–37, Irsee, Germany, July 2009.

[65] O. Zimmermann. Architectural Decisions as Reusable Design Assets. *IEEE Software*, 28(1):64–69, 2011.

[66] O. Zimmermann, T. Gschwind, J. Küster, F. Leymann, and N. Schuster. Reusable Architectural Decision Models for Enterprise Application Development. In *3rd International Conference on Quality of Software Architectures (QoSA)*, pages 15–32. Springer, 2007.

[67] O. Zimmermann, J. Koehler, and L. Frank. Architectural Decision Models as Micro-Methodology for Service-Oriented Analysis and Design. In D. Lübke, editor, *Proceedings of the Workshop on Software Engineering Methods for Service-oriented Architecture 2007 (SEMSOA 2007), Hannover, Germany, online CEUR-WS.org/Vol-244/*, pages 46–60, May 2007.

[68] O. Zimmermann, J. Koehler, F. Leymann, R. Polley, and N. Schuster. Managing architectural decision models with dependency relations, integrity constraints, and production rules. *Journal of Systems and Software*, 82(8):1249–1267, Aug. 2009.

[69] O. Zimmermann, U. Zdun, T. Gschwind, and F. Leymann. Combining Pattern Languages and Reusable Architectural Decision Models into a Comprehensive and Comprehensible Design Method. In *7th Working IEEE/IFIP Conference on Software Architecture (WICSA), Vancouver, BC, Canada*, pages 157–166. IEEE Computer Society, 2008.