Decision Buddy: Tool Support for Constraint-Based Design Decisions during System Evolution

Sebastian Gerdes, Mohamed Soliman, Matthias Riebisch
Department of Informatics, University of Hamburg
Hamburg, Germany
{gerdes, riebisch, soliman}@informatik.uni-hamburg.de

ABSTRACT

Designing a software architecture is a highly complex task and associated with a high degree of uncertainty. There are a variety of reusable and established solutions, but they differ in their impact on the system's functionality and quality. The architect has to consider different aspects like stakeholders' requirements as well as numerous constraints coming, among others, from the technical environment and organization. The context of software evolution sheds a different light on constraints. The existing system with its structure based on previous decisions is a limiting factor constraining the ongoing development. However, current approaches do not sufficiently consider constraints induced by an existing system until now. To assist the architect in taking the right design decisions efficiently, tool support for the recommendation of solutions and structured documentation of the design decisions are indispensable. In our paper, we propose a decision process focusing on the consideration of constraints in evolving systems. Furthermore, we introduce our tool Decision Buddy and show how it contributes to the application of our constraint-based decision process.

Keywords

software architecture, design decisions, constraints, software evolution, architecture knowledge, design assistant, tools

1. INTRODUCTION

Large software systems are composed of multiple, interdependent components and subsystems, which make them inherently complex. Effective means to handle this complexity are software architectures, which play an important role in fulfilling customers' requirements and particularly the quality requirements of a system. One key aspect in designing a sustainable software architecture is to take the right design decisions, which is a major challenge in the field of software engineering. The architect needs to reason about the architecture, to deliberate on different design alternatives, and to balance cross-cutting concerns in order to make sound design choices. The important role of design decisions led to a paradigm shift, which perceives a software architecture as a composition of architectural design decisions (ADD) [7] instead of limiting the description of software architecture to its underlying structure based on components and connectors. Certainly, capturing these decisions, which are part of the architecture knowledge, has proven to be difficult. Part of it is tacit as it is just in the mind of the architect, and thus is exposed to so-called knowledge vaporization inducing architectural erosion. Over the past decade, the research community proposed various approaches and tools in the field of architecture knowledge dealing with the capturing of design decisions [20], the documentation of the rationale and the assistance in taking decisions [18, 4].

It is no secret that the architect relies on existing solutions when taking decisions. There is a lot of established and well-known solutions like design patterns, architectural styles, and reusable technology solutions, such as frameworks, or COTS-components. But it is hard to decide for the right solution for the problem to solve, which might result in selecting inapplicable architectural solutions and thus increasing the risk of fail. There are multiple factors like uncertainty about advantages and disadvantages of solutions, costs, and implications of a solution, which influence the decision, also known as forces [22]. However, a survey of practitioners conducted by Tang *et al.* revealed that design constraints are one of the most important factors in design rationale [17]. Design constraints limit the solution space – the more constraints there are, the smaller the degree of freedom when deciding for a solution. Tang *et al.* classified constraints into four different categories as they can come from different sources: requirement-related, quality requirement-related, contextual and solution-related constraints [19].

Within the context of large systems, the majority of today's software engineering efforts are spent on continuous and evolutionary development [23], which sheds a different light on constraints. All previous decisions, which resulted in any kind of design or implementation, constitute a limiting factor constraining the ongoing development. However, current research is aware of the influencing nature of design constraints but does not sufficiently address them, especially not in the context of design decision reasoning during software evolution.

In our paper, we propose a decision process, which considers the constraints of an existing system as first-class entities. Furthermore, we present our tool called Decision

FoSADA'15, May 6, 2015, Montréal, QC, Canada.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3438-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2751491.2751495

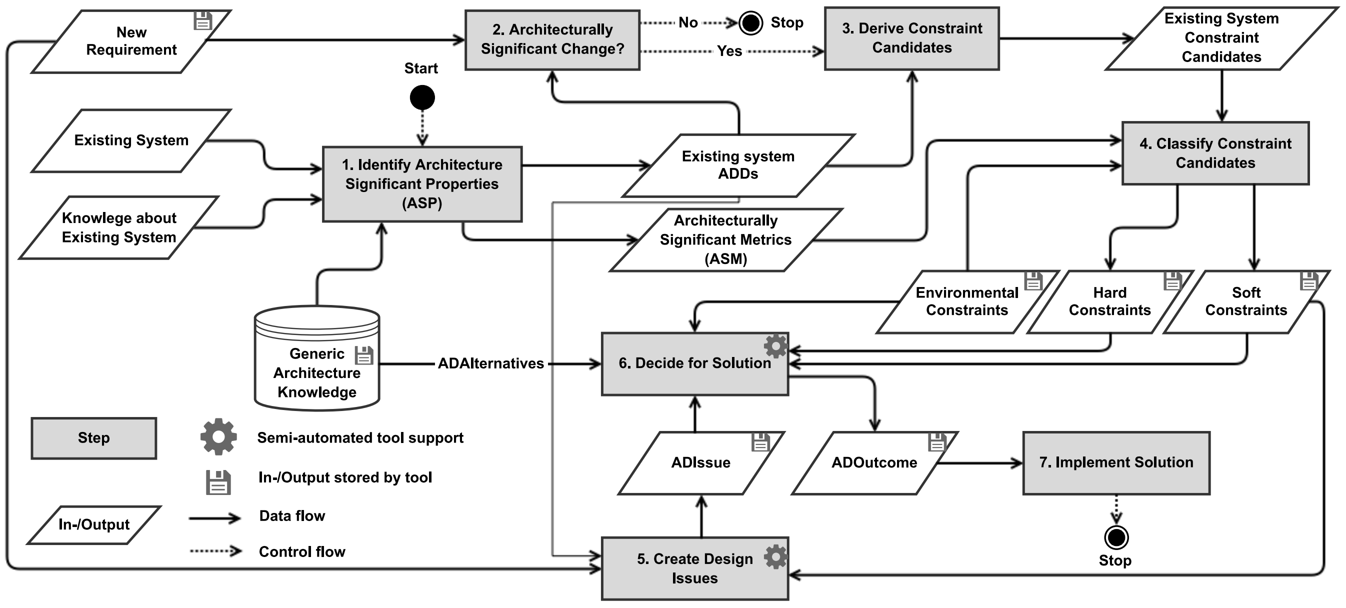

New Requirement
2. Architecturally Significant Change?
No
Stop
Yes
3. Derive Constraint Candidates
Existing System Constraint Candidades
Start
Existing System
1. Identify Architecture Significant Properties (ASP)
Existing system ADDs
4. Classify Constraint Candidates
Knowlege about Existing System
Architecturally Significant Metrics (ASM)
Environmental Constraints
Hard Constraints
Soft Constraints
Generic Architecture Knowledge
ADAlternatives
6. Decide for Solution
Step
Semi-automated tool support
In-/Output stored by tool
ADIssue
ADOutcome
7. Implement Solution
In-/Output
Data flow
Control flow
5. Create Design Issues
Stop

Figure 1: Constraint-based decision process in the context of evolving systems.

Buddy and describe how it assists the architect in taking design decisions based on our suggested concept.

2. CONSTRAINT-BASED DESIGN DECISION PROCESS

Our proposed process puts emphasis on the consideration of constraints in the context of evolving systems. Figure 1 displays the process, which comprises seven steps, which need to be executed sequentially in the order of their numbering. The process expects four inputs to start with: an Existing System, other sources of knowledge about the existing system (e.g. design documents), a New Requirement and Environmental Constraints. Each step, the mentioned inputs, and outputs are explained in detail in the following paragraphs.

Step 1 - Identification of Architecturally Significant Properties. One of the inputs is the Existing System, which comprises its source code, executable components, and configuration. Another input is diverse knowledge about the existing system, which contains documentation such as software architecture descriptions as well as tacit knowledge, which could be elicited by interviews with architects and developers. The main goal of this step is to identify the Architecturally Significant Properties (ASP), which comprises the Existing System ADDs and the Architecturally Significant Metrics (ASM). Both constitute the outputs of this step and are described below.

1. **Existing System ADDs**
   They are previous design decisions, which influenced the design of the existing system. The resulted design solutions are part of the ASPs. For example, a design decision could have led to a particular technology, framework, architectural style, design pattern, or COTS, which represent the ASPs. Furthermore, existing interfaces, components, connectors, and their configurations are the results of previous decisions and thus also part of the ASPs. In order to extract these Existing System ADDs, numerous approaches from system reverse engineering can be applied [3]. Even though the bare results of the decision without any justification are already a good start, recent approaches try to recover the rationale of design decisions (e.g. [21, 8]), which are also part of the ASPs. A further input, as shown in Figure 1, is a repository with Generic Architecture Knowledge supporting the exposure of various architectural solution properties.

2. **Architecturally Significant Metrics (ASM)**
   Another factor, which influences the ADDs, is the quality of the existing system. Metrics such as McCabe complexity, Halstead's metrics, and measurable object-oriented metrics [10] can be consolidated in different abstraction levels to support the architect in assessing the system and taking ADDs.

Step 2 - Determine the Significance of the Change. The main purpose of this step is to find out whether the New Requirement is architecturally significant or not. As this process is quite similar to the identification of Architecturally Significant Requirements (ASR), one could make use of existing approaches such as the framework proposed by Chen *et al.* [5]. The input of this step is the new requirement itself and the Existing System ADDs identified in Step 1, which influence the determination process. If the new requirement has no architectural impact, it can be implemented directly by the developers, which is out of the scope of the proposed process.

Step 3 - Derive Constraint Candidates. The input of this step are the Existing System ADDs identified in Step 1. As mentioned before, the result of each ADD is a solution, which influenced the existing system design. These solutions are source of constraints, which limit the ongoing system evolution, e.g., the application of a particular technology such as Spring MVC would discard future solutions based on Struts MVC design pattern. The goal of this step is to codify the Existing System ADDs as constraints, i.e., concrete properties need to be expressed by conditional constraints. A simple example would be the property "JRE version 7", which would be codified as "JRE version == 7" or "JRE version $\geq$ 7". OCL constraints, e.g., could be used to express object-oriented models such as interface specifications. Every single Existing System ADDs results in at least one Existing System Constraint Candidate, which constitutes the output of this step. The reason for treating the constraints as candidates is explained in the following step.

Step 4 - Classify constraint candidates. This is the core step of our process. Already known inputs are the ASMs from Step 1, and the constraint candidates from the previous step. All of them are considered as candidates due to the fact, that it still needs to be decided whether they are immutable or mutable constraints, which both constitute the output of this step. We call them hard and soft constraints, respectively. Hard constraints are immutable and mandatory. They need to be adhered to in any case. On the contrary, breaking soft constraints is at the architect's discretion. In case the architect decides to relax the constraints for the sake of meeting the new requirement or improving the design, it triggers a new design issue. For example, changing an existing interface is usually something one should avoid whenever possible. If it is a benefit to change the interface and it has a manageable amount of dependencies, it would be feasible to change it.

The main challenge of this step is how to determine the classification. Besides from the already mentioned ASMs in Step 1, the classification is driven by multiple factors depicted as Environmental Constraints in Figure 1. They consist of the four generic constraint categories proposed by Tang *et al.* [19].

- **Requirement Related Constraints:** They are derived from the functional requirements and include both the still valid requirements from the existing system and the recent ones from the new requirement.
- **Quality Requirement Related Constraints:** These constraints are derived from the non-functional requirements of both the existing system and the new requirement.
- **Contextual Constraints:** They correspond to the organizational and technological factors mentioned in the Global Analysis. Examples are factors like budget and platform but also social aspects such as skills.
- **Solution Related Constraints:** They arise during the design process, e.g., when deciding for a technology limiting further design choices.

Based on the environmental constraints and the ASMs, we are now able to decide whether a constraint candidate is considered to be hard or soft. For example a company-wide policy prohibits to use open source encryption would make an existing, easy to replace, and commercial encryption component to become a hard constraint instead of a soft one. These hard and soft constraints constitute a new category of constraints in the context of evolving systems. We call them ***Existing System Related Constraints***.

Step 5 - Create Design Issues from New Requirement and Soft Constraints. The main output of this step is a set of design issues (*ADIssue*). Firstly, design issues are triggered by the ASRs of the New Requirement. In order to perform this step, the New Requirement itself and the Existing System ADDs revealed in Step 1 are required. The Existing System ADDs enable the architect to identify sound design issues. For example, if the New Requirement demands multi-language support of the software, the Existing System ADDs would provide information about the implementation details. Hard-coded values strewn throughout the code, e.g., would result in a design issue demanding a technology to handle multiple languages like a localization framework. On the contrary, an existing localization framework would only entail the translation of a language file instead of introducing a new technology. Secondly, design issues can be triggered by the Soft Constraints, if the architect decides to relax them as mentioned in the previous step. Although creating design issues can somehow considered as initial decision-making as well, the final design decision considering different design alternatives, is taken in the following step.

Step 6 - Decide for a Solution. During this step the architect is taking the architectural design decisions, analogous to the decision making approaches widely described in the literature [18, 24, 6]. The activity requires two main inputs: the design issues *ADIssue* and a set of design alternatives *ADAlternative*. The design issues have been specified in the previous step and the design alternatives originate from the Generic Architecture Knowledge. Even though all alternatives are considered as solutions in the beginning, only very few are actually feasible. The design space with all its potential solutions is limited by multiple factors, namely our further inputs: the Environmental Constraints, the ASMs and finally the hard and soft constraints determined in Step 5. These factors are referred to as *ADDriver* [24, 4]. They all influence the architect in taking the decisions. For further assistance, feasible alternatives are ranked by suitability, e.g., by their contribution to quality attributes [2] or users' rating. The output of this step is a design choice represented by *ADOutcome*.

Step 7 - Implement Solution. To complete the process, the selected solution *ADOutcome* from the previous step is implemented in the last step. In addition, if the chosen solution yield new constraints, they are added to the Solution Related Constraints within the Environmental Constraints, as they may limit further design choices.

3. DECISION BUDDY

In this section, we describe how the tool Decision Buddy supports our constraint-based decision process. Initially the support is focused on Step 5 and 6 of the process. In these steps, the architect needs assistance in making design decisions by proposing suitable solutions. Furthermore, he needs assistance for the appropriate consideration of constraints originating from the existing system and the environment, as well as for the documentation of design issues, decisions, and their rationales. Steps 1-4 of the process provide input. Therefore, they have to be considered by the tool as they influence the decision making process. In order to provide the intended support, the tool has to offer a proper level of usability and collaborative features. Otherwise it would not contribute to improvements of the decision process in terms of efficiency, communication and reusability of solutions. In Figure 1 the semi-automated steps are marked by a cog wheel; the inputs and outputs, which are stored by the tool, are marked by a floppy disk symbol.

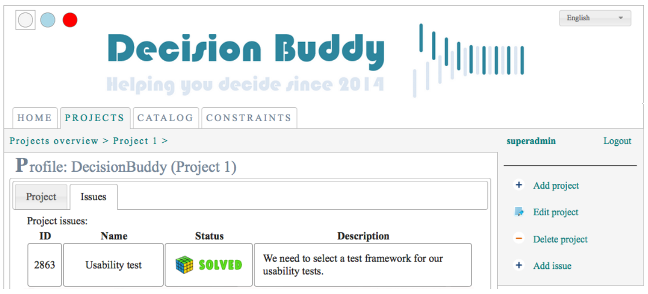
Firstly, the fundamentals of the tool are summarized. Secondly, the application of the tool concerning the decision making is described. Due to space limitations, we cannot describe the whole application in detail and concentrate on key features especially addressing the constraints and decision making.

3.1 Fundamentals of Decision Buddy

Decision Buddy is a web-based Java application. Key technologies used for the implementation are Spring MVC 4 using JavaServer Pages, and the JavaScript framework jQuery to offer a user-friendly web interface, Spring Data with Hibernate ORM and MySQL to persist operational data, and Spring Security to support authentication and authorization. Subsequently, some basic features and elements are described.

Solution Repository: The tool has a repository containing solutions, grouped by different categories like architectural styles, design pattern, frameworks, COTS, and refactoring solutions. It represents the Generic Architecture Knowledge in the aforementioned process. It can be browsed, added, modified, and deleted. Each solution is described by solution specific properties such as a version or license for frameworks or a type like structural or behavioral for design patterns. In addition, each solution can be commented and rated by the users, e.g., the solution's contribution to quality attributes or its applicability. Finally, multiple constraints can be defined and partially proposed automatically based on the solution's properties. The concept of constraints is described in the next but one paragraph.

Projects, Design Issues and Design Decisions: Further basic elements are projects, design issues and design decisions. A project groups multiple design issues as shown in Figure 2. Its advantage is the clear arrangement and structured storage of design issues, which enables the user to efficiently browse and survey the issues and the related information such as the status. A design issue represents a specific architectural design task including a description, a status and diverse meta data like author and date. It is linked to potential solutions representing design alternatives and to design decisions. Design decisions always include a justification of chosen or neglected solutions and affect the status of the issue, which can whether been solved or open. How the tool supports the decision making is described in Section 3.2. Analogously to the solutions, also design issues can specify constraints as described in the following paragraph.



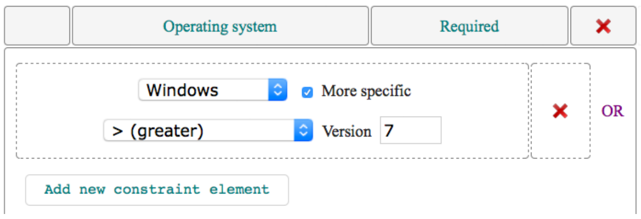
Profile: DecisionBuddy (Project 1)

Project issues:

ID	Name	Status	Description
2863	Usability test	SOLVED	We need to select a test framework for our usability tests.

Figure 2: Screenshot of the issues within a project.

Constraint Catalog: Constraints are stored within a dedicated constraint catalog. The concept to represent the constraints is based on previous work [14], which defines so-called technical terms. These technical terms consist of a meaningful identifier, a data type such as string or integer, and an informative but optional unit like kilobyte or version. String related data types further expect a set of string values. An example for a simple technical term is "Operating system" as the identifier, "stringInt" as data type and "Windows 7" as value. By the means of the technical terms, the constraints can now be specified by combining a technical term with an operator (equals, smaller than, etc.) and a value as shown in Figure 3. Each constraint may consists of multiple constraints elements, e.g., to match multiple operating systems. In addition, constraints can be combined by a logical condition. There is no differentiation between the different types of constraints in the catalog yet, as mentioned in Step 4 of our process.



Operating system | Required

Windows ☑ More specific

> (greater) Version 7

Figure 3: Screenshot of a specified constraint.

Users and Roles: The tool supports users and roles to enable the collaboration of different kind of stakeholders. Depending on the user's role, project specific information, e.g., are read only or even inaccessible.

Last but not least, to increase the ease of use, several usability features have been implemented. This includes and is not limited to the auto completion for quick search bars, inline editing and validity checks for inputs, themes and multi-language support for look and feel, and the filtering capabilities to speed up browsing of repositories and projects.

3.2 Tool-Supported Decision Making

Based on Step 5 and 6 of the decision process in Section 2 and the fundamentals explained in Section 3.1, we now describe how our tool Decision Buddy assists the architect in taking ADDs. Assuming that in previous steps of the proposed process the design issues have been created, potential solutions are available in the repository, and the constraints are known and specified, then the detailed view of each design issue displays a tab containing recommended solutions. As shown in Figure 4, the recommendation of solutions follows a process. In the beginning, the solutions are filtered in a way that only solutions, which adhere to the numerous constraints defined on both the solution's and design issue's side, are considered for further processing. After that, the solutions are ranked as specified by the user. To emphasize

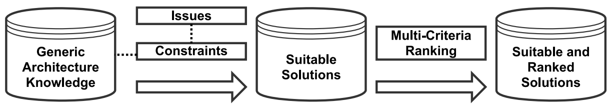
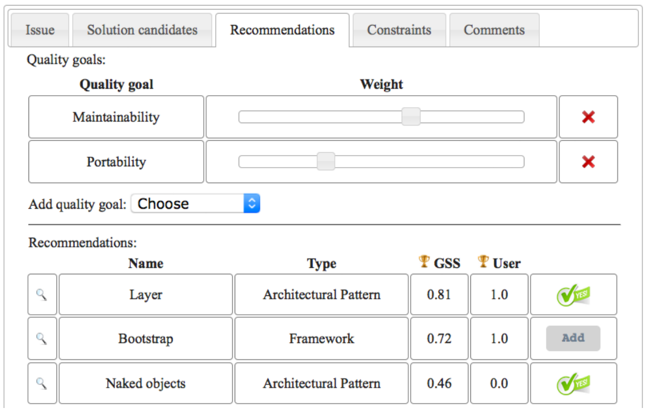


Figure 4: Process to filter and rank solutions.

the most convenient solutions, two different kind of rankings are available. The first one is the ranking by the solution's contribution to specific quality attributes determined by the Goal Solution Scheme (GSS), which has been introduced in earlier works [2]. The second ranking is based on the user's rating for specific solutions. A filtered and ranked list of solutions is depicted by Figure 5.



Quality goals:

Quality goal	Weight
Maintainability	
Portability	

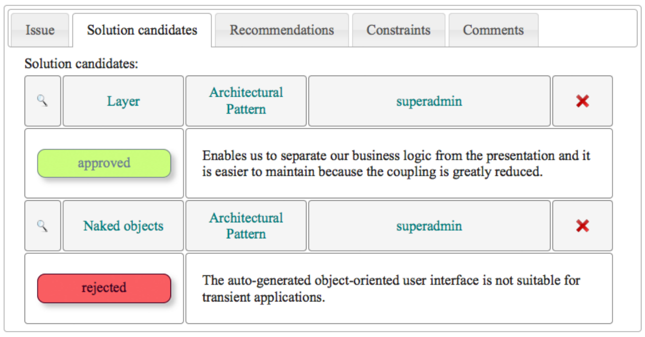
Add quality goal: Choose

Recommendations:

Name	Type	GSS	User
Layer	Architectural Pattern	0.81	1.0
Bootstrap	Framework	0.72	1.0
Naked objects	Architectural Pattern	0.46	0.0

Figure 5: Screenshot showing ranked solutions.

Usually, the architect now selects the top-ranked recommendations as the most suitable solutions candidates, which will be added to another tab. The selected candidates are considered as design alternatives (ADAlternative) to be align with the common ADD-making approaches [18, 24, 6]. Finally, for each alternative needs to be decided whether to approve it as solution (ADOutcome) or to reject it, as shown in Figure 6. In all cases the documentation of the rationale is mandatory. If at least one alternative has been accepted as a solution, the related design issue is considered as solved, which is shown in the project view (Figure 2) as well.



Solution candidates:

Layer	Architectural Pattern	superadmin
approved	Enables us to separate our business logic from the presentation and it is easier to maintain because the coupling is greatly reduced.	
Naked objects	Architectural Pattern	superadmin
rejected	The auto-generated object-oriented user interface is not suitable for transient applications.	

Figure 6: Screenshot showing the decision making.

4. RELATED WORK

Since the paradigm shift of describing the software architecture as a set of architectural design decisions [7], several tools have been developed to support capturing, reasoning, and sharing the ADDs. Each tool is based on a different architecture knowledge model and uses different terminologies. In addition, each tool provides different capabilities to assist the architect in dealing with the ADDs. Existing surveys [15, 16] have been conducted to evaluate and compare the different AK tools. However, each survey uses a different set of evaluation criteria. In order to understand the differences in the capabilities between the different tools, we analyzed and consolidated the different surveys. Moreover, we appended additional recent tools, which have not been considered by the surveys. We focused our analysis on two main aspects: A) The decision reasoning capabilities, B) The provided support for system evolution. Our analysis result showed to us the lack of support for design decision reasoning during system evolution. In other words, assisting the architect to choose the suitable architectural solution during the system evolution. In the following paragraphs, we analyze and report the differences between the different tools in regarding to both aspects.

PAKME [1] is a web-based tool, which provides services for architecture knowledge acquisition, retrieval, presentation and maintenance. In addition, the knowledge is classified between project specific and generic. The tool supports the reasoning process with a library of generic patterns, supported with an advanced searching capability. ADkwik is a web based tool, using the Web 2.0 and wiki technologies. The tool provides the ability to capture ADDs through selecting the suitable solution alternative from a list of stored architectural solutions. In addition, it relates the different ADDs using different types of relationships. The tool is based on the model of Zimmermann *et al.* [24], which guides the architect in exploring the design space. Software Architecture Warehouse [13] is a collaborative decision making web site, which provides different architectural solutions for design issues and gives the ability for different users to collaborate and discuss about solutions. In this way, each proposed solution is evaluated, which supports the decision maker to select the right solution. However, the evolution of design decisions and the influence of existing system design decisions on the future decisions are not being considered as part of the decision making model in the aforementioned tools.

On the other hand, AREL [18] is an ADDs capturing and traceability tool, which supports UML similar modeling for ADDs, as well as their drivers and constrains. Through this modeling, the architect can assess the impact of requirements or constrains changes on the existing system ADDs, which can support the architect to identify the possible impacted system changes during system evolution. Archium [9] is an eclipse based plugin tool, which provides the ability for the architect to describe the software architecture through a textual description. The Archium compiler visualizes the software architecture components and their associated ADDs. Moreover, it supports consistency check and traceability between the implemented Java code and the architecture design. An industrial implementation for an ADDs documentation framework [12] is proposed, which supports documenting the chronological order of ADDs. In addition, the tool supports the traceability between ADDs and the different artifacts within the development process. The mentioned tools in this paragraph support documenting the evolution of ADDs. Nevertheless, they lack the support for decision making guidance.

ADDSS [4] is an AK tool, which provides the ability to document ADDs for each project iteration, as well as the dependencies between them, through constrains relationships. In addition, the tool provides a repository for patterns, to support the architect selecting an architectural solution. ADvISE [11] is an eclipse plug-in, which support the architect to reason about the ADDs, through the QOC (Question, Option, Criteria) concept. An extension to the tool supports the uncertainty of taking the design decision through fuzzy logic. Additional support is provided for mapping ADDs to design diagrams and supporting the consistency between both. Both tools provide features for architecture reasoning and the documentation of ADDs evolution, but they do not support the architect in reasoning about their decision, during an existing system evolution.

5. CONLUSION AND FUTURE WORK

Architectural design decision making should not only consider requirements, organizational factors and technological limitations, but also constraints induced by existing systems. This paper presented a decision process focusing on the consideration of Existing System Related Constraints. It describes how this type of constraints is identified, classified and taken into account when making design decisions. Furthermore, the tool Decision Buddy is presented, which contributes to the decision making part of the proposed process.

As the decision process is still on-going work, there is enough space left for future research. First of all, we will elaborate more on the analysis of the existing system to improve the identification of the architecturally significant properties. Furthermore, the determination of constraints based on these properties need to be extended, especially regarding more complex properties and their formalization to enable automated processing. Another area of investigation is on how to maintain and fill the repository storing the Generic Architecture Knowledge, e.g., by extracting knowledge from websites and blogs. Finally, additional evaluation of our process is required through implementing and testing our proposed tool. We will implement further steps of the process and plan an industrial case study to reinforce its applicability.

6. ACKNOWLEDGMENTS

We thank all students who contributed to the design and implementation of Decision Buddy.

7. REFERENCES

[1] M. A. Babar, X. Wang, and I. Gorton. PAKME: A tool for capturing and using architecture design knowledge. In *INMIC'05*. IEEE, 2005.

[2] S. Bode and M. Riebisch. Impact evaluation for quality-oriented architectural decisions regarding evolvability. In *ECSA'10*, volume 6285 of *LNCS*, 2010.

[3] G. Canfora, M. Di Penta, and L. Cerulo. Achievements and challenges in software reverse engineering. *Commun. ACM*, 54(4):142–151, Apr. 2011.

[4] R. Capilla, F. Nava, S. Pérez, and J. Dueñas. A web-based tool for managing architectural design decisions. *SIGSOFT Softw. Eng. Notes*, 31(5), 2006.

[5] L. Chen, M. A. Babar, and B. Nuseibeh. Characterizing architecturally significant requirements. *IEEE Software*, 30(2):38 – 45, 2013.

[6] S. Gerdes, S. Lehnert, and M. Riebisch. Combining architectural design decisions and legacy system evolution. In *ECSA'14*, pages 50–57, 2014.

[7] A. Jansen and J. Bosch. Software Architecture as a Set of Architectural Design Decisions. In *WICSA'05*, pages 109–120, 2005.

[8] A. Jansen, J. Bosch, and P. Avgeriou. Documenting after the fact: Recovering architectural design decisions. *J. Syst. Software*, 81(4):536–557, 2008.

[9] A. Jansen, J. Der Ven, P. Avgeriou, and D. Hammer. Tool Support for Architectural Decisions. In *WICSA'07*, pages 4–4. IEEE, Jan. 2007.

[10] M. Lanza, R. Marinescu, and S. Ducasse. *Object-Oriented Metrics in Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[11] I. Lytra, H. Tran, and U. Zdun. Constraint-based consistency checking between design decisions and component models for supporting software architecture evolution. In *CSMR'12*. IEEE, 2012.

[12] C. Manteuffel, D. Tofan, H. Koziolek, T. Goldschmidt, and P. Avgeriou. Industrial Implementation of a Documentation Framework for Architectural Decisions. In *WICSA'14*, pages 225–234. IEEE, 2014.

[13] M. Nowak and C. Pautasso. Team Situational Awareness and Architectural Decision Making with the Software Architecture Warehouse. In *ECSA'13*. IEEE, 2013.

[14] A. Pacholik and M. Riebisch. Modelling technical constraints and preconditions for alternative design decisions. In *MBEES'12*, 2012.

[15] M. Shahin, P. Liang, and M. R. Khayyambashi. Architectural design decision: Existing models and tools. In *WICSA/ECSA'09*. IEEE, 2009.

[16] A. Tang, P. Avgeriou, A. Jansen, R. Capilla, and M. Ali Babar. A comparative study of architecture knowledge management tools. *J. Syst. Software*, 83(3):352–370, 2010.

[17] A. Tang, M. A. Babar, I. Gorton, and J. Han. A survey of architecture design rationale. *Journal of Systems and Software*, 79(12):1792–1804, Dec. 2006.

[18] A. Tang, Y. Jin, and J. Han. A rationale-based architecture model for design traceability and reasoning. *Journal of Systems and Software*, 80(6):918–934, June 2007.

[19] A. Tang and H. Van Vliet. Modeling constraints improves software architecture design reasoning. In *WICSA/ECSA'09*, pages 253–256, 2009.

[20] D. Tofan, M. Galster, and P. Avgeriou. Capturing tacit architectural knowledge using the repertory grid technique. *ICSE'11*, pages 916–919, 2011.

[21] J. van der Ven and J. Bosch. Making the right decision: Supporting architects with design decision data. In K. Drira, editor, *Software Architecture*, volume 7957 of *LNCS*, pages 176–183. Springer, 2013.

[22] U. van Heesch, P. Avgeriou, and R. Hilliard. Forces on Architecture Decisions - A Viewpoint. In *WICSA/ECSA'12*, pages 101–110. IEEE, 2012.

[23] H. Vliet. *Software Engineering: Principles and Practice*. Wiley, 2nd edition, 2007.

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