Meta-model Based Framework for Architectural Knowledge Management

Manoj Bhat
Technische Universität München
Boltzmannstr. 3
85748 Garching, Germany
manoj.mahabaleshwar@tum.de

Klym Shumaiev
Technische Universität München
Boltzmannstr. 3
85748 Garching, Germany
klym.shumaiev@tum.de

Andreas Biesdorf
Siemens AG - Corporate Technology
Otto-Hahn-Ring 6
81739 München, Germany
andreas.biesdorf@siemens.com

Uwe Hohenstein
Siemens AG - Corporate Technology
Otto-Hahn-Ring 6
81739 München, Germany
uwe.hohenstein@siemens.com

Michael Hassel
Siemens AG - Corporate Technology
Otto-Hahn-Ring 6
81739 München, Germany
michael.hassel@siemens.com

Florian Matthes
Technische Universität München
Boltzmannstr. 3
85748 Garching, Germany
matthes@tum.de

ABSTRACT

The need to support a software architect's day-to-day activities through efficient tool support has been highlighted both in research and industry. However, managing enterprises' Architectural Knowledge (AK) to support scenarios such as decision making, what-if analysis, and collaboration during the execution of large industrial projects is still a challenge. In this paper, we propose the architecture of an AK management framework to support software architects to manage AK. The components of this framework including SyncPipes and rule engine allow software architects to consolidate projects' data from disparate sources and to define domain-specific rules to address the challenges in inconsistency analysis, context-sensitive recommendations, and tracking of artifacts within projects. The technical details for realizing the components of the framework are also presented. The proposed AK management framework has been successfully implemented as part of an industrial project and is currently being evaluated in different domains.

CCS Concepts

•**Software and its engineering** → **Designing software;** *Software implementation planning;*

Keywords

Architectural knowledge management; meta-models; domain-specific language; tool support

1. INTRODUCTION

ECSAW '16, November 28-December 02, 2016, Copenhagen, Denmark
 ISBN 978-1-4503-4781-5/16/11…$15.00
DOI: http://dx.doi.org/10.1145/2993412.3004848

During the past few years, several approaches, frameworks, and meta-models to capture and manage Architectural Knowledge (AK) have been proposed and applied in research as well as in industries [5]. This trend is mainly due to the fact that AK is becoming more and more an organizational asset [12]. Harnessing this asset enables knowledge reuse, supports decision making, and avoids knowledge evaporation within an organization. The aim of AK Management (AKM) is to codify the tacit knowledge residing in the minds of software architects explicitly in either structured or semi-structured knowledge bases. Subsequently, such knowledge bases can be used to support software architects to take architectural decisions in applying the application-generic knowledge including architectural styles, design patterns, and architectural standards to specific project instances [14].

Architectural decisions are made by considering various influencing factors including project context, architecture-significant requirements, past decisions, availability of time, budget, and human resources [21]. Therefore, we agree with R. Capilla et al. [11] and emphasize the need for linking AK elements to different artifacts in the software engineering lifecycle. These artifacts include project plans, minutes of meeting, requirement specification documents, architecture design documents, source-code commits, bug reports, and change requests. Typically in large industrial projects, the aforementioned artifacts are maintained by multiple stakeholders in a variety of disparate information systems. Thus, providing a consolidated view to software architects to make informed decisions based on the current state of the project becomes challenging. This challenge has also been discussed in [10], wherein the authors highlight the need for AKM tools to interoperate with tools from different phases of the software engineering lifecycle and to establish traces between the artifacts generated during the process.

To address the aforementioned challenges, we propose an AKM framework that supports software architects during the execution of projects in a collaborative environment. In particular, we first focus on the generic use cases identified in [10] and [17] for AKM tools. We further consider organization-specific use cases that were captured to support scenarios within an industrial project. The proposed

AKM framework captures the essential components such as a meta-model based platform, a knowledge base, a rule engine, and a data synchronization (SyncPipes) component. In particular, the SyncPipes component addresses the challenge of selective model selection, alignment, and transformation of data spread across different tools into the AKM platform. The rule engine component allows experts to capture domain-specific rules that are used for generating context-sensitive recommendations. These components rely on a meta-model based approach that provides the flexibility to adapt the domain models to meet different organization and project context. The architecture of the framework and the technical details to realize the components within the framework form the main contributions of this paper.

The remainder of this paper is organized as follows. Section 2, presents the architecture of the AKM framework. In Section 3, we demonstrate the results of the application of our framework by considering usage scenarios in an industrial setting. Section 4, presents an overview of the related work. Finally, the paper concludes with a short summary and an outlook on further research.

2. AKM FRAMEWORK

The AK can be classified into four broad categories, namely *context*, *design*, *general*, and *reasoning* knowledge [26]. The *context knowledge* captures the project-specific information such as management information and architectural significant requirements. The *design knowledge* comprises of the architectural designs of the software systems. In our framework, we do not distinguish between context and design knowledge as we consider the design knowledge to be part of the context knowledge which is dynamic and evolves as the project progresses. We refer to the context and design knowledge as *dynamic* knowledge and the general knowledge as *static* knowledge (as changes are less frequent). The *general* knowledge captures architectural methods, styles, patterns, and organization-specific corporate information (e.g., processes and standards) that helps architects while designing software systems. Finally, *reasoning* knowledge contains information that guides software architects to apply static knowledge in the project context. It also maintains design decisions, rationales, and alternatives that were considered during decision making in specific project instances, which can be reused in similar projects.

The high-level layered architecture of the AKM framework is shown in Figure 1. The components within the framework capture the aforementioned AK categories and present the information to software architects through different client applications including a synchronization component (SyncPipes), model designer, dashboard, rules manager, and what-if simulator. The dynamic knowledge is kept in-sync with the current state of the project through the *SyncPipes* component. The meta-model and the ability to create a domain model (static and dynamic knowledge model) at runtime is handled by the *platform* component which forms the core of the framework. A software architect can configure the domain model through the model-designer client application. The reasoning knowledge is managed by the *rule engine* component which captures the reasoning logic using model-based expressions that accesses the meta-model for computations. The software architect manages the rules using the rules-manager client application. The domain models, model-based expressions, and model instances can be managed through REST APIs. Such an architecture ensures "separation of concerns" between client applications and the platform. The components of the framework are elaborated in detail in the subsequent subsections. Each subsection corresponds to an individual component and captures the technical details on how to realize each component.

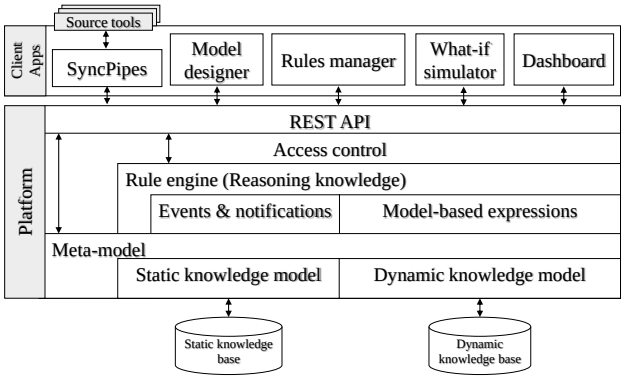

Figure 1: Architecture of the AKM framework

2.1 Platform component

The two main use cases as discussed in [10], which needs to be addressed by an AKM platform are: (a) one-size-fits-all approach does not work: the domain model (e.g., AKM model in this context) should be organization-, domain-, and project-specific and (b) the domain models should be reusable and configurable to the project's context (team size, development methodology, processes, etc.). To address these use cases, we propose the use of a meta-model based content management system that supports the co-evolution of the domain model and its data (cf. [23]). Such a meta-model based system allows experts to adapt the domain model and domain-specific rules at runtime to accommodate different project needs. In particular, we use the hybrid-wiki meta-model [20] for the creation of domain models within our platform. Alternatively, practitioners can also refer to meta-model based "backend as a service" platforms (e.g., Parse[1]) that provide similar flexibility. The main concepts of the meta-model are *Type* and *Property* (analogous to EClass and EAttribute in Ecore meta-model) which allow users to create domain models. The Type concept is used to create a domain-specific concept (e.g., architectural decision) and Property captures its features (e.g., description or link to decision alternative). Furthermore, each Type can have multiple *Entities* (representing data) and each Property can be mapped to multiple *attribute values*. For example, a user can create an instance of an Entity "use model-view-controller pattern" and can link it to "architectural decision" Type.

We have created the dynamic and static knowledge models by instantiating the above meta-model. The dynamic knowledge model comprises of concepts from the domain of project management, requirements management, architecture management, implementation, and maintenance. As motivated in Section 1, providing a consolidated view over the current state of the different phases of the software engineering lifecycle is crucial for decision making. For constructing our dynamic knowledge model and to support the use cases discussed in Section 3, we analyzed the existing ontologies [3] [13] [27], Open Services for Lifecycle Collaboration standard [28], and data models of specific tools such as

[1]https://parse.com/

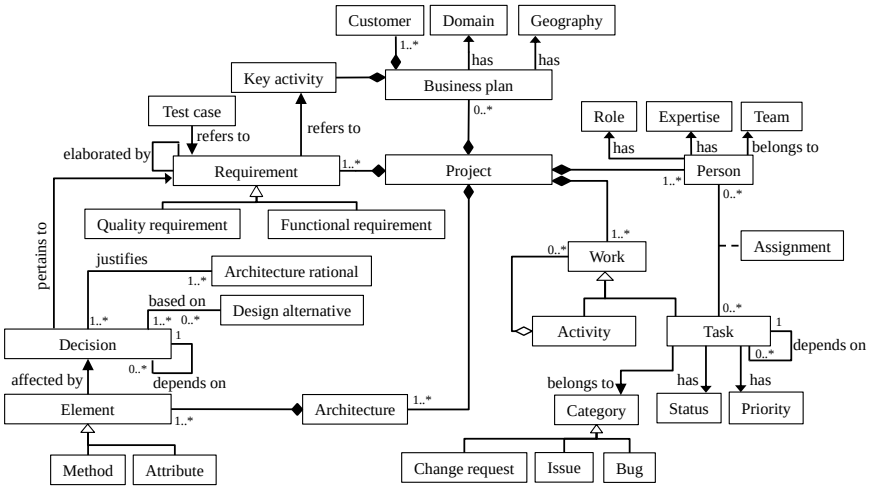

Figure 2: Dynamic knowledge model to capture the AK

Business Canvas Model, MS Project, Enterprise Architect, JIRA, and Bugzilla which are extensively used in practice. It should be noted that since the dynamic knowledge model is an instance of the meta-model, it can be configured through the model designer and can be adapted to the project needs at runtime. The technical details for implementing such a meta-model based system are described in [9].

The core concept within the dynamic knowledge model is *Project* with multiple attributes such as name, description, start date, and end date. As shown in Figure 2, a *Project* is associated with its corresponding *Business plan* which is further associated with concepts derived from the Business Model Ontology [22]. The concept *Key activity* in the business model ontology is referenced by the concept *Requirement*. A project has multiple requirements which are classified into *Functional* and *Quality requirements*. As also modeled in [13] and [27], an architectural *Decision* depends on the quality requirements and results in a specific *Architecture*. A decision is made by considering multiple *Design alternatives*. An *Architecture rationale* justifies the decision made by an architect (cf. [1]). The architecture is further elaborated with concepts such as architecture *Element* composed of *Attributes* and *Methods*. All these concepts are derived from the Enterprise Architect [4] modeling tool's schema. Furthermore, since an architect needs to have a consolidated view over the current project plan and the availability of resources, we have included the concepts from the project management domain in our domain model. The concepts and the relationship between the concepts such as *Person*, *Task*, and *Assignment* are derived from the work in [8] and the XML schema of the Microsoft Project management tool. Also, since tools such as JIRA and Bugzilla are commonly used during the development and maintenance phase for managing tasks, we also investigated these tools and incorporated concepts such as task *Categories* (*Issue*, *Bug*, and *Change request*) and their corresponding workflows. To provide better readability, Figure 2 captures only a subset of concepts and their relationships.

2.2 Knowledge base component

The knowledge base component manages the static knowledge base model and its data. The static knowledge base captures application-generic knowledge including styles, reference architectures, and corporate knowledge (e.g., templates for architectural methods and experts who can help instantiate architectural methods). As shown in Figure 3, the static knowledge model focuses on the *Methods* applicable in the architecture lifecycle of a project. Examples of architectural methods include design methods (attribute driven design and 4+1 views) or analysis methods (scenario-based analysis, architecture analysis method, and architecture trade-off). The architectural method belongs to a *Phase* in the architecture life cycle and can be composed of multiple *Method steps*. A *Method* confirms to a specific architectural *Standard* and requires human, budget, and time *Resources*. Within an organization, software architects with expertise (*Experts*) in specific methods can help other project partners in applying these methods. Furthermore, specific *Methods* could also use the *Architectural elements* such as architecture styles and patterns in their method steps. Each *Method* generates corresponding *Artifacts* which can be instantiated using organization-specific *Templates*.

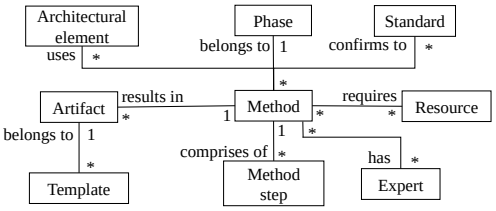

Figure 3: Static knowledge model

The knowledge base component covers the following use cases presented in [17]: (a) persist, modify, and delete AK elements, (b) search AK elements using keywords and categories, (c) subscribe to AK elements and get notifications when they are updated, and (d) manage versions of the AK.

The need for these use cases are also highlighted in [17]. The focus of the knowledge base component is to manage (gather, structure, store, search, and version) the static architectural knowledge and to make them available for the rule engine that guides software architects based on the current state of the project and its context.

2.3 Rule engine component

The rule engine component forms the core of the recommender system. Using the current state of the project (dynamic AK) and the application-generic knowledge (static

AK), the rule engine evaluates the rules in the rules repository and presents relevant recommendations to software architects. These recommendations include suggestion of applicable architectural methods, domain experts who can be consulted for the method execution, and highlighting missing project artifacts and traceability links. The rules within the rule engine are event condition action (ECA) rules implemented using the model-based expression language (MxL) [24]. The MxL is a domain-specific language that allows querying data in the knowledge base. A rule represented using MxL is an expression and executing a rule implies evaluating the corresponding expression. Since MxL is defined over the meta-model, it can access all the Types - which represents concepts in the domain model. A simple expression such as "*find(Project).where(Status='ongoing')*", returns all the instances of Projects that are currently ongoing. For implementing such rules, practitioners can also use alternative business rules management systems such as Drools [7]. However unlike MxL, these rule engines are typically model-based and not meta-model based systems. This allows MxL to automatically update the rules as the domain model evolves. For instance, if the concept non-functional requirement is changed to quality requirement in the domain model then the respective rules are updated accordingly.

An event in an ECA rule triggers the execution of the corresponding rules. We categorize these events into three broad classes as described below.

- *Domain events* are triggered due to data-level changes within the artifacts of a project. Typically, these are the frequently occurring events since they reflect the changes in the state of a project. The domain events could represent, for instance, uploading an architecture document of a project, changing a task's status in the project, or adding a decision to use a specific reference architecture. Furthermore, since data is captured as instances of Entity and Attribute types in the meta-model (cf. Section 2.1), operations (create, update, or delete) on these types would trigger a domain event.

- *Model-change events* are triggered when the domain model is updated to capture the changing project context. For instance, when a domain expert deletes the relationship between Decision and Architecture and adds a relationship between Decision and Architecture Element, the corresponding rules need to be updated and re-evaluated. Since concepts belonging to the domain model are represented as instances of the Type and Property in the meta-model any operation (update and delete) on them will trigger this event.

- *User-triggered events* are used when an architect actively executes (triggers manually) the rules to identify the actions that need to be performed. These events are generated through the user interface, for instance when an architect clicks an architectural method link.

If the condition in the ECA rule is satisfied, then the corresponding action is performed. The conditions are expressed using MxL expressions and can be either simple or nested expressions. For instance, a simple expression could include checking the status of a project and a nested expression could represent multiple such queries joined by 'and' and 'or' operators. The actions are the recommendations provided to software architects only if all the conditions are satisfied. These actions are also represented as MxL expressions as they also allow invocation of operations on the data.

To illustrate the application of a simple rule, let us consider the following use case: the system should recommend domain experts with a minimum of five years of experience in architecture review and analysis. Furthermore, such a recommendation should be context-aware, i.e., it is relevant for software architects only when the following conditions are satisfied c1) the status of the project is "ongoing", c2) the project is in the "design" phase, and c3) the task "prepare architecture documentation" is "complete". The corresponding ECA rule can be captured as a MxL expression along with the meta information such as name, description, and parameters as shown in Listing 1. Parameters are a list of input variables that can be used within an expression.

Listing 1: An exemplary MxL expression

```
Name: getDomainExperts
Description: Recommend domain experts
Parameters: projectName:String
Return Type: Sequence<String>
Method stub:
let projectVar = find(Project).single(name=
    projectName) in
let task = find(Task).single(name="prepare
    architecture documentation" and project
    = projectVar) in
let conditions = if projectVar.status = "
    ongoing" and projectVar.Phase = "Design
    " and task.status = "closed" then true
    else false in
let actions = find(Expert).where(Expertise
    = "Architecture review and analysis"
    and Experience > 5).select(name) in
if conditions then actions else []
```

The above rule is evaluated for every domain event related to Project or Task. This rule can also be triggered by a user-triggered event and by executing the *getDomainExperts('Amelie')* expression. This example shows how the system uses the dynamic knowledge about the current state of the project along with the static knowledge (with expertise information) to recommend context-sensitive information to software architects. It also indicates that the expressions rely on the concepts from the domain model introduced in Figure 2. The rules in the repository not only include such recommendations, but also include actions such as generating reports or downloading templates based on the decisions taken by the architects during the project. The rule engine component specifically addresses the use case to offer automated support in decision making for architects.

2.4 SyncPipes – synchronization component

The SyncPipes component addresses the general use case of AK integration identified for AKM tools (cf. [17] and [10]). Specifically, this component empowers architects in enabling the synchronization of the data that is spread across software engineering lifecycle tools within one centralized AKM tool. SyncPipes acts like a "bot" with sensors and actuators for our platform. The sensors monitor the current state of the projects' artifacts and actuators keep the information within the knowledge base synchronized.

Figure 4 illustrates the conceptual model of the SyncPipes component. The *Source* concept represents various source tools that can be handled by SyncPipes (e.g., JIRA, MS Project, Excel, Enterprise Architect) whereas the *Target* concept corresponds to our platform. Both the source and target concepts are specializations of the *ToolModel*. The

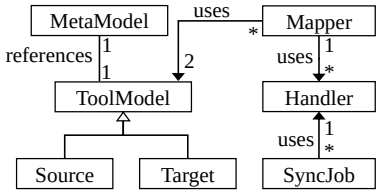

Figure 4: Conceptual class diagram of SyncPipes

ToolModel provides interfaces to connect with the tools using user credentials. It also implements methods such as getTypes and getEntities to work with a unified data-model representation. The data-model of the source and target tools confirm to a *meta-model* represented using the JSON schema [16] format. The SyncPipes component uses a *Mapper* to capture the mapping of the concepts from the source to the target tool's data-model. For instance, a software architect can map a concept "New Feature" in JIRA to "Requirement" in our domain model. The mapper allows architects to consider only those concepts and attributes that are relevant and necessary to establish traceability across the artifacts. Once the mapping is specified by a software architect, the *Handler* concept uses the mapping information to extract data from the source tool, transform the data, and persist it in the target tool. Subsequently, the *Handler* also starts the *SyncJob* that triggers the transformation process based on specific events or at regular time intervals. The above mentioned concepts support selective model selection, alignment, transformation, and synchronization of data residing in different tools into an AKM platform.

3. USAGE SCENARIOS AND EVALUATION

The proposed architecture of the AKM framework has been validated in an industrial project named "Architecture Management Enabler for Leading Industrial softwarE (AMELIE)". Currently, the prototypical implementation of the AMELIE system is being tested in different business domains to evaluate the different components presented in our AKM framework. To illustrate the major usage scenarios, we consider one of the projects within which the project stakeholders manage their artifacts in different tools and formats that are most suited for their purposes. The software architecture team, however, is responsible for consolidating all the information, taking architectural decisions, and driving managerial decisions. The considered project deals with building a component for a data analytics system.

A software architect logs into the AMELIE system, and creates a new project or selects an existing one. A web page similar to the one in Figure 5 is shown to the architect. On the left sidebar of the page, the architect can access different functionalities of the system including gap analysis, SyncPipes component, architecture metrics, data editor and personal settings. In the center of the web page, the gap analysis feature is presented in a dashboard. It consists of an upper triangular matrix where each row and column represents a phase in the software engineering lifecycle. The diagonal cells in the matrix correspond to a specific phase and indicate semantic gaps within the artifacts of the corresponding phases. Initially, for a new project the matrix is empty and the user gets the recommendation to associate artifacts for each of the phase. For instance, at the start of the project, the architect is guided to upload requirement specification documents to the project using the SyncPipes component. Grey color indicates that the artifacts are not yet available in AMELIE. The yellow cell indicates missing concepts within the synchronized document, for instance, if the requirements document does not address quality attributes. On resolving inconsistencies within an artifact, the corresponding cell changes to green color.

For the remaining non-diagonal cells, yellow color indicates discrepancies between phases at the crossing point. For e.g., a yellow cell within the Requirements row and Functional Architecture column indicates inconsistencies between concepts in these phases (e.g., an architecture element is not linked to any requirement). As shown in Figure 5, this information is presented in the lower section of the screen. These discrepancies are identified based on the rules in the rule engine and the concepts in the AK model.

In our reference project the high-level requirements are managed by project managers in an Excel file as a list of features with feature IDs, descriptions, responsible person, feature start date, and feature end date. The concrete tasks performed by team members are maintained in the JIRA system. The functional and technical architecture are designed in the Enterprise Architect software and finally, the source code is maintained in a Git version control system.

A software architect while synchronizing these artifacts, maps the concepts from the source system to the domain model of the Amelie system. The mapping hence relates artifacts. For instance, while importing tasks from JIRA, an architect can map a field within JIRA such as "feature id" to "requirements id" in the domain model using the SyncPipes component. The SyncPipes component resolves the references and establishes links between requirements and tasks. Moreover, synchronization of artifact also triggers a domain event that executes a rule to suggest context-dependent information. For e.g., a rule such as "*if project.where(Status='ongoing' and requirementsDocument=true) then ['Architecture review'] else null*" is executed to recommend architects to perform requirements review. Based on such suggestions, architects can also start a new activity within the system. In this case, the decision to perform the activity is automatically captured as an architectural decision and persisted in the knowledge base. These rules can also be customized to the project needs using the rules manager component.

Corresponding to each activity and based on the current state of the project, a software architect gets recommendations (on the right-hand side of the dashboard) about experts within the company (cf. rule in Listing 1), training material, software tools, and templates. These rules can also be triggered by clicking on each of the cells in the dashboard which invokes a user-triggered event to evaluate the rules. The training material and templates for instance are retrieved from the static knowledge base component presented in Section 2.2. Finally, a software architect can also add information as instances of concepts (or entities) in the domain model (Decision, Design alternative, Architecture, etc.) using the data editor. Each entity is further represented using a wiki page which supports multiple off-the-shelf features including versioning, sharing and discussion.

The viability of the proposed framework is validated through the presented usage scenario. However, an extensive verification of its utility needs to be performed in a broader scope.

4. RELATED WORK

There is a large body of knowledge that captures mod-

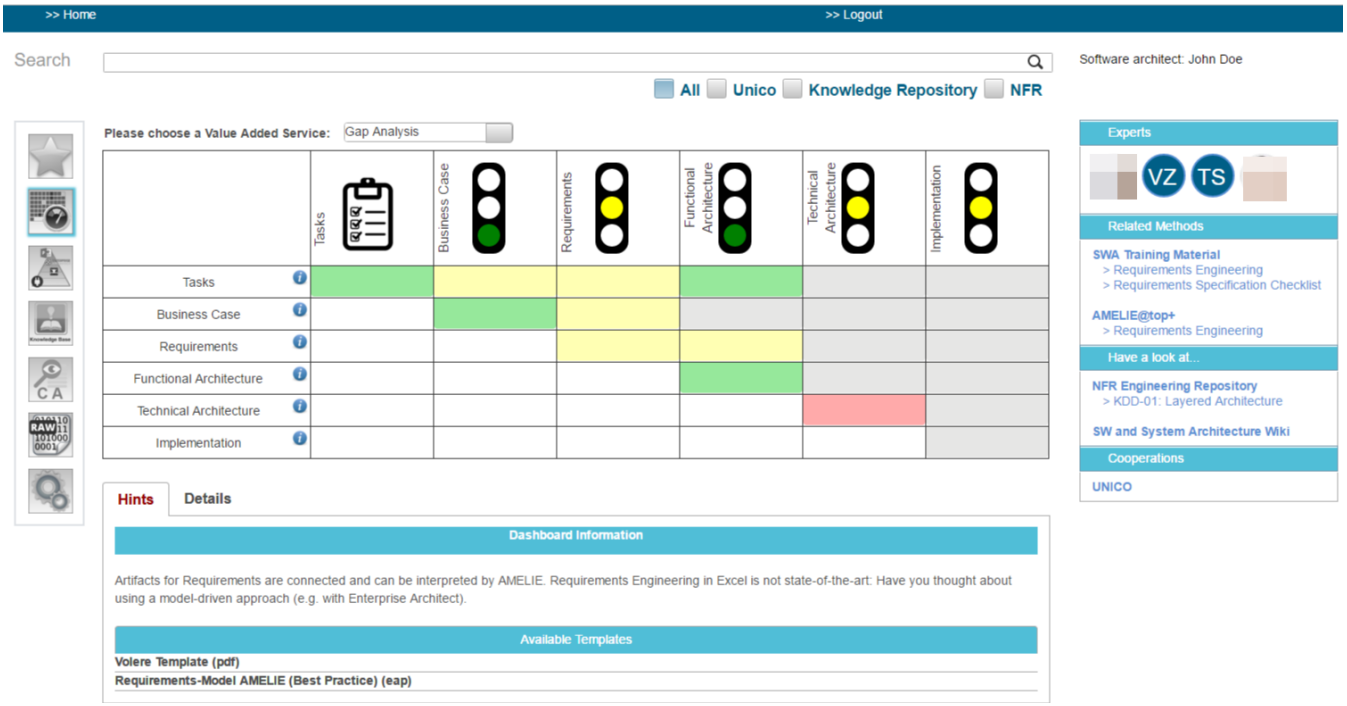

Figure 5: Architecture Management Enabler for Leading Industrial softwarE - Dashboard

els, approaches, and frameworks for managing AK. Models for AKM focus on the core concept of design decisions and its associated concepts. We have considered some of these models [3], [11], [27], [13] in the previous sections to derive our own AK domain model that supports the use cases of the AMELIE system. In this section, we focus on some of the AKM tools which have commonalities with our solution.

Farenhorst et al. [15] propose an AK sharing system named "Environment for Architects to Gain and Leverage Expertise". This system emphasizes the use of a knowledge base for maintaining best practices, architecture documents, and yellow pages that provides user-specific content. The system also includes blogs, wikis, and discussion boards for enabling collaboration between architects. The user interface of the system is personalizable and helps users to easily access and manipulate the content. However, this AK sharing tool does not provide any context-specific information based on the current state of the projects.

Many of the AK platforms including PAKME [6], AdKwik [25], and ArchiMind [13] achieve collaboration and knowledge sharing through wiki-based systems. PAKME [6] considers concepts from project management and contact management in its domain model, similar to the dynamic knowledge model in our platform. The focus of this tool is only to persist, search, retrieve, and present AK to end users and does not cover aspects of knowledge synchronization and rule-based context-sensitive recommendations. On the other hand, the ADkwik platform has a broader scope and the components in this platform including dependency management, decision workflow and import/export have commonalities to our proposed solution. However, the authors of [25] do not provide details on how to realize such an architecture. Furthermore, since these platforms are not meta-model based systems, the ability to adapt the domain model and the rules at runtime is limited.

AKM tools such as Decision Architect (DA) [19] and ADvISE [18] enable software architects to explicitly document, trace, and analyze architectural decisions. DA is a plug-in for the Enterprise Architect system and concepts within the DA can be mapped to the modeling elements in the Enterprise Architect. Since Enterprise Architect provides plug-ins to capture artifacts from different phases of the software engineering lifecycle, DA can trace the impact of architecture decisions across artifacts. The major constraint for such systems is that stakeholders need to use a standard set of tools during projects' lifecycle. Weinreich et al. [2] further analyze these tools elaborately in their literature study.

Capilla et al. [10] also present a detailed comparison of AKM tools. Through their research they have identified interesting challenges that restrict the adoption of AKM approaches in industry. Some of these challenges are addressed through our AKM framework, including interoperability with tools from different phases of software engineering lifecycle to establish traces between artifacts and the support for runtime decision making in ongoing projects.

5. CONCLUSIONS AND FUTURE WORK

AKM systems play a vital role in supporting software architects during the execution of large industrial software projects. These systems need not only provide reference material using knowledge repositories, but also observe the current state of the project to provide context-sensitive information. This can only be achieved if the system can consolidate organization- and project-specific data from different sources and then reason about the data.

In this paper, we have presented the architecture of a meta-model based AKM framework to support software architects for managing AK. The meta-model based approach allows architects to configure the AK model to their project-specific needs at runtime. Using this meta-model, other components such as the rule engine and synchronization engine provide different services to the client applications. The

rule engine allows experts to write domain-specific rules to provide context-sensitive recommendations during the execution of projects. The SyncPipes component allows an architect to consolidate data from all the relevant sources.

The presented AKM framework is realized in the AMELIE system that is currently being used and evaluated in different industrial settings. We have demonstrated the application of our AKM framework using an industrial project. Our claim that meta-model based platforms provide the ability to adapt domain models at runtime has only been proved in domain such as enterprise architecture management, collaborative product development and content management systems (cf. [23]), but not explicitly in the software architecture domain. Therefore, as part of our future work, we plan to extensively evaluate the components of the framework in different project contexts and share the lessons learned from applying a meta-model based AKM framework in practice.

6. ADDITIONAL AUTHORS

Additional authors: Florian Mittruecker (Siemens AG - Corporate Technology, Otto-Hahn-Ring 6, 81739 München, Germany, email: `florian.mittruecker@siemens.com`).

7. REFERENCES

[1] Iso/iec/ieee systems and software engineering – architecture description. *ISO/IEC/IEEE 42010:2011(E)*, pages 1–46, Dec 2011.

[2] A fresh look at codification approaches for SAKM: A systematic literature review. *Lecture Notes in Comput. Sci.*, 8627 LNCS:1–16, 2014.

[3] D. Ameller and X. Franch. Ontology-based architectural knowledge representation: structural elements module. In *Advanced Inform. Syst. Eng. Workshops*, pages 296–301. Springer, 2011.

[4] E. Architect. Sparx systems, 2010.

[5] M. A. Babar, T. Dingsøyr, P. Lago, and H. van Vliet. *Software architecture knowledge management*. Springer, 2009.

[6] 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*, page 11. IEEE Computer Society, 2007.

[7] P. Browne. *JBoss Drools business rules*. Packt Publishing Ltd, 2009.

[8] B. Bruegge and A. H. Dutoit. *Object-Oriented Software Engineering Using UML, Patterns and Java-(Required)*. Prentice Hall, 2004.

[9] T. Büchner, F. Matthes, and C. Neubert. Data model driven implementation of web cooperation systems with tricia. In *Objects and Databases*, pages 70–84. Springer, 2010.

[10] R. Capilla, A. Jansen, A. Tang, P. Avgeriou, and M. A. Babar. 10 years of Software Architecture Knowledge Manag.: Practice and Future. *J. of Syst. and Soft.*, 2015.

[11] R. Capilla, O. Zimmermann, U. Zdun, P. Avgeriou, and J. M. Küster. An enhanced architectural knowledge metamodel linking architectural design decisions to other artifacts in the software engineering lifecycle. In *Software Architecture*, pages 303–318. Springer, 2011.

[12] R. C. De Boer and H. Van Vliet. Experiences with semantic wikis for architectural knowledge management. In *WICSA*, pages 32–41. IEEE, 2011.

[13] K. A. De Graaf, A. Tang, P. Liang, and H. Van Vliet. Ontology-based software architecture documentation. In *WICSA and ECSA, 2012 Joint Working IEEE/IFIP Conf. on*, pages 121–130. IEEE, 2012.

[14] R. Farenhorst and R. C. de Boer. Knowledge management in software architecture: State of the art. In *Soft. Architecture Knowl. Manag.*, pages 21–38. Springer, 2009.

[15] R. Farenhorst, P. Lago, and H. Van Vliet. Eagle: Effective tool support for sharing architectural knowledge. *Int. J. of Cooperative Inform. Syst.*, 16:413–437, 2007.

[16] F. Galiegue and K. Zyp. Json schema: Core definitions and terminology. *Internet Engineering Task Force (IETF)*, 2013.

[17] P. Liang and P. Avgeriou. Tools and technologies for architecture knowledge management. In *Soft. Architecture Knowl. Manag.*, pages 91–111. Springer, 2009.

[18] I. Lytra, H. Tran, and U. Zdun. Supporting consistency between architectural design decisions and component models through reusable architectural knowledge transformations. In *Software Architecture*, pages 224–239. Springer, 2013.

[19] C. Manteuffel, D. Tofan, H. Koziolek, T. Goldschmidt, and P. Avgeriou. Industrial implementation of a documentation framework for architectural decisions. In *WICSA*, pages 225–234. IEEE, 2014.

[20] F. Matthes and C. Neubert. Wiki4eam: Using hybrid wikis for enterprise architecture management. In *Proc. of the 7th Int. Symp. on Wikis and Open Collaboration*, pages 226–226. ACM, 2011.

[21] C. Miesbauer and R. Weinreich. Classification of design decisions–an expert survey in practice. In *Software Architecture*, pages 130–145. Springer, 2013.

[22] A. Osterwalder et al. The business model ontology: A proposition in a design science approach. 2004.

[23] T. Reschenhofer, M. Bhat, A. Hernandez-Mendez, and F. Matthes. Lessons learned in aligning data and model evolution in collaborative information systems. In *Companion Proc. of ICSE*, 2016 in press.

[24] T. Reschenhofer, I. Monahov, and F. Matthes. Type-safety in ea model analysis. In *EDOCW*, pages 87–94. IEEE, 2014.

[25] N. Schuster, O. Zimmermann, and C. Pautasso. Adkwik: Web 2.0 collaboration system for architectural decision engineering. In *SEKE*, pages 255–260. Citeseer, 2007.

[26] A. Tang, P. Avgeriou, A. Jansen, R. Capilla, and M. A. Babar. A comparative study of architecture knowledge manag. tools. *J. of Syst. and Soft.*, pages 352–370, 2010.

[27] A. Tang, P. Liang, V. Clerc, and H. van Vliet. Supporting co-evolving architectural requirements and design through traceability and reasoning. *Relating Software Requiremens and Software Architecture*, 2011.

[28] O. C. S. Workgroup. Oslc core specification version 2.0. *OSLC, Tech. Rep*, 2010.