Model-Driven Approach to Software Architecture Design

Daniel Perovich, María Cecilia Bastarrica, Cristián Rojas
Department of Computer Science
Universidad de Chile
{dperovic,cecilia,crirojas}@dcc.uchile.cl

Abstract

Software Architecture (SA) allows for early assessment of and design for quality attributes of a software system, and it plays a critical role in current software development. However, there is no consensus on fundamental issues such as design methods and representation organization and languages, and current proposals lack specificity and precision. Thus, it is extremely difficult to build a complete and appropriate software architecture, even though it is recognized as a fundamental artifact. In this paper we define an architecture design method that enables the systematic and assisted construction of the SA of Enterprise Applications, taking into account major quality attributes involved in this family of systems. We apply Model-Driven Engineering techniques to achieve this goal. The architecture is treated as a mega-model (a model composed of related models) and the application of design decisions is encoded in terms of model transformations. The architectural rationale is explicitly registered as the set of transformations that yields the complete SA from scratch. We illustrate the application of the approach by designing the SA of a case study from the literature.

1. Introduction

Since Perry and Wolf’s paper [19], an evolving community has actively studied the theoretical and practical aspects of Software Architecture (SA). In the years to follow, its adoption in industry has been broad and the research community has grown [3]. Software development processes have turned into architecture-centric either for dealing with complexity, risk management or effective resolution of quality attributes (QAs). SAs are built following Software Architecture Design Methods (SADMs), which mainly consist of three major activities [7, 10]: Requirement Analysis, Decision Making and Architectural Evaluation. Figure 1 depicts this general method. The Decision Making activity is intrinsically the core of a SADM; in it, requirements are resolved and the architecture is actually built. However, in order to be enacted successfully, the other two activities should be also carefully integrated [10]. There is a wide variety of SADMs, and while some provide general guidelines and checklists, others also offer QA resolution techniques [7]. However, no SADM is precise enough to encode all details on how a software architecture must be manipulated when performing an activity of the design method.

![Figure 1. General Software Architecture Design Method.](image)

The architect’s experience is still crucial for the success of architecture construction, even though architectural knowledge is widely reported in the literature. While a SADM encodes the knowledge on how to proceed to build an architecture, tactics and patterns encode the knowledge of well-known solutions to common problems or requirements. For example, N-tier and Client/Server are examples of enormously successful architectural patterns [23] widely used in industry. Tactics have less impact than patterns, but they are beginning to be used in industry [2]. While patterns resolve general QAs mainly from a logical perspective, tactics organize their application and have a broader architectural impact.

Therefore, the amount and variety of information that is...
necessaries to manage in order to make the right decisions while building the software architecture is huge, and thus, it is really complex to make these decisions, to keep track of them and to be able to eventually roll back some of these decisions. Although tool-support would be essential for systematic application of architectural knowledge on architecture design, and to explore different resolution alternatives, such tools are still lacking.

The IEEE 1471 Standard [1] has placed the concepts of Architectural View and Viewpoints as the crucial constituents of an architecture representation. However, there is no unified vision on which set of viewpoints must be used when deciding the particular view set for a system architecture. Several proposals of viewpoints are available [14, 20, 21], and some of them are particular to certain kinds of applications. Furthermore, the language constructs provided by each viewpoint for specifying a view are not agreed upon. While some authors position UML as the one-fits-all Architecture Description Language [23], other authors wonder to what extent it can be considered an ADL at all [9].

In this paper we present a systematic and tool-enabler approach for manipulating the software architecture when performing the Decision Making activity. It presents the following features:

i. it conforms to current architectural representation proposals by using mainly UML for architectural view representation,

ii. it encodes current architectural knowledge on quality attribute resolution,

iii. it is evolvable by enabling the inclusion of new knowledge,

iv. it enhances the separation of concerns, and

v. it preserves the architectural rationale and makes it traceable.

Even though it is hard to define such an approach for a general domain, it is feasible for particular types of systems. We present the case of Enterprise Applications. Not only this family of systemsshares the expected quality attributes and there are several proposed techniques to address them, but also specific architecture description proposals are available [21]. For other types of systems such as web applications or embedded systems, other attributes are relevant and also other view sets may be the most appropriate. We apply Model-Driven Engineering [22] techniques to specialize and enhance a SADM targeting Enterprise Applications. The architecture representation is treated as a mega-model organized in Architectural Views that are the constituent and related models, using Model-Driven Architecture to improve separation of concerns. Also, we understand the application of architectural decisions as model transformations which encode the architectural knowledge on QA resolution. Thus, the architectural rationale is explicitly recorded as the set of transformations that yields the complete SA from scratch.

The rest of the paper is structured as follows. Section 2 discusses related work. Section 3 describes the proposed approach and Section 4 illustrates its application to the design of the software architecture of a case study from the literature. Section 5 discusses architectural rationale representation. Finally, Section 6 states the conclusions and suggests some further work.

2. Related Work

Model-Driven Development. The Model-Driven Architecture (MDA) is a framework that separates platform independent from platform specific concerns to improve reusability, portability and interoperability of software systems. It guides the direction of model transformations from abstract to concrete models by incorporating technology-related details. In [24], Tekinerdoğan et al. consider MDA and Aspect-Orientation as complementary techniques for separation of concerns (SoC), and develop a systematic analysis of cross-cutting concerns within the MDA context. This work is strongly related to ours, but we use model transformations not only for refining elements in higher levels of abstractions into lower levels, but also for incrementally building the software architecture of a system and documenting its rationale. Also, in [8] Fuentes et al. identify the problems faced when applying these techniques to the development of distributed collaborative applications. However, in contrast to our work, theirs only deals with vertical model transformations.

The primary focus and work products of Model-Driven Engineering (MDE) are models, and combines Domain-Specific Languages (DSLs) and transformation engines and generators. These two mechanisms allow to encapsulate the knowledge of a particular domain. Software Architecture has benefited from DSLs as several Architecture Description Languages emerged in the last decade [4]. However, the application of MDE techniques has recently emerged in the discipline, and is mainly focused on MDA [15]. In [18] Merilinna works on horizontal model transformations at the platform independent abstraction level. He defines a language and supporting tool set for specifying model transformations mainly concerned with the application of architectural patterns. Besides, Matinlasi [17] aims automation of his Quality-driven Architecture Model Transformation approach. He focuses on transformations at the platform independent level of abstraction but is mainly concerned with
how the architecture model needs to be modified according
to changes or variations in the required quality properties.

Architectural Design Decisions. Virtually all decisions
during architectural design are implicitly present in the re-
sulting software architecture, lacking a first-class represen-
tation. Some approaches are emerging to overcome this problem. Jansen et al. [11] present the Archium approach
which defines the relationship between design decisions and
software architecture, proposing a meta-model for stating
such a relationship, currently providing tool support [12].
Dueñas et al. [6] study how to incorporate a Decision View
to architecture descriptions, mainly to Kruchten’s 4+1 Ar-
chitectural Framework. They identify requirements for such
a view and define the elements that are used to populate it.

All the previous approaches tackle views based on the
Component & Connector viewtype [5]. In contrast, our
approach deals with various viewpoints required in archi-
tecture description. Besides, we use MDE techniques not
only for easing architecture manipulation, but also for con-
structing the software architecture from scratch. Thus, the
sequence of applied model transformations is a first-class
mechanism for expressing design decisions, stating explic-
itly the architecture rationale.

3. Proposed Model-Driven Approach

A SADM is a process for designing a software architec-
ture from the needs and concerns of stakeholders, mainly
the expected system Quality Attributes (QAs). Several tech-
niques have been proposed for tackling each major activity
of such a process, being the Decision Making the most de-
manding task. Intuitive design approaches are effective in
organizing and processing requirements but they depend to
a large extent on the architect’s expertise to find solutions
that meet the QAs. In particular, the Attribute-Driven De-
sign (ADD) [25] method follows a recursive design process
in which a part of the system is selected for decomposi-
tion, architectural drivers are identified, architectural pat-
terns and tactics that satisfy them are applied, and pending
requirements are refined in terms of the new organization;
Figure 2 depicts the main steps of this method. The architect
incrementally builds the software architecture by iteratively
resolving the QAs. We define a specialization of the ADD
method, using Model-Driven Engineering techniques, that
systematizes and assists the Decision Making activity.

In order to effectively systematize the method in a way
that tool-support could be achieved, the architecture repre-
sentation is precisely stated. To this end, we use the pro-
sal of Rozanski et al. [21] for Enterprise Application soft-
ware architecture representation. They define six architec-
tural viewpoints, each addressing a cohesive set of archi-
tectural concerns: Functional, Information, Concurrency,
Development, Deployment, and Operational. Each View-

point is defined in terms of a set of models and activities
to create these models. Although the authors comment on
different notations for each viewpoint, no precise language
definition is provided. Then, we follow the recommenda-
tion in [5] that clearly states which kinds of elements can
be part of different types of views. When defining a model,
we select the viewtype that best suits the model intention.
We use UML notation for depicting models, and somehow
complement the language definitions provided by the view-
type approach. A precise definition in terms of the OMG’s
four-layer meta-modeling approach is part of the ongoing
work.

In order to enhance the SoC in the architecture represen-
tation, we apply additional techniques to improve modular-
ization. Following MDA, we structure architectural views
in three levels of abstraction. The most abstract level con-
ists of a Computation Independent perspective of the ar-
chitecture (CIA), mainly populated by the critical concerns
specified as functional and quality scenarios. The second
level consists of a Platform Independent perspective of the
architecture (PIA) in which those concerns are resolved
without taking into account the peculiarities of any underly-
ing platform. This level is organized in terms of views, and
they are built by applying patterns and tactics that address
the identified concerns. The bottom-most level provides a
Platform Specific perspective of the architecture (PSA). It
provides a technological solution to the abstract architec-
ture in the level above. To populate the PSA platform spe-
cific patterns are applied, as well as frameworks, middle-
ware and COTS are selected and incorporated. This verti-
cal division not only organizes architectural views, but also
separates platform independent from platform specific ar-
chitectural decisions. We illustrate in Figure 3 the Software
Architecture Model we propose.

In order to assist the decision making activity, we ap-
ply MDE techniques to automate the manipulation of the
architecture representation. To this end, we consider the ar-
chitecture representation as a mega-model that follows the
structure depicted in Figure 3. Each step of the recursive de-
sign method is encoded in terms of a model transformation
which transforms a version of the architecture into a subse-
quent one. Thus, given a significant QA to be addressed, a
particular architectural decision is made and hence the cor-
responding model transformation is applied, resulting in a new version of the software architecture in which the QA is resolved. Then, the method is understood as the successive application of model transformations, starting from an empty representation and ending with the complete architecture representation. Figure 4 illustrates this mechanism. Although architecture design is presented as a sequence of transformations, it can actually be organized in a tree structure, following the refinement of different architectural elements.

The sequence of model transformations is, by itself, an explicit representation of the architecture rationale. Thus, a model transformation is a first-class construct to represent an architectural decision. Furthermore, so as to integrate our approach in the contextual SADM, it is possible to define additional model transformations in order to also obtain other artifacts such as models, diagrams, and input artifacts for external tools. The possibility of automatically deriving a working system skeleton depends on the completeness of the Software Architecture Model built and the power of the available model transformations.

4. Applying our Approach

In order to exemplify the application of the defined approach, we address the design of the software architecture of the Point-of-Sale case study, originally presented in [16]. To this end, we follow the work direction suggested in Figure 3. First, we define the scenarios to be addressed in the Computation Independent Architecture. Second, we resolve these scenarios by applying our approach. After deciding which views we use to organize the Platform Independent Architecture, we follow the Attribute-Driven Design method sketched in Figure 2, particularly using our systematized approach based on model transformations depicted in Figure 4.

4.1. Computation Independent Architecture

The Point-of-Sale (POS) system is an Enterprise Application used, in part, to record sales and handle payments in a retail store. The POS is a realistic case study as retail stores and supermarkets do have computerized registers used by cashiers to sell goods to customers. Such a system usually includes hardware components such as a computer, a bar code scanner and receipt printers, and the software to run it. Also, it generally interfaces with external services such as third-party tax calculator and payment authorization systems. Even though many scenarios need to be defined to develop a realistic version of the POS system, we select a particular set of them that allows us to clearly illustrate the defined approach.
4.2. Platform Independent Architecture

Once the set of architectural significant scenarios is captured and documented in the CIA, the set of views for the PIA must be selected. We define three architectural views, namely Functional, Information and Deployment, based on the homonymous Viewpoints proposed by Rozanski et al. Next, following the ADD method, we address each of the scenarios documented in the CIA.

FS1: Process Sale. This scenario describes the user-system interaction to append a new sale to the system. A thorough specification of this scenario is built by means of an information structure and information flow models. While the former is expressed in terms of conceptual classes and relationships, the latter uses a state machine; Figure 5 depicts the state machine for this scenario. Then, the first model transformation to be applied is such that incorporates both models to the Information View of the architecture; this transformation mainly clones the input model into the architecture model. Notice that model transformations encoding Fowler’s Analysis Patterns may be defined and applied to build the Information View.

QS1 & QS2: Persist Sale Data & Multiple Front-End Devices. Considering these two quality scenarios, a three-layer architecture is decided to organize the Functional Structure Model of the Functional View; Figure 6 illustrates this model. A model transformation is used to decompose the entire system in terms of three components following the Layers pattern. We further refine this first organization following Fowler’s enterprise application architectural patterns that suggest different approaches to structure each of the layers. First, provided the complexity of the POS domain, we decide the joint use of the Table Module pattern to organize the Domain layer and the Table Data Gateway pattern to organize the data access part of the Infrastructure layer. Then, two model transformations are applied to achieve such a refinement. They not only consider the current Functional Structure Model of the Functional View, but also the Information Structure Model of the Information View which defines the major concepts to be managed. Thus, a Table Module and a Table Data Gateway component for each concept populates the two layers. Finally, provided QS2, different front-end components are defined. We follow the Page Controller pattern for easing development and apply the Application Controller pattern to factor out common behavior of the page controllers. All these decisions are enforced by successively applying model transformations that refine a single component into a set of interconnected components that embodies/materializes the decision made.

In turn, a distributed runtime platform is also decided separating front-end from back-end processing. We apply a model transformation that organizes the Runtime Plat-
form Model of the Deployment View in terms of the client/server distribution pattern. We actually decided to split the back-end in an application and a database server dedicated nodes. QS2 renders the need for in-site workstations (Register node) and a web server dedicated node for attending different thin-clients. Figure 7 illustrates the Run-time Platform Model. Different input and output devices for the Register node are decided following the Process Sale (FS1) functional scenario.

Figure 7. Runtime Platform Model.

QS3: Mandatory User Authentication. To address QS3, we first identify the types of resources that need to be protected, together with the actions that can be made on them. Resources and actions can be obtained from the other models in the Information View by means of model transformations. A Security Resources Model is built to this end. Afterwards, principals are identified together with the assigned permissions with respect to the defined resources. Then, a Security Policies Model is built. Figure 8 and Figure 9 illustrate each of these models.

Figure 8. Security Resource Model.

Figure 9. Security Policy Model.

Then, we apply a model transformation that automatically appends a sign-in and sign-out process to the Information Flow Model; such transformation appends the model elements illustrated in Figure 10. The transformation also records the composition rules for this view: additional components in the presentation are required, the Application Controller will require sign-in if there is no current user, security information data must be preserved by the system. Then, this aspect can later be weaved into the Functional View by another model transformation.

5. Architecture Rationale

At each step of the defined approach, an architectural concern is addressed and a set of architectural decisions is made. The architecture mega-model is automatically updated by applying the model transformations corresponding to such decisions. The sequence of applied transformations
is itself the rationale of the architecture built.

Although originally proposed in the Domain Analysis area and rarely used in the Software Architecture discipline, Feature Models proved to be useful for us when classifying design alternatives. Feature Models’ ability to express variability allows us to concisely define the set of alternative architectural mechanisms that can be used. A Feature Model consists of one or more Feature Diagrams (first level elements) which organize features into hierarchies. The Feature Model renders a tree which expressively states variability such as optional features (grey dots) or selection (grouped squares). A Feature Configuration is an instance of a Feature Model in which particular alternatives are selected, i.e. no variability remains. Then, a Feature Configuration can embody a representation of the rationale that yields the complete architecture.

Figure 11. Deployment Decisions.

Figure 11 illustrates the Feature Model with all possible design decisions with respect to the Deployment View. It states that the view consists of a Runtime Platform model consisting of the Distribution of computational nodes; only a Client/Server distribution is shown in the diagram. Such a distribution enables several rich clients possible holding devices, and several thin clients. In turn, servers can include a web server, an application server, and a database server dedicated node. Figure 12 presents the rationale for the POS System. The particular Feature Configuration uses a Client/Server distribution, one rich client with four devices and three thin clients were decided. Also, one server of each kind was selected, including two external providers to the application server. This configuration resumes the decisions made and can be straightforwardly mapped to the architectural elements present in the Runtime Platform Model depicted in Figure 7.

Figure 12. Deployment Rationale.

6. Conclusions & Further Work

Architecture design is a creative task in which tradeoffs among different alternatives strongly rely on the architect’s experience, and are generally conditioned by resource constraints. For these reasons, a fully automated method seems unfeasible now. However, this activity can be systematized so as to enable actual tool-assistance that automates repetitive tasks. Such automation not only reduces the architecture design effort, but also it eases the exploration and evaluation of different architectural design alternatives.

Our approach conceives the architecture representation as a mega-model, understanding it as a well-structured self-contained representation of the system, expressed in a precise language. In this context, the architecture design activity can be seen as a large model transformation which obtains, from an initially empty architecture, the complete system architecture. This large transformation is composed of a sequence of smaller sub-transformations, each encapsulating the application of a design decision, i.e. the resolution of a particular architectural concern. It is an interactive transformation as the software architect selects which sub-transformation to apply next. Then, the set of sub-transformations available to the architect can be regarded as the definition of a family of large transformations, i.e.
as all the possible ways to design the complete architecture from scratch. Thus, by incorporating additional subtransformations to this set, a large number of architectures can be designed using the method. Then, the application of Model-Driven Engineering techniques not only favors the evolution of the approach, but also increases its power.

By using Model-Driven Architecture as an additional mechanism for separation of concerns, we might be making the architecture representation more complex and thus hindering comprehensibility. However, using MDA not only favors modularization and reuse, but also organizes and systematizes the architect’s task. Besides, the architect must learn additional DSLs as precise ones are required to enable automation; however, model transformations ease the usage of such languages as views are automatically manipulated.

Feature Models proved to be useful for representing architecture design alternatives, being each feature a particular tactic or pattern that addresses a given concern. So, the Feature Model describes the power of the designs that can be achieved. Then, Feature Configurations embody a first-class representation for the architecture rationale. Furthermore, such a Feature Configuration can be used by a tool to automatically apply all decisions made (i.e. all the model transformations corresponding to the selected features) obtaining the corresponding architecture design.

We have developed a tool prototype using Eclipse and ATL [13]. As further work, we plan to develop a Computer-Aided Software Architecture Design Environment that deals with architecture representation and that provides an evolvable set of model transformations. Also, we are formalizing the DSLs for architecture representation following OMG’s four layer meta-modeling approach. By these means, we enable the applicability of model transformation languages and tools, most based on OMG’s Meta-Object Facility (MOF), easing the codification and incorporation of new architectural knowledge to the tool.

References