Aspect-Oriented Model-Driven Approach to Architecture Design

Daniel Perovich*, María Cecilia Bastarrica, and Cristian 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 is playing 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 preciseness. In this paper we define an architecture design method that enables the systematic and assisted construction of the software architecture of Enterprise Applications, taking into account major quality attributes involved in this family of systems. We apply Model-Driven Engineering and Aspect-Oriented techniques to achieve this goal. The architecture is treated as a model organized in Architectural Views, using aspects to improve separation of concerns, and encoding the application of design decisions 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 [22], 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 [4]. 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 (SADM), which mainly consist of three major activities [9, 12]: 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 carefully integrated [12]. There is a wide variety of SADMs, and while some provide general guidelines and checklists, others also offer QA resolution techniques [9]. However, no SADM is precise enough to encode all details on how a software architecture must be manipulated when performing an activity

* The work of D. Perovich was partially funded by CONICYT Chile.

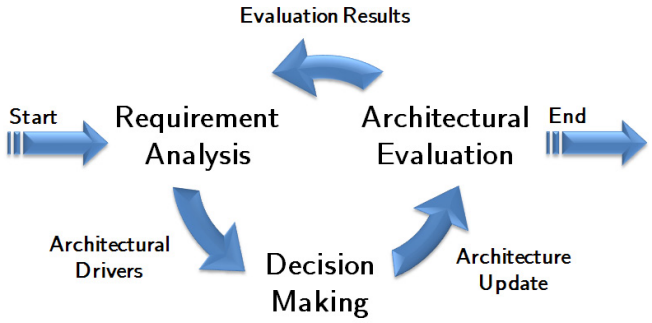

Fig. 1. General Software Architecture Design Method.

of the design method and in some cases these details are somehow delegated to a companion tool set. The architect's experience is 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, patterns and tactics encode the knowledge of well-known solutions to common problems or requirements. N-tier and Client/Server among others, are examples of enormously successful architectural patterns [28]. 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 have a broader architectural impact. However, tool support is essential to ease their systematic application on architecture representation, and to explore different resolution alternatives.

The IEEE 1471 Standard [1] has placed the concepts of Architectural View and Viewpoints as the crucial constituents of an architecture representation. Though, 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 [16, 23, 25], and some of them are particular to a certain kind of applications. Furthermore, language constructs provided by each viewpoint for specifying a view are not agreed upon. Although some authors position UML as the one-fits-all architecture description language (ADL) [28], other authors wonder to what extent it can be considered an ADL by itself [11].

An Architectural View tackles a particular set of concerns [25], and hence, there are concerns that can be traced directly into a particular view. However, architecture organization in terms of views provides a single mechanism for decomposition and modularization. Then, to choose a specific set of views implies that several concerns may cross-cut such division. Thus, while some concerns are scattered throughout various views, views may end up incorporating tangled concerns, yielding the tyranny of the dominant decomposition [30]. Consequently, QA resolution generally spans across views and then, to resolve a QA implies to update all pertinent views to incorporate the desired solution. Some techniques, mainly inspired in Aspect-Orientation [24, 30], are emerging and improving sep-

aration of concerns at the architectural level. Moreover, architectural decisions cross-cut the architecture representation, and such decisions lack first-class representation and they are usually lost during architecture construction [4].

In this paper we present a systematic and tool-enabler approach for manipulating the software architecture when performing the Decision Making activity that presents the following features: (i) it conforms to current architectural representation proposals, (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 Enterprise Applications. Not only this family of systems share the expected quality attributes and there are several proposed techniques to address them, but also specific architecture description proposals are available [25]. We apply Model-Driven Engineering [27] and Aspect-Oriented techniques to specialize and enhance a SADM targeting this family of systems. The architecture representation is treated as a model organized in Architectural Views, using Model-Driven Architecture and Early Aspects 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 taken from the literature. Finally, Section 5 summarizes the conclusions and suggests directions for further work.

2 Related Work

Separation of Concerns. SoC is achieved by means of modularization and relies on hiding significant decisions from each other, mainly those which are likely to change. The ability to achieve quality and to facilitate maintenance and evolution depends on the ability to keep separate all relevant concerns [30].

Aspect-oriented (AO) approaches propose an additional asymmetrical decomposition mechanism, called *aspect*, to encapsulate cross-cutting concerns which can be composed non-invasively. Early Aspects approaches focus on the requirement and architectural level. [3] proposes to factor out concerns that cross-cuts the architectural views in aspectual views, and in [26] a partially automated process for deriving an AO software architecture from AO requirements using Model-Driven techniques.

Multi-dimensional SoC approaches use dimensions to encapsulate concerns. As such dimensions can overlap by sharing constituent elements, composition rules are also provided. In [21] Moreira et al. apply this approach to Requirement Engineering providing a uniform treatment of concerns regardless of their nature and tackling traceability to architectural choices. Besides, Kandé et al.

in [15] apply this technique at the architectural level. Also, Suvée et al. [29] propose a component architecture and a description language, namely FuseJ, which allows the specification and construction of component based architectures. The symmetric nature of the modularization technique may hinder comprehensibility. In contrast, our approach is asymmetric as those of Early Aspects.

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 [31], Tekinerdoğan et al. consider MDA and AO as complementary techniques for 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 [10] 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.

MDD's primary focus and work products 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 [5]. However, the application of model-driven techniques are recently emerging in the discipline, and are mainly focused on MDA [17]. In [20] 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, Matinlassi [19] 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 on how the architecture model needs to be modified accordingly to changes or variations in the required quality properties.

Architectural Design Decisions. Virtually all decisions m during architectural design are implicitly present in the resulting software architecture, lacking a first-class representation. Some approaches are emerging to overcome this problem. Jansen et al. [13] present the Archium approach which defines the relationship between design decisions and software architecture, proposing a meta-model for stating such relationship, currently providing tool support [14]. Dueñas et al. [8] study how to incorporate a Decision View to architecture descriptions, mainly to Kruchten's 4+1 Architectural Framework. They identify requirements for such a view and define the elements that can be used to populate it.

All the previous approaches tackle views based on the Component & Connector viewtype [6]. In contrast, our approach deals with various viewpoints required in architecture description. Besides, we use MDD techniques for easing

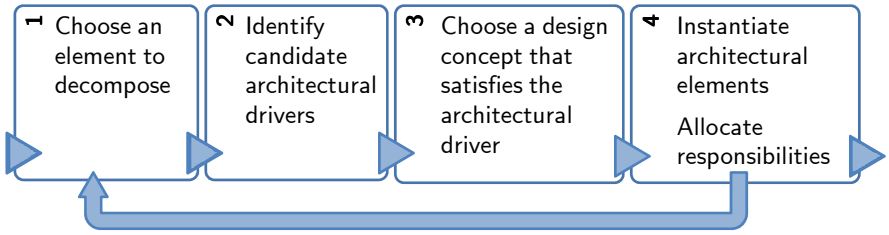

Fig. 2. Steps of the Attribute Driven Design method.

architecture manipulation, and also for constructing the software architecture from scratch. Thus, the sequence of applied model transformations is a first-class mechanism for expressing design decisions, explicitly stating the architecture rationale.

3 Aspect-Oriented Model-Driven Approach

A SADM is a process for designing a software architecture from the needs and concerns of stakeholders, mainly the expected system QAs. Several techniques have been proposed for tackling each major activity of such a process, being the Decision Making the most demanding task. Intuitive design approaches are effective in organizing and processing requirements but depend to a large extent on the architect to find solutions that meet the QAs. In particular, the Attribute-Driven Design (ADD) [32] method follows a recursive design process in which a part of the system is selected for decomposition, architectural drivers are identified, architectural patterns 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 constructs the software architecture by iteratively resolving the QAs. We define a specialization of the ADD method 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 representation is precisely stated. To this end, we use Rozanski et al. proposal [25] for Enterprise Application software architecture representation. They define six Architectural Viewpoints, each addressing a cohesive set of architectural concerns: Functional, Information, Concurrency, Development, Deployment, and Operational. Each Viewpoint 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 recommendation in [6] 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 defini-

tions provided by the viewtype approach. A precise definition in terms of the OMG's four-layer meta-modeling approach is suggested as further work.

In order to enhance the SoC in the architecture representation, we apply additional techniques to improve modularization. Following MDA, we structure architectural views in three levels of abstraction. The most abstract level consist of a Computation Independent perspective of the architecture (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 underlying platform. This level is organized in terms of views, and they are constructed by applying patterns and tactics that address the identified concerns. The bottom-most level provides a Platform Specific perspective of the architecture (PSA). It embraces a technological solution to the abstract architecture in the upper level. To populate the PSA platform specific patterns are applied, as well as frameworks, middleware and COTS are selected and incorporated. This vertical division not only organizes architectural views, but also separates platform independent from platform specific architectural decisions.

Orthogonal to this vertical division, we provide an horizontal mechanism of modularization by applying EA techniques to enhance SoC. At the CIA level of abstraction, we use aspectual scenarios to factorize and/or isolate requirements that cross-cut the core scenarios. The impact of aspectual scenarios on core scenarios is also included in the architecture representation. In the PIA and PSA levels, we append Aspectual Views to the architectural representation. An Aspectual View focuses and describes in isolation a particular architectural concern that otherwise would cross-cut the architecture representation. Our application of aspect-orientation is asymmetric as we consider traditional views as the core architectural representation, and aspectual views as separate views which refer to elements in the core views. Hence, a glue section is also needed in order to indicate how the aspectual views are bound to the core views. Different versions of the PIA and PSA levels might be automatically obtained by using weaving techniques in which the aspectual views are weaved into the core views. Particularly, when building the PSA, while certain aspectual views at the PIA level may be used, others may not be present as the concern they deal with is completely addressed by a particular framework or COTS. We illustrate in Figure 3 the Software Architecture Model we propose.

In order to assist the design of the decision making activity, we apply MDE techniques to automate the manipulation of the architecture representation. To this end, we consider the architecture representation as a model expressed in DSLs that follow the structures depicted in Figure 3. Then, each step of the recursive design method is encoded in terms of a model transformation which transforms a version of the architecture into a subsequent one. Thus, given a significant QA to be addressed, a particular architectural decision is made and hence the corresponding model transformation is applied, resulting in a new version of the SA in which the QA is resolved. Then, the method is understood

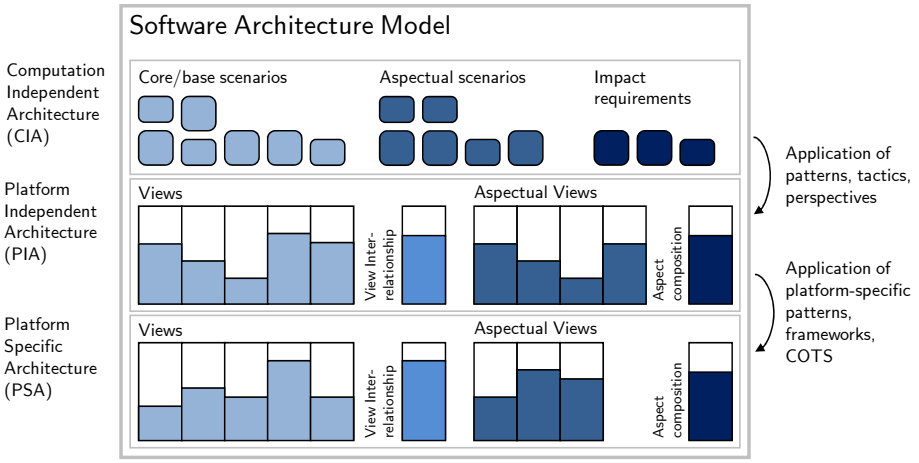

Fig. 3. Software Architecture Model for architecture description.

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, they can actually be organized in a tree structure, following the refinement of 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 citizen construct to represent an architectural decision. Furthermore, so as to integrate our approach in the contextual SADM, by defining additional model transformations to obtain other artifacts such as models, diagrams, and input artifacts for external tools. The automatic derivation of a working system skeleton depends of the completeness of the Software Architecture Model built and the power of the available model transformations.

4 Applying the 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 [18]. 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, and particularly using our systematized approach based on model transformations depicted in Figure 4. For the sake of space, we do not illustrate all intermediate steps.

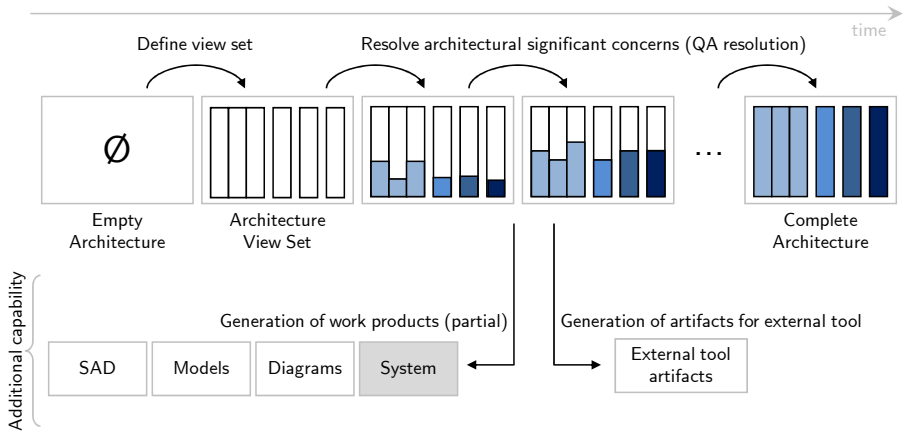

Fig. 4. Architecture Decision Making activity.

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; we provide a synopsis next.

Scenarios

FS1: Process Sale. A customer arrives at a checkout with items to purchase. The cashier uses the POS system to record each purchased item. The system presents a running total and line-item details. The customer enters discounts, coupons and payment information, which the system validates and records. The system updates inventory. The customer receives a receipt from the system and then leaves with the items.

QS1: Persist Sale Data. The POS system must persist the sale information between successive executions of the system. Sales data include date, item description, discounts and coupons if used, and payment information.

QS2: Multiple Front-End Devices. A POS system must support multiple and varied client-side terminals and interfaces. These include a thin-client Web browser terminal, a regular personal computer with something like a form-based graphical user interface, touch screen input, wireless PDAs, and so forth.

QS3: Mandatory User Authentication. The POS system accepts requests from users only after they are authenticated.

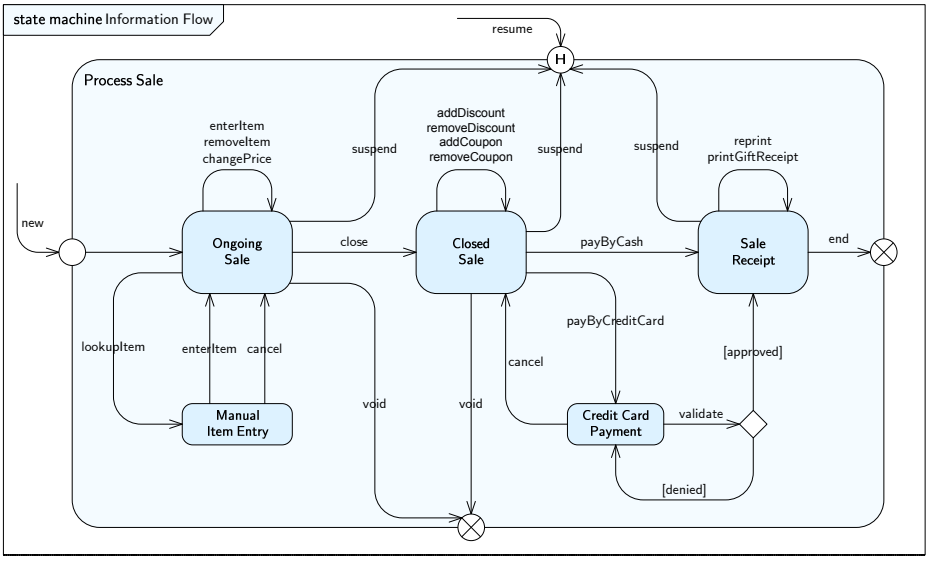

Fig. 5. Information Flow Model in the Information View.

AS1: Enable Authorization Overriding. Some requests can only be placed by privileged users. For instance, removing items, discounts, and coupons, or voiding the ongoing sale, can only be done by a supervisor, not by the cashier. The POS system must allow authorization overriding for these particular request without signing out the current user.

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 of Rozanski et al. proposal. In addition, we define a Security aspectual view to deal with this concern in isolation. 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.

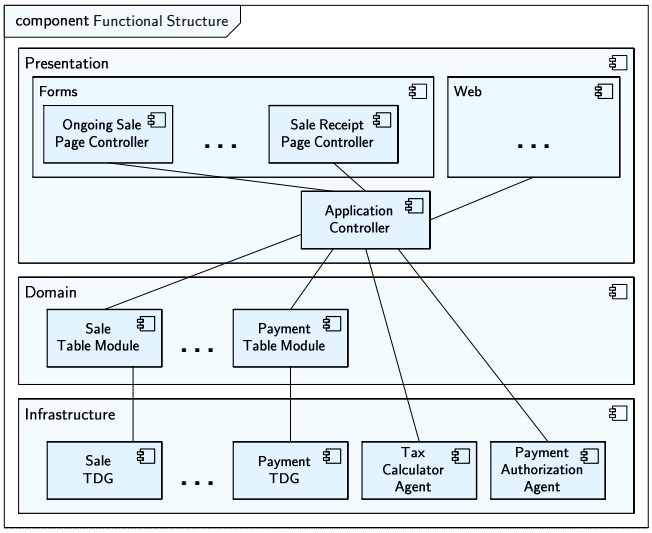

Fig. 6. Functional Structure Model in the Functional View.

QS1 & QS2: Persist Sale Data & Multiple Front-End Devices. Considering these two quality scenarios, a three-layer architecture is decided to organized the Functional Structure Model of the Functional View; Figure 6 illustrate 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. These patterns 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-ends 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.

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 Platform 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

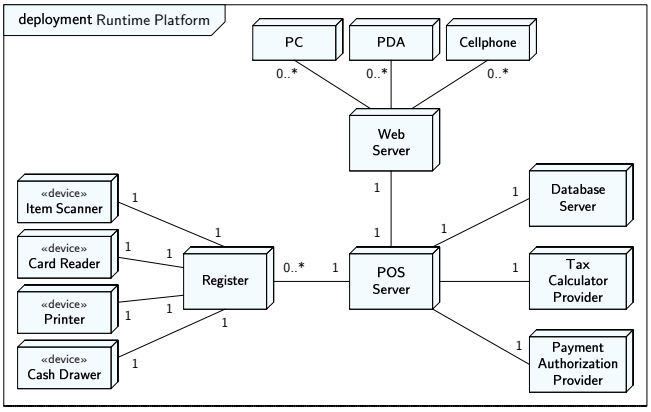

Fig. 7. Runtime Platform Model in the Deployment View.

thin-clients. Figure 7 illustrates the Runtime Platform Model. Different input and output devices for the Register node are decided following the Process Sale (FS1) functional scenario.

QS3: Mandatory User Authentication. We specify security related concerns in the Security Aspectual View, and we use the *Secure UML* profile as the specification language. 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 are deduced by the architect from the models in the Information View. 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. A model transformation is used to incorporate this models into the Security View; Figure 8 illustrates both models.

To address mandatory user authentication we apply the single sign-on tactic. Then, we apply a model transformation that automatically appends a sign-in and sign-out process to the Security View. Similar to the state machine in the Information Flow Model, the transformation uses a state machine to state how these processes proceed; it is illustrated in Figure 9. The transformation also records the composition rules for this view: additional components in the presentation are required, the Application Controller must 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.

AS1: Enable Authorization Overriding. Finally, to address AS1 we apply a model transformation that encodes the knowledge of how to provide authorization override. Such transformation appends a parameterized state machine to the Security Aspectual View and the binding information relies on the Security Policies Model. Figure 10 depicts this state machine. Weaving this aspect

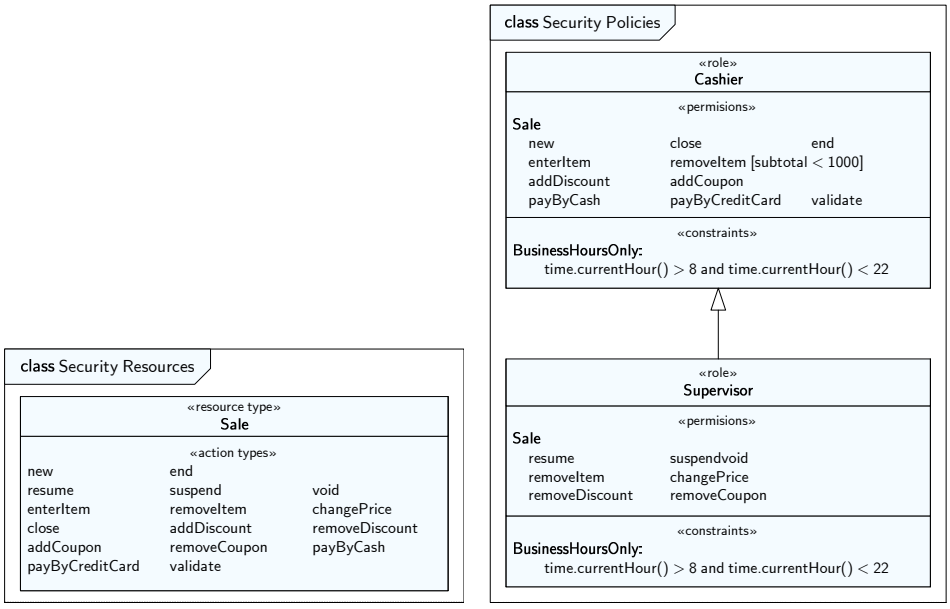

Fig. 8. Security Resources and Policies Models in the Security View.

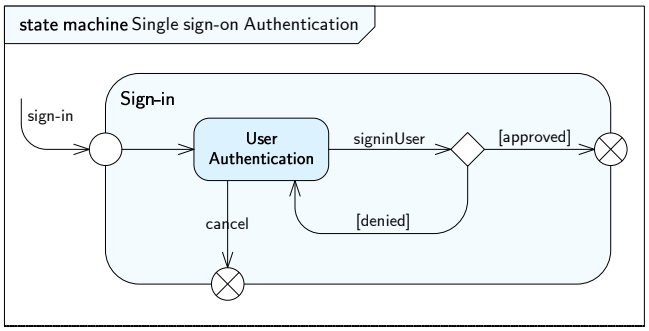

Fig. 9. Single Sing-On Security Aspect.

into the Functional View implies to append this responsibility to the Application Controller.

Decisions for the Deployment View. At each step of the defined approach, an architectural concern is addressed and a set of architectural decisions are made. The architecture model is automatically updated by applying the model transformations corresponding to such decisions. The set of applied transformations is itself the rationale of the architecture built. We use a Feature Model diagram to illustrate such a rationale. A Feature Model consists of one or more

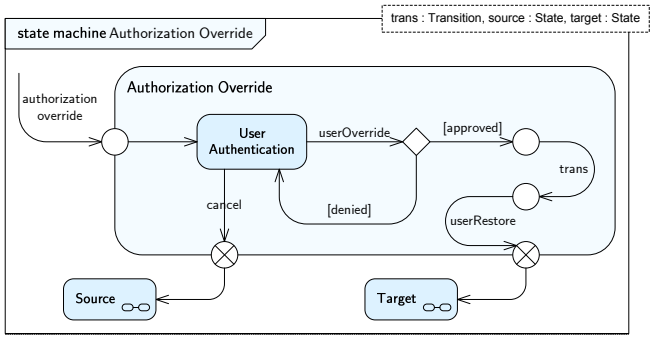

Fig. 10. Authorization Override Aspect.

Feature Diagrams (first level elements) which organizes 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 in which a particular alternative is selected.

The Feature Model to the left of Figure 11 depicts all possible design decisions with respect to the Deployment View. It states that the view consists of a Runtime Platform model which uses a *Distribution*. Currently, only a Client/Server distribution is provided 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. At the right of Figure 11, the particular Feature Configuration for the POS is illustrated. Client/Server distribution is used, 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 clearly mapped into the architectural elements present in the Deployment View.

5 Conclusions & Further Work

Architecture design is a creative task in which tradeoffs among different alternatives strongly rely on the architect experience and require some extent of creativity, and are generally conditioned by resource constraints. For these reason, a fully automated method seems unfeasible. 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 design alternatives.

Our approach conceives the architecture representation as a 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 initial empty

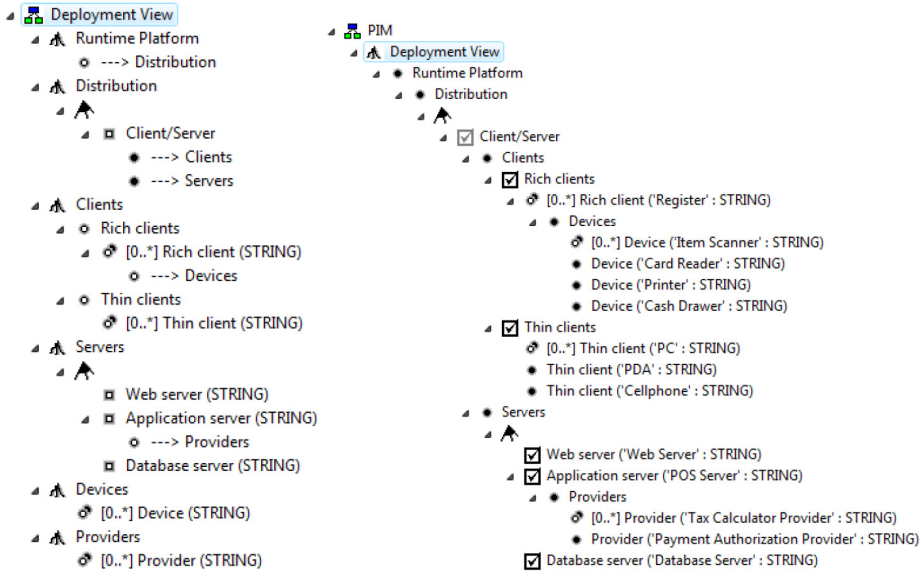

Fig. 11. Deployment Decisions.

architecture the complete architecture for the system. 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 sub-transformations to this set, a large number of architectures can be designed using the method. Then, the application of Model-Driven techniques not only favors the evolution of the approach, but also increases its power.

Although originally proposed in the Domain Analysis area and rarely used in the SA discipline, features models proved to be useful when classifying design alternatives. As we illustrated in the application of the approach, feature models' ability to express variability allows us to concisely define the set of alternative architectural mechanisms that can be used. Then, a configuration of a feature model, i.e. a feature model in which no variants remain, can be seen as a representation of the rationale that yields the complete architecture. In this context, we envision that feature models can be useful to describe the design power of the family of sub-transformations, and hence, as a mechanism for defining the particular large transformation. We foresee that our approach can result in a method for building software product lines [7] in which variation points are present in the variety of alternatives available to resolve quality attributes.

By using additional mechanisms for separation of concerns, such as MDA and EA, we may be making the architecture representation more complex and thus hindering comprehensibility. However, the approach not only favors modularization and reuse, but also organizes and systematizes the architect's task. Moreover, 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 built.

Directions for further work include to provide actual tool support for the approach. 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 plan to formalize the DSLs for architecture representation and particularly the aspectual views, mainly following OMG's four-layer meta-modeling architecture 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 new architectural knowledge to the tool.

References

1. IEEE Std 1471-2000, IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, 2000.
2. F. Bachmann, L. Bass, and M. Klein. Deriving Architectural Tactics: A Step Toward Methodical Architectural Design. Technical Report CMU/SEI-2003-TR-04, Software Engineering Institute, Carnegie Mellon University, 2003.
3. E. Baniassad, P. C. Clements, J. Araújo, A. Moreira, A. Rashid, and B. Tekinerdoğan. Discovering Early Aspects. *IEEE Software*, 23(1):61–70, 2006.
4. J. Bosch. Software Architecture: The Next Step. In *EWSA'2004*, pages 194–199, 2004.
5. P. C. Clements. A Survey of Architecture Description Languages. In *IWSSD'1996*, pages 16–25. IEEE Computer Society, 1996.
6. P. C. Clements, D. Garlan, L. Bass, J. Stafford, R. L. Nord, J. Ivers, and R. Little. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2002.
7. P. C. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 2001.
8. J. C. Dueñas and R. Capilla. The Decision View of Software Architecture. In *EWSA'2005*, pages 222–230, 2005.
9. D. Falessi, G. Cantone, and P. Kruchten. Do Architecture Design Methods Meet Architects' Needs? In *WICSA'2007*, page 5, 2007.
10. L. Fuentes, M. Pinto, and A. Vallecillo. How mda can help designing component- and aspect-based applications. In *EDOC 2003*, pages 124–135. IEEE Computer Society, 2003.
11. D. Garlan, S.-W. Cheng, and A. J. Kompanek. Reconciling the Needs of Architectural Description with Object-Modeling Notations. *Science of Computer Programming*, 44(1):23–49, 2002.
12. C. Hofmeister, P. Kruchten, R. L. Nord, J. H. Obbink, A. Ran, and P. America. Generalizing a Model of Software Architecture Design from Five Industrial Approaches. In *WICSA'2005*, pages 77–88, 2005.

13. A. Jansen and J. Bosch. Software Architecture as a Set of Architectural Design Decisions. In *WICSA'2005*, pages 109–120, 2005.
14. A. Jansen, J. van der Ven, P. Avgeriou, and D. K. Hammer. Tool support for Architectural Decisions. In *WICSA'2007*, page 4, 2007.
15. M. M. Kandé and A. Strohmeier. On the Role of Multi-Dimensional Separation of Concerns in Software Architecture. In *OOPSLA'2000*, 2000.
16. P. Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–50, 1995.
17. P. Kruchten, J. H. Obbink, and J. Stafford. The Past, Present, and Future for Software Architecture. *IEEE Software*, 23(2):22–30, 2006.
18. C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall PTR, third edition, October 2004.
19. M. Matinlassi. *Quality-driven software architecture model transformation: Towards automation*. PhD thesis, ESPOO: VTT Technical Research Centre of Finland, 2006. VTT Publications 608.
20. J. Merilinna. A Tool for Quality-Driven Architecture Model Transformation. Master's thesis, ESPOO: VTT Technical Research Centre of Finland, 2005. VTT Publications 561, ISBN 951-38-6439-1;951-38-6440-5.
21. A. Moreira, A. Rashid, and J. Araújo. Multi-Dimensional Separation of Concerns in Requirements Engineering. In *RE'2005*, pages 285–296. IEEE Computer Society, 2005.
22. D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.
23. J. R. Putman. *Architecting with RM-ODP*. Prentice Hall PTR, 2000.
24. A. Rashid, A. Moreira, and B. Tekinerdoğan. Early Aspects: Aspect-oriented Requirements Engineering and Architecture Design. *IEE Proceedings - Software*, 151(4):153–156, 2004.
25. N. Rozanski and E. Woods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional, 2005.
26. P. Sánchez, J. Magno, L. Fuentes, A. Moreira, and J. Araújo. Towards mdd transformations from ao requirements into ao architecture. In *EWSA 2006*, volume 4344 of *Lecture Notes in Computer Science*, pages 159–174. Springer, 2006.
27. D. C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.
28. M. Shaw and P. C. Clements. The Golden Age of Software Architecture. *IEEE Software*, 23(2):31–39, 2006.
29. D. Suvée, B. D. Fraine, and W. Vanderperren. A Symmetric and Unified Approach Towards Combining Aspect-Oriented and Component-Based Software Development. In *CBSE'2006*, pages 114–122, 2006.
30. P. L. Tarr, H. Ossher, W. Harrison, and S. M. S. Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *ICSE'1999*, pages 107–119, 1999.
31. B. Tekinerdoğan, M. Aksit, and F. Henninger. Impact of Evolution of Concerns in the Model-Driven Architecture Design Approach. *ENTCS*, 163(2):45–64, 2007.
32. R. Wojcik, F. Bachmann, L. Bass, P. C. Clements, P. Merson, R. L. Nord, and B. Wood. Attribute-Driven Design (ADD), Version 2.0. Technical Report CMU/SEI-2006-TR-023, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, 2006.