ISSN:0975-9646

Model Driven Design Method for Software Architecture

Thodeti Srikanth, Dachepally Ravi Kumar , Mahi Naveen Kumar
Research Scholars, Ph.D. (Computer Science),
Dravidian University, Andhra Pradesh, INDIA

Abstract— **Software Architecture allows for early assessment of and design for quality attributes of a software system. It provides an important help for current software development. The development of software architecture is complex due to the absence of a standard way that lead the generation of software architecture artifacts. In this paper we define an architecture design method that provides the systematic method for software architecture of business application. We apply model driven engineering techniques to achieve the goal, the architecture is treated as a model composed of related models and application of design decision is encoded in terms of model transformation. We define a specialization of the attribute driven design(Add) method using model driven engineering techniques that systematizes and assists the Decision Making activity.**

Keywords— Model Driven Architecture, Model Driven Development, UML, Model Driven Architecture tools etc.

INTRODUCTION

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 [5, 7]: Requirement Analysis, Decision Making and Architectural Evaluation. Figure 1 depicts this general method. Perry and Wolf's paper [12], 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 [2]. There is a wide variety of SADMs, and while some provide general guidelines and checklists, others also offer QA resolution techniques [5]. However, no SADM is precise enough to encode all details on how software architecture must be manipulated when performing an activity of the design method.

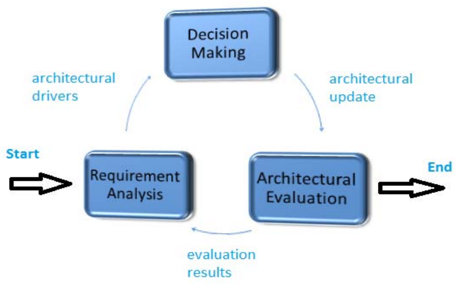


Figure 1. General Software Architecture Design Method.

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 [16] widely used in industry.

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[10, 13, 14], and some of them are particular to certain kinds of applications. For specifying a view, the language constructs provided by each viewpoint is not agreed upon. Some authors position UML as the one-fits-all Architecture Description Language [16], other authors wonder to what extent it can be considered an ADL at all [6].

In this paper, we present a systematic and tool-enabler Design method 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
v. it preserves the architectural rationale and makes it traceable.

We present the case of Enterprise Applications. Not only this family of systems shares the expected quality attributes and there are several proposed techniques to address them, but also specific architecture description proposals are available [14].We apply Model-Driven Engineering [15] 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.

The rest of the paper is structured as follows: (i)- Related work, (ii)- proposed method, (iii)- illustrates its application to the design of the software architecture of a case study from the literature, (iv)- architectural rationale representation, (v)- states the conclusions.

RELATED WORK

Model-Driven Development. The OMG's Model Driven Architecture initiative is aimed at increasing productivity and re-use through separation of concern and abstraction. A Platform Independent Model (PIM) is an abstract model which contains enough information to drive one or more Platform Specific Models (PSM). Possible PSM artifacts may include source code, DDL, configuration files, XML and other output specific to the target platform. MDA aims to enhance portability by way of separating system (abstract) architecture from platform (concrete) architecture. Platform Independent Models describe the structure and function of a system, but not the specific implementation. MDA has the capability to define transformations that map from PIMs to PSMs. In [17], Tekinerdo˘gan 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. 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. A model is a formal specification of the function, structure and behaviour of a system within a given context, and from a specific point of view (or reference point). A model is often represented by a combination of drawings and text, typically using a formal notation such as UML, augmented where appropriate with natural language expressions.

Architectural Design Decisions. Virtually all decisions during architectural design are implicitly present in the resulting software architecture, lacking a first-class representation. The architecture of a system is a specification of the parts and connectors of the system and the rules for the interactions of the parts using the connectors.

Some approaches are emerging to overcome this problem. Jansen et al. [8] 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 [9]. Due˜nas et al. [4] 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 are used to populate it. All the previous approaches tackle views based on the Component & Connector view-type [3].

In contrast, our approach deals with various viewpoints required in architecture description. Besides, we use MDE techniques not only for easing architecture manipulation, but also for constructing the software architecture from scratch. Thus, the sequence of applied model transformations is a first-class mechanism for expressing design decisions, stating explicitly the architecture rationale.

PROPOSED MODEL-DRIVEN DESIGN METHOD

A MDDM is a process for designing a software architecture from the needs and concerns of stakeholders, mainly the expected system Quality Attributes (QAs). Several techniques have been proposed for tackling each major activity of such a process, being the Decision Making the most demanding task. In particular, the Attribute-Driven Design (ADD) [18] method takes as input a set of quality attribute scenarios and employs knowledge about the relation between quality attribute achievement and architecture in order to design the architecture. The ADD method can be viewed as an extension to most other development methods, such as the Rational Unified Process. ADD is an approach to defining a software architecture that bases the decomposition process on the quality attributes the software has to fulfill.

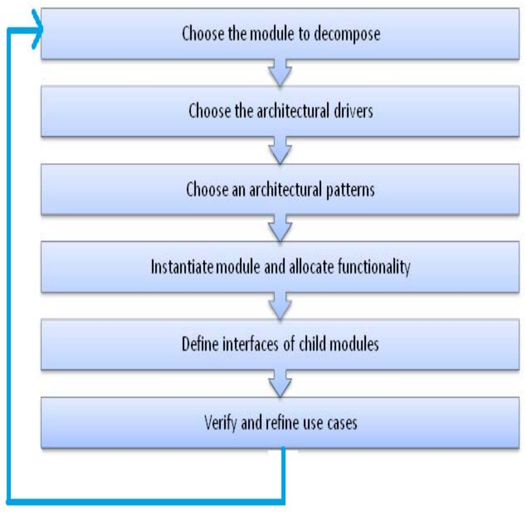

Choose the module to decompose
Choose the architectural drivers
Choose an architectural patterns
Instantiate module and allocate functionality
Define interfaces of child modules
Verify and refine use cases

Figure 2. Steps of the Attribute Driven Design method.

It is a recursive decomposition process where, at each stage, tactics and architectural patterns are chosen to satisfy a set of quality scenarios and then functionality is allocated to instantiate the module types provided by the pattern. ADD is positioned in the life cycle after requirements analysis and, as we have said, can begin when the architectural drivers are known with some confidence. Figure 2 depicts the main steps of this method. We define a specialization of the ADD method, using Model-Driven Engineering techniques, which systematizes and assists the Decision Making activity. To this end, we use the proposal of Rozanski et al. [14] 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 view-point is defined in terms of a set of models and activities. A precise definition in terms of the OMG's four-layer meta-modeling approach is part of the ongoing work. We follow the recommendation in [3] that clearly states which kinds of elements can be part of different types of views. When defining a model, we select the view-type that best suits the model intention. We use UML notation for depicting models.

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 consists 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 built by applying patterns and tactics that address the identified concerns. The third level provides a Platform Specific perspective of the architecture (PSA). It provides a technological solution to the abstract architecture to second level. This division not only organizes architectural views, but also separates platform independent from platform specific architectural decisions. We illustrate in Figure 3 Model Driven Design Method we propose.

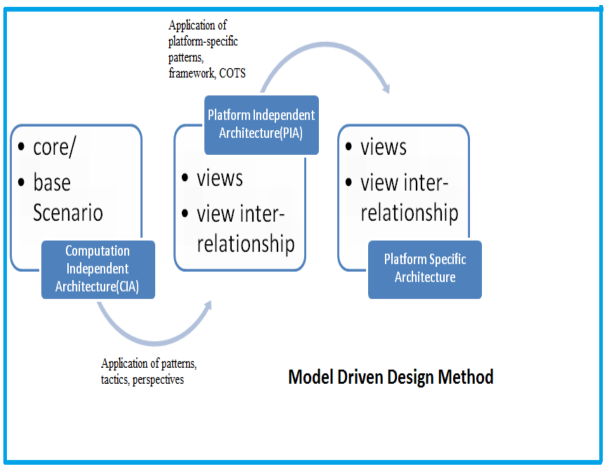

Application of platform-specific patterns, framework, COTS
Platform Independent Architecture(PIA)
• core/
• base
Scenario
Computation Independent Architecture(CIA)
• views
• view inter-relationship
• views
• view inter-relationship
Platform Specific Architecture
Application of patterns, tactics, perspectives
Model Driven Design Method

Figure 3. MDDM for architecture description.

In order to assist 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 mega-model that follows the structure depicted in Figure 3.

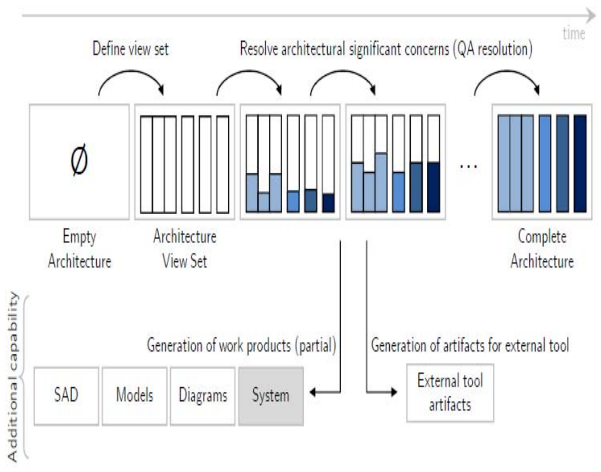


Figure 4. Decision Making activity.

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. 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.

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 [11]. 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.

A. 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 have following scenario for case study:

GS1: *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.

VS1: 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.

VS2: 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.

VS3: Mandatory User Authentication. The POS system accepts requests from users only after they are authenticated. Also, some requests can only be placed by privileged users.

B. *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:

GS1: 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.

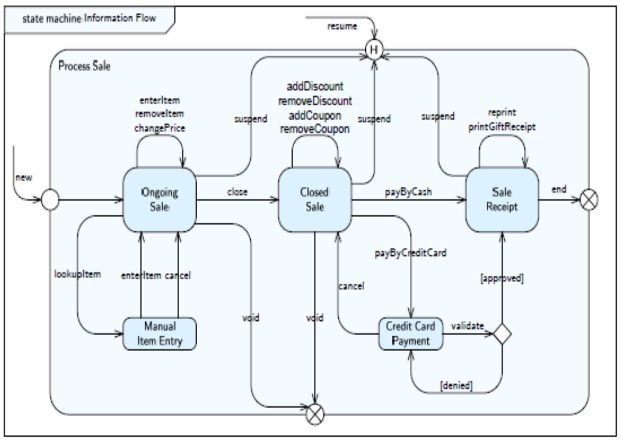


Figure 5. Information Flow Model.

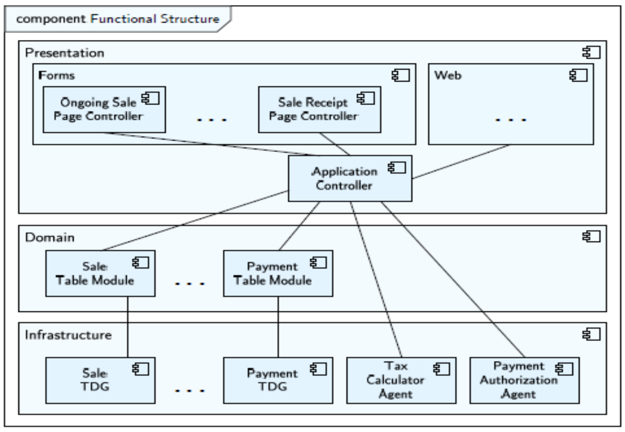


Figure 6. Functional Structure Model.

VS1 & VS2: 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 VS2, 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 behaviour 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. VS2 renders the need for in-site workstations (Register node) and a web server dedicated node for attending different thin-clients. Figure 7 illustrates the Runtime Platform Model. Different input and output devices for the Register node are decided following the Process Sale (GS1) functional scenario.

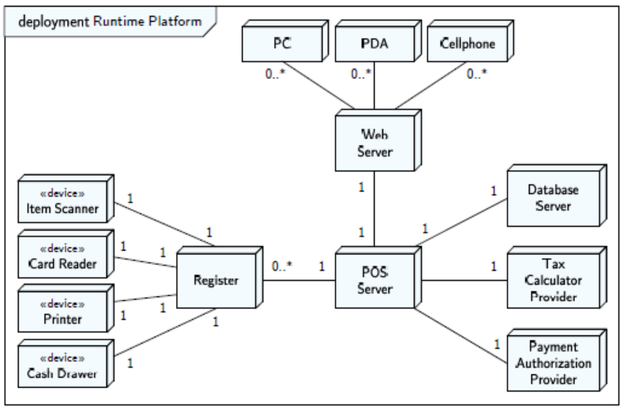


Figure 7. Runtime Platform Model.

VS3: Mandatory User Authentication. To address VS3, 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.

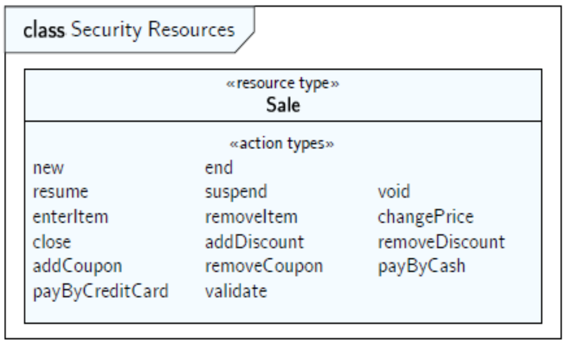


Figure 8. Security Resource Model.

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.

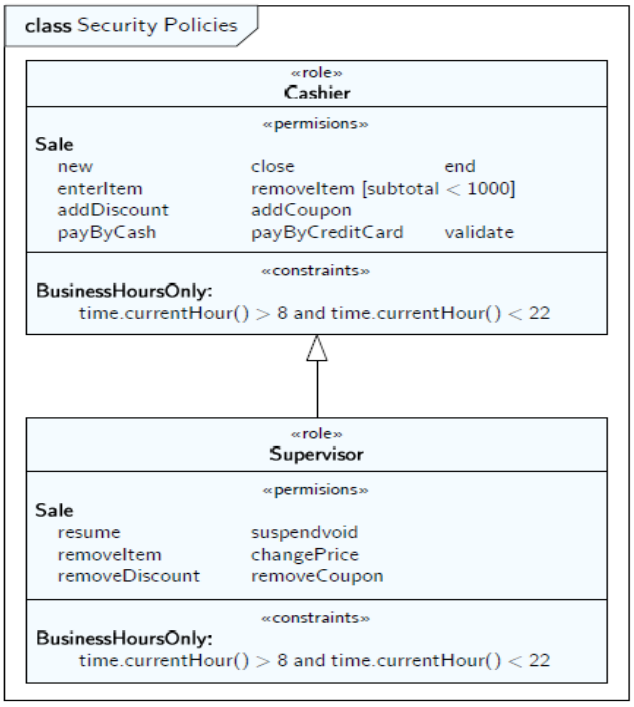


Figure 9. Security Policy Model

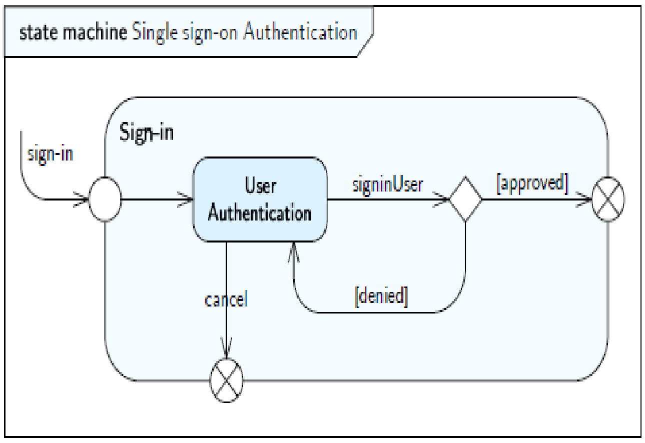


Figure 10. Single Sign-On aspect in Information Flow Model.

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.

ARCHITECTURE RATIONALE

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.

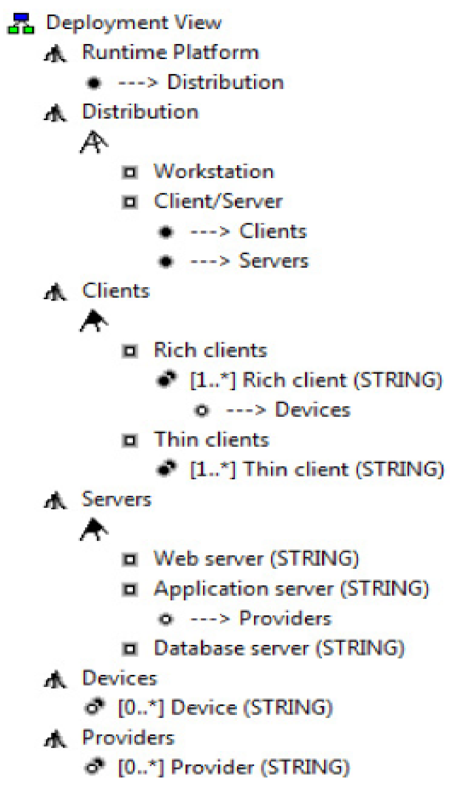
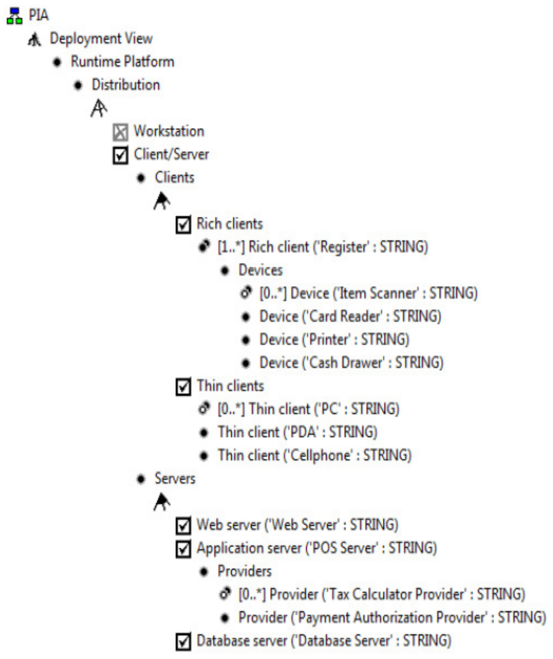


Figure 11. Deployment Decisions.

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.



```
PIA
  Deployment View
    • Runtime Platform
      • Distribution
          ☒ Workstation
          ☑ Client/Server
            • Clients
                ☑ Rich clients
                  [1..*] Rich client ('Register' : STRING)
                    • Devices
                      [0..*] Device ('Item Scanner' : STRING)
                      • Device ('Card Reader' : STRING)
                      • Device ('Printer' : STRING)
                      • Device ('Cash Drawer' : STRING)
                ☑ Thin clients
                  [0..*] Thin client ('PC' : STRING)
                  • Thin client ('PDA' : STRING)
                  • Thin client ('Cellphone' : STRING)
            • Servers
                ☑ Web server ('Web Server' : STRING)
                ☑ Application server ('POS Server' : STRING)
                  • Providers
                    [0..*] Provider ('Tax Calculator Provider' : STRING)
                    • Provider ('Payment Authorization Provider' : STRING)
                ☑ Database server ('Database Server' : STRING)
```

Figure 12. Deployment Rationale

CONCLUSIONS

Our method 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 sub transformations to this set, a large number of architectures can be designed using the method. 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 favours modularization and reuse, but also organizes and systematizes the architect's task. 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.

REFERENCES

[1] IEEE Std 1471-2000, IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, 2000.

[2] J. Bosch. Software Architecture: The Next Step. In EWSA'2004, pages 194–199, 2004.

[3] 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.

[4] J. C. Due˜nas and R. Capilla. The Decision View of Software Architecture. In EWSA'2005, pages 222–230, 2005.

[5] D. Falessi, G. Cantone, and P. Kruchten. Do Architecture Design Methods Meet Architects' Needs? In WICSA'2007, page 5, 2007.

[6] 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.

[7] 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.

THODETI SRIKANTH received his Master of Computer Applications degree from Kakatiya University, Andhra Pradesh, INDIA in 2004. He is pursuing **Ph.D.** (Computer Science) from Dravidian University, Andhra Pradesh, INDIA. He has more than 6 papers published in various reputed National / International Journals and Conferences.

DACHEPALLY RAVI KUMAR received his Master of Science degree from Osmania University, Andhra Pradesh, INDIA in 2008. He is pursuing **Ph.D.** (Computer Science) from Dravidian University, Andhra Pradesh, INDIA.

MAHI NAVEEN KUMAR received his Master of Technology degree from IASE Deemed University, Rajastan, INDIA in 2005. He is pursuing **Ph.D.** (Computer Science) from Dravidian University, Andhra Pradesh, INDIA