Dynamic Scenario Transformation in Software System Design

G. Manjula, Saleem Malik S, Deepa, Kannani and Jayprakash M

Abstract—The ability of reconfiguring software architectures in order to adapt them to new requirements or a changing environment has been of growing interest. We propose a uniform algebraic approach that improves on previous formal work in the area due to the following characteristics. First, components are written in a high-level program design language with the usual notion of state. Second, the approach deals with typical problems such as guaranteeing that new components are introduced in the correct state (possibly transferred from the old components they replace) and that the resulting architecture conforms to certain structural constraints. Third, reconfigurations and computations are explicitly related by keeping them separate. This is because the approach provides semantics to a given architecture through the algebraic construction of an equivalent program, whose computations can be mirrored at the architectural level.

Keywords—Dynamic Scenario, Software, Design.

I. INTRODUCTION

I. Literature Survey on Software Architecture

The goal of software architecture is to capture the persistent parts of the program and to derive the transient versions using architecture refinement. A number of architecture description languages are under development. Common too many of those architecture description languages is the concept of components and connections between them. The work at Northeastern on software architecture is distinguished by using adaptive programming (AP) to capture architecture. This has the advantage that architectural refinements can be automated in many cases. An adaptive program tries to capture the persistent parts of an object-oriented program by using traversals and context objects. Traversals are specified succinctly taking advantage of the structure of object-oriented programs. Work on AP is collected in the adaptive programming book and recent developments are described in Collaborative Behavior Modification.

II. APPLICATION ARCHITECTURE:

Software Applications come in all shapes and sizes. Based on their design and architecture enterprise applications can be classified into various categories such as:

- Distributed Applications
- Web Applications
- Web Services
- Smart Client Applications

This reviews some of the architecture Design patterns for Enterprise Applications built using the .NET Platform. Simply put Application architecture is:

- Set of significant decisions about how a software system is organized
- Selection of the elements that the system comprises of
- Interfaces of the elements
- Behavior of these elements
- Interaction of these elements within the system and with other systems

Problem Analysis

Model-View-Controller is a fundamental design pattern for the separation of user interface logic from business logic. Unfortunately, the popularity of the pattern has resulted in a number of faulty descriptions. In particular, the term "controller" has been used to mean different things in different contexts. Fortunately, the advent of Web applications has helped resolve some of the ambiguity because the separation between the view and the controller is so apparent.

The Model-View-Controller (MVC) pattern separates the modeling of the domain, the presentation, and the actions based on user input into three separate classes. Model The model manages the behavior and data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller). View The view manages the display of information. Controller The
controller interprets the mouse and keyboard inputs from the user, informing the model and/or the view to change as appropriate.

**Existing System Variations**

In Adaptive Programming Model-View-Controller (MVC) describes two variations of MVC: a passive model and an active model. The passive model is employed when one controller manipulates the model exclusively. The controller modifies the model and then informs the view that the model has changed and should be refreshed. The scenario is completely independent of the view and the controller, which means that there is no means for the model to report changes in its state. The HTTP protocol is an example of this. There is no simple way in the browser to get asynchronous updates from the server.

II. TESTING CONSIDERATIONS

Testability is greatly enhanced when you employ employing Model-View-Controller. Testing components becomes difficult when they are highly interdependent, especially with user interface components. These types of components often require a complex setup just to test a simple function. Worse, when an error occurs, it is hard to isolate the problem to a specific component. This is the reason why separation of concerns is such an important architectural driver. MVC separates the concern of storing, displaying, and updating data into three components that can be tested individually.

Apart from the problems posed by interdependencies, user interface frameworks are inherently difficult to test. Testing user interfaces either requires tedious (and error-prone) manual testing or testing scripts that simulate user actions. These scripts tend to be time-consuming to develop and brittle. MVC does not eliminate the need for user interface testing, but separating the model from the presentation logic allows the model to be tested independent of the presentation.

III. KEY FUNCTIONAL AND NONFUNCTIONAL CHARACTERISTICS

In our experience, effective business-process design and execution infrastructure should provision:

- Composition of business processes and integration of back-end systems.
- Smart-client interaction with the services’ infrastructure, composition, and front-end/back-end integration.
- Conversational capabilities of processes.
- Shortcut of the analyst-developer interaction with process-driven development.
- Reduction of the impact of change requests.
- Online/offline support at the process level; the ability to relocate business processes out of the center.
- Large-scale deployment, configuration, and management.
- Separation of rules from process code for maximum flexibility.

i. **Patterns.** We gathered patterns that are related to business-process development that illustrate a collection of architectural and workflow patterns:

- Basic workflow patterns—Control-flow, resource, data, and exception-handling–related patterns.
- Advanced workflow patterns—Wizard-based tools that drive contract-first development, inclusion of transparent processes or intermediaries, context-based process selection, enabling state full interactions, and externalization of business rules. Some of these patterns are not formalized elsewhere.
- Workflow-hosting patterns—A flexible hosting platform exposing workflows as Web services, locating and selecting business processes from a repository based on context, scalable process
execution environment, and pluggable workflow runtimes.

- Business-activity monitoring—Monitoring the current state of a business activity.

These patterns force you to build service compositions with the flexibility to dynamically change the activities through an appropriate service implementation based on business rules.

 Technologies Realizing Patterns.

System design offers two technology solutions to address business-process development and hosting: MDA and Workflow (WF). MDA offers a platform and tools to connect with services inside and outside of organizations, including a large number of adapters to legacy systems. WF is the programming model, engine, and tools for quickly building workflow-enabled applications. In addition, Microsoft consulting developed a wizard-driven toolkit and runtime, Distributed Connectivity Service (DCS), to provide guidance on developing and hosting business processes that are based on the aforementioned patterns. DCS provides service-management capabilities such as service location transparency, dynamic clients, context-driven business-process selection, scalable hosting of workflows and a security token service for exposing business processes as services. Depending on the enterprise scenario and hosting requirements, you could select MDA, DCS, or a combination.

IV. DYNAMIC SCENARIO TRANSFORMATION FOR MODEL TRANSFORMATION

A core concept of MDA is the transformation of models, where models are understood in the MDA sense of residing under some meta-model, itself an instance of the MOF. Such transformations are envisaged as occurring in many different scenarios within the software system design. The aim is to be able to define transformations easily and unambiguously; however, for this a language is required that can capture the concepts of arbitrary meta-models (provided they are MOF instances).

A case for graph transformation as a core technology upon which to base such a model transformation language. Briefly recapitulate the concepts of graph transformation; then we discuss the requirements for its application as a model transformation language. Illustrate these ideas by showing have recently given a graph transformation-based semantics to an experimental object-oriented language.

V. OBJECTIVES AND SCOPE

One feature of interesting applications of AP is that the weaving algorithm is sophisticated enough so that from weave (A1,A2, A3) it is not possible to uniquely determine a part even if all other parts are given. On the other hand, the weaving algorithm cannot be too sophisticated since it must be fast. For example, in simplest form of AP with class graphs and traversal specifications, it is usually impossible to uniquely determine the traversal specifications from the object-oriented program, the class graph and the context objects.

The typing of adaptive programs is a challenge. Call a program (A1, A2,*) type-correct if there exists a part A3 such that weave (A1, A2, A3) is type-correct. Jens Palsberg is currently studying the typing of adaptive programs: Given an adaptive program consisting of traversal specifications and context objects, is there a class graph which creates a type-correct object-oriented program.

The typing of adaptive programs is influenced by the generality of the weaving algorithm. If the weaving algorithm needs to make many restrictions regarding what can be woven together, the typing question becomes more difficult. It is important that the weaving algorithm is general purpose. See the work on compiling adaptive programs Efficient Implementation of AP and Automata Theoretic Compilation. The work on meta-object-protocols is also concerned, to some degree, with persistence versus transience and with software architecture.

VI. DESCRIPTION OF THE RESEARCH WORK:

Research Problem Definition

Mining software engineering secure software architectures using Model-driven architectures and Agile modeling for Requirements engineering, this is validated for Web Services case study and its related challenges. Our Strategies include text mining for source code and Design Patterns Mining and Graph Mining for architecture mining. Our Research Motivation is Integrating Security and Software Engineering using mining strategies. This approach involves: Evaluating the different Software Engineering Paradigms with respect to their appropriateness to integrate security; Developing new techniques, methods, processes that consider security as part of the software development life cycle; Tool Support/define a Suitable Exemplar; Transfer of security knowledge / transit research results to mainstream system development. Key research questions addressed are: Security Analysis and Design issue:

1. “How well the system authenticates the users and protects the application and data elements access control?”
2. “How Model-driven architectures and Agile Modeling are useful for security architecture?

How to develop business process conforming to the well-known workflow patterns; how to adapt quickly to process-change requirements?

How to ensure the same service contracts can be implemented in multiple fashions?

How to move processes seamless among the machines that need to host them, to handle disconnected scenarios and faults?

How to write client applications (across multiple channels) to compose existing UI components and services quickly, in different ways?
How to allow a process to maintain its state across multiple calls coming from one or more caller?

How to intercept service calls transparently, modifying their behavior via state full interceptors able to correlate multiple operations—for example, to apply new marketing-driven, customer state-specific behaviors?

How to monitor the current state of a process or business activity?

VII. CONCLUSIONS

More work is needed to explore techniques which allow us to better express the persistent properties of program families. The ideas behind AP should prove useful to formulate software architectures and patterns at a higher level of abstraction. At the programming language level, it would be useful to integrate the AP concepts of traversals and contexts into the programming language C, C++, Java, etc.

VIII. ADDITIONAL INFORMATION

The challenges for building the component-based software architecture is how to estimate the assembly of reusable software components and make the properties forecast to the associated architecture. To address these issues, this paper discerns the assembly patterns of components and proposes using graph theory to depict the component-based architecture at first. Then, the component assembly graph has been defined as the abstract representation of the component assembly in the architecture, which is a key factor in determining the suitable architectural structure. Based on such graph theory, some metrics have been defined to measure the components assembly. At the end, how to optimize the architecture from component level, which also shows the graph theory, should be paid attention to for the software engineering in future.

IX. LIMITATIONS

Using Evolution Patterns to Evolve Software Architectures - Presentation Transcript

1. Using Evaluation Patterns to Evolve Software Architectures Joint work with Dalila Tamzalit, Université

2. Context: Software Architectures Software development process – Requirements– models of software development like prototyping, formal transformations; extreme programming had been explored for secure programming practices. The present and future industry needs related to security architectures had been studied. Text mining and design pattern graph mining are applied for secure Source code and security architectures respectively. A case study on Web Services Security Architectures had been carried out to justify approach.

Architecture– Design– implementation Architectural descriptions– Capture strategic decisions and Ramonale at a high-level of abstractor– Provide a basis for detailed design– Are essential for expressing and constraining large-scale and criminal system


REFERENCES

[1] Karl J. Lieberherr College of Computer Science, Northeastern University Cullinane Hall, Boston, MA 02115


table of contents archive Volume 44 Issue 2, August 2002 Publisher: Elsevier North-Holland, Inc. Amsterdam, The Netherlands The Netherlands ISSN: 0167-6423 doi:10.1016/S0167-6423(02)00036-9
