Evaluating Software Architectures Against Requirements-level Scenarios

Mamadou H. Diallo, Leila Naslavsky, Hadar Ziv,
Thomas A. Alspaugh, and Debra J. Richardson
Department of Informatics
Donald Bren School of Information and Computer Sciences
University of California, Irvine
{mamadoud,lnaslavs,ziv,alspaugh,djr}@ics.uci.edu

ABSTRACT

Scenarios have been used to express requirements and system behavior throughout software development. Scenarios are used with different representation and semantics across software phases, and these can be related. This paper argues for exploring scenarios as one means for mapping requirements to architecture as well as evaluating architectures against requirements-level scenarios. Additionally, our approach facilitates consistency-checking between requirements and architectures. In our approach, software requirements take the form of ontology-based scenarios, while architectures are described using both structural and behavioral models. Mapping from requirements to architectures is modeled explicitly, then the mapping-model is used to evaluate architectures against original requirements-level scenarios.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Requirements/Specifications; D.2.1 [**Software Engineering**]: Software Architectures

Keywords

Scenarios, ontology, requirements, evaluation

1. INTRODUCTION

Software architecture, its components, and their behavior are specified as a refinement of users' original requirements. This refinement can be described as a mapping from requirements to architecture. When users' requirements are expressed in natural language sentences, the mapping cannot be precisely defined, often resulting in inconsistencies between requirements and architecture [10].

Scenarios have been used as an alternative (and sometimes complementary) way to express requirements and system behavior throughout the phases of software development. Scenarios are used with different representation and semantics across software phases, and these can be related. Generally, they can describe a system at different levels of abstraction [1], can be linked to software architecture, and be used for testing [5, 7]. At the requirements level, scenarios can be expressed in many forms, including prose. We use the ScenarioML language [1]. It provides a scenario syntax with a clear structure for events, and user-defined ontologies defining terms and type definitions, including event types.

This paper describes a three-step approach to architecture evaluation that aids software architecture designers. First, user requirements are specified using a structured scenario language; second, requirements scenarios are mapped into architectures using an architecture description language; and third, the resulting architectures are evaluated against original requirements scenarios.

An architectural description comprises structural and behavioral specifications. In this paper, we assume the structural description is written using xADL [6], an XML-based Architectural Description Language (ADL). An extension of xADL for behavioral description using statechart was proposed in [14]. Each component in the architecture description has their responsibilities precisely defined and the components' services provided through their interfaces.

We propose a schema that maps terms used in requirements scenarios (described using ScenarioML) to architectural components (described with xADL). This schema is used to evaluate the consistency of the architecture with the requirements. Our approach's main contribution is leveraging ScenarioML to establish the relation from requirements to architecture. Because ScenarioML supports an ontology of events, we can map events in the requirements to the components in the architecture responsible for them.

This paper is organized as follows. Section 2 summarizes the ScenarioML features supporting our approach. We introduce our approach to architecture evaluation in Section 3, and apply it to a textbook example in Section 4. Section 5 describes related work, and Section 6 concludes the paper.

2. SCENARIOML

ScenarioML [1] is an XML-based language for expressing scenarios by providing structure to support the aspects of textual scenarios that people treat and interpret consistently. It makes use of the division of scenarios into events; natural language *simple events* whose meaning is understood by humans; *compound events* consisting of subevents in a temporal pattern; *event schemas* for alternation, iteration, and the like; *episodes* that reuse an entire scenario as a single event of another; *ontology* consisting of a collection of term and type definitions; and a number of other features. It is designed both to support being read and written by all stakeholders (using software tools), and to accommodate machine processing. Here we discuss only the parts of ScenarioML most significant for our method.

An *ontology* consists of a collection of term and type definitions that can be interrelated. Term definitions allow the definition and explanation of unfamiliar terms and concepts in the scenarios and can be referenced anywhere in the document. Type definitions permit the grouping of related concepts under one named type. ScenarioML distinguishes two ways of grouping related concepts, namely *instanceType* for relating general concepts and *eventType* for relating events.

An *instanceType* is a defined type. The text contents of the element give a description not of the type, but of any individual instance of the type. Specific instances of the type may be listed as sub-elements. The type may be parameterized, in which case the parameters may be referenced in the description, and corresponding arguments must be given for the instances. The type may be defined to be a subtype of another instance type by the super element. If so, then any instance of this type is also an instance of the supertype.

An *instance* is an instance of a type. The instance's (human-readable) definition is given in the contents of the element. The type is the containing instanceType. At least one of name or text must be present. An instance contained in a parameterized instanceType must have an argument naming each of the type's parameters. An instance contained in an instanceType with relatedTypes must have relatedTo identifying the corresponding instances it is related to.

An *eventType* is a defined *typedEvent* type. An *typedEvent* is used as a way of abstracting one or more events, without the cognitive and practical overhead of creating, naming, and managing a new scenario. The event type's (human-readable) definition is given by the Event in the contents of the element. The event type may be parameterized, in which case the parameters may be referenced in the definition. The event type may be defined to be a subtype of another event type by the super element. If so, then any event of this type is also an event of the supertype. The event type can be instantiated to give an instance.

3. METHOD

In this section, we describe our method for evaluating software architectures based on requirements specification. Our method assumes the existence of the requirements scenarios and the architecture of the system. The method is based on ScenarioML for expressing the ontology-based scenarios and uses xADL for describing the architecture. We have selected xADL because it supports structural [6] as well as behavioral [14] description of architectures. Central to the method is the ScenarioML ontology that models the actions performed by the different actors in the scenarios using *eventType* and associated elements.

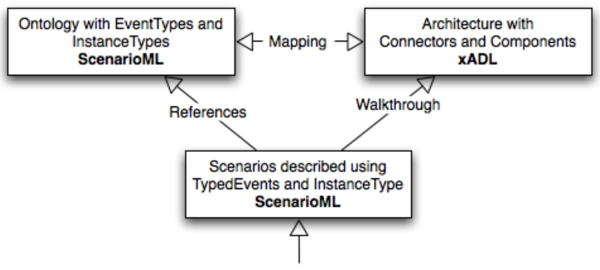
Ontology with EventTypes and InstanceTypes
ScenarioML
Mapping
Architecture with Connectors and Components
xADL
References
Walkthrough
Scenarios described using TypedEvents and InstanceType
ScenarioML

Figure 1: Overview of the Method

3.1 Approach

Our method takes requirements scenarios described in ScenarioML and evaluates them against the architecture of the system described with components and connectors. The method is based on explicit mappings between the events declared as types in the ontology of the scenarios and the components of the architecture. If the architecture includes the decomposition of the components into modules and interfaces, then the task of mapping becomes even simpler. The method comprises three main steps: (1) description of the important scenarios of the system in ScenarioML, (2) mapping the ontology *eventTypes* to the architectural components, and (3) walkthroughs of the scenarios in the architecture. Figure 1 shows an overview of the approach.

3.1.1 Scenarios Description in ScenarioML

ScenarioML as described earlier allows the expression of scenarios with clarity and consistency through the use of its ontology. Terms, concepts, and actions used throughout the scenarios can be defined in the ontology. The ontology plays the role of a placeholder where the scenarios can draw information from. The description of the scenarios in ScenarioML is accomplished through the following three steps:

1. Identify the different actors of the scenarios and the actions they perform. Then generalize the actions as much as possible to reduce the number of event types. The fewer the event types, the simpler is to perform the rest of the method.
2. Define the type of the events based on the identified actions in the first step using the *eventType*. This includes sub-typing and parameterizing the events as necessary. For example, a group of events can be grouped under a super-event, or a general event can includes parameters to allow specialization.
3. Write the scenarios using these event types. The element used for this purpose is the *typedEvent* element, which references a named *eventType*.

Scenarios are evaluated at the designer's will. Our approach do not propose a rule that ranks scenarios by importance and evaluates the most important. In general, the importance of a scenario is defined according to the importance of their event types (devised by the designer).

3.1.2 Mapping Ontology Elements to Architectural Components

The assumption here is that the software architecture is described using an architectural description language such as xADL. The architecture description needs to include components, connections, and constraints on the communication between the components. The role of each component must be specified unambiguously. If the architecture decomposes the components into modules, the responsibilities of each module needs to be identified precisely.

The mapping is performed between the event types of the ontology and the components in the structural description of the architecture. The mapping needs to be performed based on the meaning of the events and the responsibilities of the components. For example, in an ATM system, the event "The user inserts the ATM card into the ATM machine" is matched to the component that represents the user interface; the event "The system store the customer's information into the database." is matched to the component that represents the storage of the system. A table can be used to capture the mapping, with row headings representing the events and column headings the components.

The mapping is many-to-many. One event type from the requirements scenario describes a high level action that can be decomposed into component's low level actions. Therefore, to execute an event type from the requirements scenario, multiple component's low level actions may be executed. In addition, each component supports many low level actions, where each action can result from the decomposition of different event types from the requirements scenario.

3.1.3 Architecture Evaluation against Scenarios

The task of evaluating an architecture against a set of scenarios consists of going through the sequence of the events in the scenarios, using the established map to match events to components, while simulating the behavior of the matched components. The resulting architecture behavior is then evaluated for inconsistencies with the scenario.

The architecture can be inconsistent with the requirements in a number of ways. The inconsistency can take the form of a missing link between two components that is required by the scenarios. If two successive events match two components, where the first event in the sequence matches the first component and the second event matches the second component, then the two components may need to be able to communicate; if so, and there is no path joining the components, the architecture is inconsistent with the requirements. For the architecture to satisfy the requirements in this case, the second component needs to provide an interface to the first component to allow it to request the necessary services.

Another possible inconsistency is when the communication between two components violates the constraints imposed by the requirements. For example in a distributed system, a requirement can be "Clients need to communicate through a central server". This constraint can be violated if the architecture provides a direct link between two clients. Our approach however, currently only deals with inconsistencies based on component interactions.

3.2 Discussion

One of the main difficulties of using scenarios to evaluate architecture is finding the necessary information in the architecture to allow the execution of the scenarios on the behavioral description of the architecture. The difficulty becomes apparent when trying to match the events in the scenarios and the components in the architecture. The main issue is that the events are described at a very low level, while the components are described at a very high level. This issue is addressed in ScenarioML by the use of an ontology to express scenarios at different level of abstraction. The ontology allows the generalization and specialization of the actions performed by the actors in the scenarios. Considering for example an ATM system. The error message for invalid PIN, invalid card, and insufficient funds can be generalized under one error message action. One *eventType* can be used to generalize these three actions and a *typedEvent* can then be used to define each error message with different arguments.

Another difficulty in scenario-based architecture evaluation methods is how to choose the important scenarios to use in the evaluation process. The number of possible scenarios can be very large for even small systems, which makes it impractical to test every scenario. ScenarioML facilitates this task through the grouping of actions made possible by the ontology.

4. EXAMPLE

In this section, we illustrate the use of our method by applying it to an example application. We selected the PIMS (Personal Investment Management System), used in Jalote's book [11] and presented in details in the book's website, as an extended case study. PIMS is intended to help users keep accounts of their invested money in institutions such as banks and stock market. It includes all the development artifacts from the requirements documents to the Java source code. The requirements are presented in the form of use cases and a list of non-functional requirements. The architecture not only includes the main architecture diagram, but also the different modules and their interfaces comprising in each component.

4.1 PIMS ScenarioML Scenarios

The PIMS requirements document comprises 22 use cases. Each use case contains a main scenario and some alternative scenarios. The two principle actors of the system are "User" and "System". Based on the various scenarios, the actions performed by each actor were identified and described using the ScenarioML ontology. This description included generalizing and parameterizing the actions for simplicity and clarity. Figure 2 shows the actions performed by the actor "User" expressed using the ontology element *eventType.* In addition to defining the events for each actor, the general concepts of the system are also captured using the elements *term* and *instanceType* of the ontology.

Based on the *eventTypes* defined in the ontology, the various scenarios are described. The scenarios "Create Portfolio" in Figure 3 and "Login" in Figure 4 illustrate how the events can be defined using *typedEvents*, which refer to the *eventTypes* in the ontology. The scenario "Create Portfolio" describes the steps required to create a new portfolio and the "Login" lists the steps to be performed to log into

EventType **UserActions**

EventType **startApplication**
Super #UserActions
Simple Event The user starts the system.

EventType **initiateAction**
Parameter **action**
Action
Super #UserActions
Simple Event The user initiates the action action.

EventType **enterData**
Parameter **data**
Data
Super #UserActions
Simple Event The user enters the data.

EventType **editData**
Parameter **data**
Data
Super #UserActions
Simple Event The user edits the data.

Figure 2: EventTypes performed by the User

3. Create Porfolio

Sequence
1. TypedEvent **[cpf1]** : #initiateAction (•*action*='Create Portfolio')
2. TypedEvent **[cpf2]** : #requestData (•*data*='portfolio name')
3. TypedEvent **[cpf3]** : #enterData (•*data*='portfolio name')
4. Alternatives
   A. TypedEvent **[cpf4]**
      Comment Portfolio is created.
   : #createEntity (•*action*='create empty portfolio')
   B. Sequence
      Comment Portfolio with the same name exists
      1. TypedEvent **[cpf2]** : #requestData (•*data*='portfolio name')
      2. TypedEvent **[cpf3]** : #enterData (•*data*='portfolio name')
      3. TypedEvent **[cpf4]**
         Comment Portfolio is created.
      : #createEntity (•*action*='create empty portfolio')

Figure 3: Create Portfolio scenario

the system. In addition to providing a mean for mapping requirements and architecture, the ontology helps to keep the scenarios clear and consistent.

4.2 Mapping PIMS Ontology Elements to PIMS Architecture Components

The PIMS architecture is designed using the Access Layer Model. The architecture, described in xADL, comprises the components, connectors, and interfaces that can be visualized graphically. Each component in this architecture has a well defined role, which facilitated the mapping event types-components. The table in Figure 5 shows the mapping between some elements of the ontology and some components of the architecture. Each event type is mapped at least to one component and each component is mapped at least by one event type.

4.3 PIMS Scenarios Walkthrough

Since the PIMS architecture was carefully designed to be part of a book, it is consistent with all the scenarios describing the system functional requirements. In order to illustrate how our method discovers inconsistencies between

2. Login

Sequence
1. TypedEvent **[l1]** : #startApplication
2. TypedEvent **[l2]** : #requestData (•*data*='login and password')
3. TypedEvent **[l3]** : #enterData (•*data1*='login and password')
4. TypedEvent **[l4]** : #validateAction (•*action*='authentication')
5. Alternatives
   A. TypedEvent **[l4]**
      Comment Authentication succeeds.
   : #displayEntity (•*action*='main screen')
   B. Sequence
      Comment Authentication fails
      1. **[l4a1]** System prompts the user that he typed the wrong password.
      2. **[l4a2]** System allows user to re-enter the password. Give the user 3 changes.

Figure 4: Login scenario

eventTypes	Components
startApplication	Master Controller
initiateAction	Master Controller
enterData	Master Controller
requestData	Master Controller
executeAction	Authentication, Data Access Authentication, Data Repository
reportError	Master Controller

Figure 5: Mapping between Ontology Elements and Architecture Components

requirements and architecture, we introduced an error in the PIMS architecture by removing the link between the "Data Access Authentication" and "Data Repository" components. By introducing this error, our expectation was that the walkthrough of the "CreatePortfolio" scenario would succeed while the "Login" scenario would fail.

We performed the walkthroughs of the two scenarios manually. The "CreatePortfolio" main scenario contains four simple events in a chain and matches four components in the architecture. As expected, the walkthrough was successful because the sequence of the events in the scenario matches an appropriate sequence of the components.

The "Login" main scenario is also composed of four simple events in a chain and matches four components in the architecture. However, the sequence of the events in the scenarios does not go through the architecture due to the missing link between the "Data Access Authentication" and "Data Repository" components. The first two events of the scenario happen in the "Master Controller" component. The third event includes three components "Authentication" in the business logic layer, "Data Access Authentication" in the data access layer, and "Data Repository" in the data repository layer. Since the link between "Get/Set Authentication" and "Data Repository" is missing, the authentication data cannot be retrieved. Therefore, the architecture is inconsistent in regard to this scenario.

5. RELATED WORK

Software architecture specifies the high-level structure, behavior, and characteristics of a program that satisfies software product requirements. Evaluating an architecture against requirements during architecture design is important because after the product is built, it will be too late to fix architectural defects easily. However, evaluating an architecture is challenging because it is not possible to guarantee that the architecture meets its requirements [9]. So, evaluating an architecture gives only an estimate of the likelihood that it satisfies its requirements.

Scenarios have been proposed as a mean for analyzing and evaluating architectures by many researchers. Barber and Holt proposed using a scenario space for evaluating architectures [3]. The scenario space is a directed graph that represents possible threads of execution composed of services in the software architecture, which provides a high level view of the architectural execution. This visualization helps evaluate whether executing the architecture will support the anticipated scenarios for the application domain. Kazman *et al.* also used scenarios to analyze architectures with the focus on achieving quality attributes in their method, Scenario-based Architecture Analysis Method (SAAM) [13]. A number of other scenario-based architecture analysis methods are geared also toward evaluating architectures against the desired quality attributes described using scenarios. These include Software Architecture Level Modifiability Analysis (ALMA) [4], Performance Assessment of Software Architecture (PASA) [8], Architecture Level Usability Assessment (SALUTA) [8], and Architecture Trade-off Analysis Method (ATAM) [2]. None of these methods use an ontology to improve the clarity and efficiency of scenarios, and to provide an effective mapping between requirements and architecture to facilitate architecture evaluation.

Kaiya and Saeki developed a method for analyzing requirements based on ontology [12]. In this method, an ontology is used as a semantic domain for detecting defects in requirements such as inconsistencies and incompleteness. An ontology is not used to express the requirements as is the case in ScenarioML.

6. CONCLUSION AND FUTURE WORK

In this paper, we proposed a method for evaluating software architectures against requirements-level scenarios. The method evaluates architectures described using xADL [6] against requirements scenarios described in ScenarioML [1]. The method is a work in progress; however, the example included in this paper has shown that it can be used effectively to evaluate the likelihood that an architecture will satisfy its requirements.

This method differs from other scenario-based architectural evaluation methods in the sense that it uses an ontology for the underlying connection between the requirements and the architecture. The ontology provides, among other benefits, clarity and consistency in the scenarios and a base for efficient mapping between requirements elements and architecture components. We believe these will allow the development of effective automatic tool support.

Currently, the method focuses on two specific technologies, ScenarioML and xADL. As the method gains maturity, we intend to generalize the technique to include other architectural description languages. The method is also limited to functional requirements. However, for the method to be more useful, it needs to incorporate the evaluation of the architecture against non-functional requirements as well. Furthermore, for the method to be practical for designers, it needs to be provided with automatic tool support. In the future, we are planning to investigate the possibility of supporting the method with automatic tool support such as simulation tools.

7. ACKNOWLEDGMENTS

The authors would like to thank the ROSATEA group at the Donald Bren School of Information and Computer Science of the University of California, Irvine for their valuable suggestions and insights.

8. REFERENCES

[1] T. A. Alspaugh. Relationships between scenarios. Technical Report UCI-ISR-06-7, Institute for Software Research, University of California, Irvine, May 2006.

[2] M. R. Barbacci, S. J. Carriere, P. H. Feiler, R. Kazman, M. H. Klein, H. F. Lipson, T. A. Longstaff, and C. B. Weinstock. Steps in an architecture tradeoff analysis method: Quality attribute models and analysis. Technical Report CMU/SEI-97-TR-029, Software Eng. Inst., 1998.

[3] K. S. Barber and J. Holt. Software architecture correctness. *IEEE Software*, 18(6):64–65, 2001.

[4] P. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet. Architecture-level modifiability analysis (ALMA). *J. Syst. Softw.*, 69(1-2):129–147, 2004.

[5] A. B. Bertolino. A practical approach to UML-based derivation of integration tests. In *4th International Software Quality Week Europe and International Internet Quality Week Europe (QWE 2000)*, 2000.

[6] E. M. Dashofy, A. V. der Hoek, and R. N. Taylor. A highly-extensible, XML-based architecture description language. In *Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, page 103, 2001.

[7] J. Dick. Rich traceability. In *International Workshop on Traceability in Emerging Forms of Software Engineering*, Edinburgh, UK, 2002.

[8] E. Folmer, J. v. Gurp, and J. Bosch. Scenario-based assessment of software architecture usability. In *ICSE Workshop on SE-HCI*, pages 61–68, 2003.

[9] C. Fox. *Introduction to Software Engineering Design.* Addison-Wesley, 2007.

[10] P. Grünbacher, A. Egyed, and N. Medvidovic. Reconciling software requirements and architectures with intermediate models. *Software and System Modeling*, 3(3):235–253, 2004.

[11] P. Jalote. *An Integrated Approach to Software Engineering.* Springer-Narosa Publishing House, 2006.

[12] H. Kaiya and M. Saeki. Ontology Based Requirements Analysis: Lightweight Semantic Processing Approach. In *5th Int. Conf. on Quality Software (QSIC)*, pages 223–230, 2005.

[13] R. Kazman, G. D. Abowd, L. J. Bass, and P. C. Clements. Scenario-based analysis of software architecture. *IEEE Software*, 13(6):47–55, 1996.

[14] L. Naslavsky, L. Xu, M. Dias, H. Ziv, and D. J. Richardson. Extending xadl with statechart behavioral specification. In *Third Workshop on Architecting Dependable Systems (WADS)*, pages 22–26, May 2004.