Toward Architecture Evaluation Through Ontology-based Requirements-level Scenarios

Mamadou H. Diallo, Leila Naslavsky,
Thomas A. Alspaugh, Hadar Ziv, and Debra J. Richardson

Department of Informatics
Donald Bren School of Information and Computer Sciences
University of California, Irvine
{mamadoud,lnaslavs,alspaugh,ziv,djr}@ics.uci.edu

Abstract. We describe an approach for evaluating whether a candidate architecture dependably satisfies stakeholder requirements expressed in requirements-level scenarios. We map scenarios to architectural elements through an ontology of requirements-level event classes and domain entities. The scenarios express both functional requirements and quality attributes of the system; for quality attributes, the scenarios either operationalize the quality or show how the quality can be verified. Our approach provides a connection between requirements a stakeholder can understand directly, and architectures developed to satisfy those requirements. The requirements-level ontology simplifies the mapping, acts as the focus for maintaining the mapping as both scenarios and architecture evolve, and provides a foundation for evaluating scenarios and architecture individually and jointly. In this paper, we focus on the mapping through event classes and demonstrate our approach with two examples.

1 Introduction

Designing architectures that are consistent with their requirements is crucial in the development of large software systems. Software architecture evaluation methods have been proposed as a way of determining the fitness of an architecture with respect to its functional requirements as well as its quality attributes such as availability, reliability, performance, maintainability, and security [4, 7, 15, 18, 23, 31]. These quality attributes are in general expressed by stakeholders in natural language sentences, which are difficult to use in the evaluation methods. Scenarios have been used as an alternative (and sometimes complementary) way to express requirements and system behavior throughout the phases of software development. Scenarios have been used by the evaluation methods to relate requirements and architectures [3]. However, in most methods this relationship is not conserved for later use. For evaluation methods to be more useful and effective, the relationship between requirements and architectures needs to be maintained to support the evolving nature of these two processes.

Scenarios are used with different representations and semantics across software phases. They can describe a system at different levels of abstraction [1], can

be linked to software architecture, and be used for testing [9, 14]. At the requirements level, scenarios can be expressed in many forms, including prose. We use the ScenarioML language [1, 2]. It provides a scenario syntax with a well-defined structure for events, and user-defined ontologies defining domain concepts.

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

An architectural description comprises structural and behavioral specifications. Our proposed approach is not dependent on a particular Architectural Description Language (ADL). It does require, however, that each component in the architecture description have precisely defined responsibilities and services, which are provided through their interfaces. In this paper, we use structural descriptions written in xADL [12], an XML-based ADL. In addition, we adopt an extension of xADL for behavioral description that uses statecharts, proposed in [25].

We propose an approach that maps domain events, classes, and individuals used in requirements scenarios (described using ScenarioML) to architectural components (described using xADL). This approach supports evaluation of consistency between architecture and requirements. Our approach's main contribution is leveraging ScenarioML to establish the relation from requirements to architecture more effectively and evolvably. This is possible because ScenarioML supports an ontology of domain events and entities, which enables a straightforward and compact mapping from events in the requirements to the components responsible for those events in the architecture.

We use a domain ontology as the basis for the mapping between requirements and architectures because it facilitates the mapping. In this work, an *ontology* is a collection of domain class, individual, and event type definitions that are typically interrelated. ScenarioML supports and encourages reuse of event types as templates for specific events in scenarios, and unambiguous links to domain classes and individuals wherever these are referred to. The ontology not only improves the clarity of the scenarios [2], but also effectively reduces the complexity of links between the requirements and architecture elements. Without the ontology, each appearance of a scenario element is linked individually to all relevant architecture elements; with the ontology, the appearances are linked to its definition in the ontology, and only that definition is linked to the architecture elements. The more extensive the reuse of the ontology definitions in the scenarios, the greater is the reduction in complexity. ScenarioML supports reuse of event types that appear as equivalent events in the same or several scenarios; specialization and generalization of events through their event types; explicit relationships among a parameterized event type's instances with different argu-

ments; and domain classes and individuals referred to in events. In the initial work presented here, we focus on reuse of equivalent events.

To illustrate how our approach supports evaluation of software architectures against requirements-level scenarios, we apply it to two examples. The first example shows how our approach identifies inconsistencies between functional requirements and architecture. The second example shows that our approach can be applied to distributed systems and can also evaluate the architecture against non-functional requirements (e.g. availability, and reliability).

The remainder of the paper is organized as follows. Section 2 briefly discusses the portion of ScenarioML used by our approach. We present our approach to architecture evaluation in Section 3, and illustrate its application to two example systems in Section 4. Section 5 discusses some of our findings. We place the research in the context of related work in Section 6, and summarize in Section 7. Section 8 outlines our future work.

2 ScenarioML

ScenarioML is a language for expressing scenarios that provides structures to represent the aspects of textual scenarios that people treat and interpret consistently [1, 2]. 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; *ontologies* defining domain concepts; and a number of other features. It is designed 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 approach.

A ScenarioML *ontology* consists of a collection of domain class, individual, and event type definitions. The definitions typically refer to each other and are interrelated. A domain class (an *instanceType*) defines a class of entities in the domain that are in some sense equivalent. A domain individual (an *instance*) in the ontology defines a specific entity of a class whose existence is assumed or guaranteed; ScenarioML also provides structures for referring to individuals that are newly created or identified during the course of a scenario. An event type (*eventType* in ScenarioML) acts as a template for reusing the same event in several scenarios or several times in the same scenario. A domain class may be specified to be subsumed by another in a subclass/superclass relationship, as can an event type. Both domain classes and event types may be parameterized, in which case their instances are as well and must be given an argument for each parameter.

3 Approach

In this section, we describe our approach for evaluating a software architecture against requirements-level scenarios. The requirements-level scenarios need to

describe not only functional requirements, but also non-functional requirements as discussed below. With these scenarios one can evaluate whether the architecture meets the functional requirements. They also enable one to assess how well a selected architecture style can support the dependability qualities of the system such as availability, reliability, maintainability, and safety. The software architecture needs to be expressed in an ADL, with precisely defined responsibilities and services for each component.

The requirements-level scenarios are modeled with ScenarioML. The ScenarioML ontology models the actions performed by different actors in the scenarios using *eventType* and associated elements such as *super* and *parameter*. The scenarios are expressed by instantiating the *eventType*. Indeed, ScenarioML is the foundation of our approach.

We use xADL for describing the architecture in the examples presented in Section 4. However, any other ADL with similar features could be used. xADL is an XML-based architectural description language that is highly extensible. It supports structural [12] description of architectures and also behavioral [25] description of architectures using statecharts. Additionally, it has tool support for runtime and design time modeling, architecture configuration management, and model-based system instantiation.

3.1 Overview of approach

Our approach takes requirements-level scenarios described in ScenarioML and evaluates them against the architecture of the system described with components and connectors. The scenarios describe the functional and non-functional requirements that are important to the stakeholders. The approach is based on explicit mappings between *eventType*s in the ontology and components in the architecture. The mapping is created by examining in conjunction the meaning of the event in the scenarios and the roles played by different components in the architecture.

The approach comprises four main steps: (1) description of the important scenarios of the system in ScenarioML, (2) description of the architecture using an architectural description language, (3) mapping the ontology event types to the architectural components, and (4) walkthroughs of the scenarios in the architecture. Figure 1 shows an overview of the approach.

3.2 Scenarios Description in ScenarioML

Description of scenarios with ScenarioML can be accomplished through the following three steps:

1. Identify actors of the scenarios and actions they perform. Then, generalize the actions where possible to reduce the eventual number of event types. The fewer the event types, the simpler it is in the approach's later steps.
2. Define the event types based on the identified actions in the first step using the *eventType*. This includes sub-typing and parameterizing events where

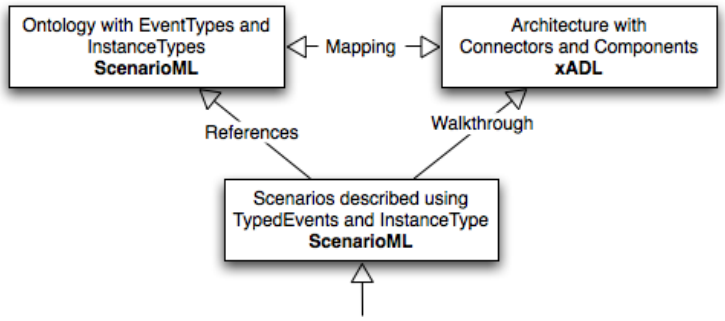

Fig. 1. Overview of the approach

appropriate. For example, a group of related events can be grouped under a super-event, or a general event can include parameters to allow specialization to particular contexts.

3. Write scenarios using the previously defined event types. The element used for this purpose is the *typedEvent* element, which references and reuses a defined *eventType*.

The requirements scenarios for a system are often quite numerous. Our approach does not propose a method for ranking scenarios by importance, so that limited evaluation time can be focused on the most important ones. In general, we expect that the importance of a scenario is determined by the system's designers.

3.3 Architecture Description

We assume the architecture is described with an ADL that supports both structural and behavioral descriptions. The structural description provides the infrastructure for mapping, while the behavioral description allows dynamic checking of the architecture against scenarios. The architecture description should include components, connections, and constraints on the communication between the components. The role of each component must be specified unambiguously to facilitate the mapping of event types and components. If the architecture decomposes the components into subcomponents, the responsibilities of each subcomponent should be identified precisely. It this case, the mapping can be done at the subcomponent-level, which can give more detailed information about the fitness of the architecture in regard to requirements.

3.4 Mapping Ontology Elements to Architectural Components

The mapping is performed between event types in the ontology and components in the architecture's structural description. It is based on the meaning of the

events of the scenarios and the responsibilities of the components. For example, in the PIMS (Personal Investment Management System) system [21] described later in Section 4, the event "The user enters the portfolio's name" is matched to the component "Master Controller", which manages the user interface; the event "The system authenticates the user" is matched to the component "Authentication", which is responsible for the authentication task. A table can be used to capture the mapping, with row headings representing the events and column headings the components (such as Table 1).

The mapping is many-to-many. One event type from the requirements-level scenario describes a high level action that can be decomposed into several of a component's low level actions. Therefore, to execute an event type from the requirements-level scenario, multiple 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-level scenario.

3.5 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 mapping 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. In this case, the structural description of the architecture should allow such communication, else the architecture may be inconsistent with the requirements. The architecture satisfies the requirements-level scenario if the second component provides an interface to the first component to allow it to request the necessary services, for example.

Another possible inconsistency occurs when the structural description of the architecture violates constraints imposed by the requirements. For instance, a requirement for a distributed system could be "Clients need to communicate through a central server." This constraint can be violated if the architecture allows two clients to communicate directly, bypassing the central server.

Some quality attributes can be more effectively described using *negative scenarios*. A negative scenario describes an undesirable behavior of a system. In this case, the inconsistency is identified by a successful execution of the negative scenarios. For instance, for security reasons a requirement for a distributed system could be "Users need to be authorized to access the network." A scenario could describe a user with inadequate authentication information accessing the system. The successful execution of such a scenario implies the system is not secure.

4 Two Applications

In this section, we illustrate our approach by applying it to two example applications. The first example is a single-process textbook system, which we used to show how our approach detects inconsistencies between functional requirements of a system and its architecture. The second example is a realistic distributed and decentralized system, which we used to demonstrate how our approach can be used to analyze the fitness of an architecture with regard to non-functional requirements (quality attributes). Both the functional and non-functional requirements of this system are described using scenarios.

We chose these applications rather than real-world industrial systems because they have relatively complete requirements and architectures. Our earlier study found that few if any publicly available industrial systems have both [13].

For this study, we selected xADL to describe the architecture of the systems. We used Archipelago, a user-friendly graphical editor for xADL included in ArchStudio 4 to describe the PIMS's architecture [12]. ArchStudio 4 is an open-source software and systems architecture development environment. It is an environment of integrated tools for modeling, visualizing, analyzing and implementing software and systems architectures, based on the Eclipse open development platform.

4.1 PIMS

The first example we selected for this study is PIMS (Personal Investment Management System), included in Jalote's book [21] and presented in detail on the book's website as an extended case study. PIMS is used by customers to keep track of their invested money in institutions such as banks and in the stock market. It includes all the development artifacts from the requirements documents to the Java source code.

The PIMS functional requirements are presented in the form of use cases. In total the system's requirements comprise 22 uses cases. Each use case contains a main scenario and some alternative scenarios. The system contains only few non-functional requirements, which pertain to performance, security, and fault tolerance. For the purpose of demonstrating our approach, we focus on two functional scenarios, "Create portfolio" and "Get the current prices of shares." The scenario "Create portfolio" describes the steps required to create a new portfolio and "Get the current prices of shares" lists the steps to be performed to get the current prices of shares from the Internet (Figure 2).

The PIMS use cases in this example are:

"Create portfolio" main scenario: (1) User initiates the "create portfolio" functionality. (2) System asks the user for the portfolio name. (3) User enters the portfolio name. (4) An empty portfolio is created.

"Create portfolio" alternate scenario: (4.a) Portfolio with the same name exists. (4.a.1) System asks the user for a different name. (4.a.2) User enters a different name. (4.a.3) Empty portfolio gets created.

"Get the current prices of shares" main scenario: (1) User initiates the "download current share prices" functionality. (2) The system downloads the current share prices from a particular website. (3) The system display the current share prices. (4) The system save the current share prices.

"Get the current prices of shares" alternate scenario: (2.a.1) The system is not able to download (due to network failure, site down, ...). (2.a.2) The system gets current value saved from before. (2.a.3) The system display current value saved from before; ask the user to change it.

The PIMS architecture (Figure 3) is designed using the Layered Architectural Style. The architecture not only includes the main architecture diagram, but also the different modules and their interfaces comprising in each component. It comprises a data access layer separating the business logic and data repository. Data retrieval and modification is done via this data access layer, while all the processing of data or implementation of the business logic done in the business logic layer. The fourth layer is the presentation layer ("Master controller") which is responsible for interacting with the user and invoking modules of the business logic layer.

PIMS ScenarioML Scenarios This first step of the approach comprises using the ScenarioML language to develop an ontology for PIMS, and describing the selected PIMS scenarios using the ontology elements. The two principal 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, and identifying equivalent events that can be defined once and shared. Figure 2 shows some 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 and included in the PIMS ontology.

Based on the *eventTypes* defined in the ontology, the selected scenarios are described. Figure 2 shows the description of the scenarios "Create portfolio" and "Get the current prices of shares" respectively. The important events in these scenarios are defined using *typedEvents*, which refer to the *eventTypes* in the ontology.

PIMS Architectural Description in xADL In this second step, we described the PIMS architecture using xADL. Figure 3 shows the structure description of the PIMS architecture. We used the Archipelago editor in the ArchStudio 4 environment to draw the diagram and define all the elements of the architecture.

Mapping PIMS Ontology Elements to PIMS Components In this third step, we created a mapping between PIMS ontology elements and the PIMS architecture components. The description of the PIMS architecture is presented in

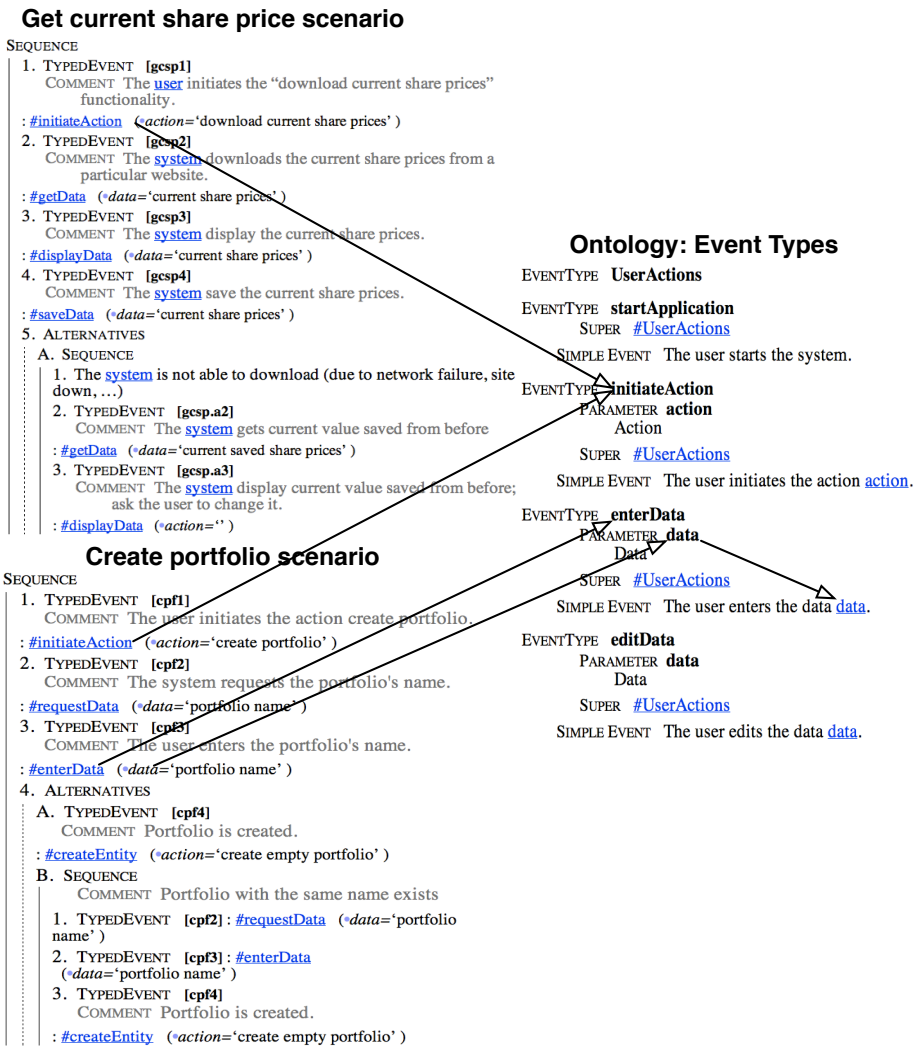

Fig. 2. PIMS scenarios and part of mapping to PIMS ontology

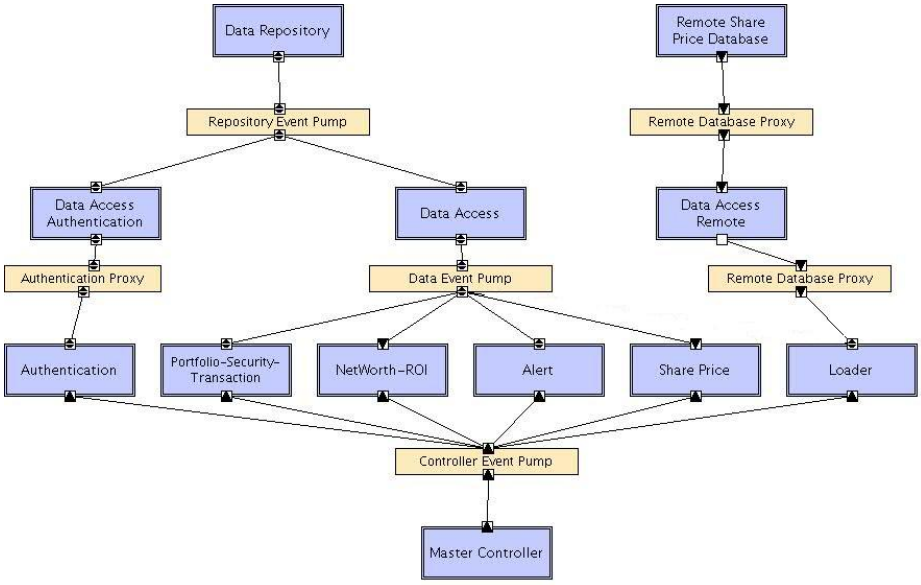

Fig. 3. The architecture of PIMS described in xADL

Figure 3. The architecture comprises the components with their interfaces and connectors, that can be visualized graphically. Each component in this architecture has a well defined role, which facilitated the mapping from event types to components. Table 1 shows the mapping between some elements of the ontology and some components of the architecture. Each ontology event type is mapped at least to one component and each component is mapped to by at least by one ontology event type.

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

Table 1. Mapping between ontology event types and architecture components

PIMS Scenarios Walkthrough In this final step, we performed a walkthrough of the scenarios in the architecture. Since the PIMS architecture was carefully

designed to be part of a book, we were not surprised to find it is consistent with all the scenarios describing the system functional requirements. In order to illustrate how our approach discovers inconsistencies between requirements and architecture, we artificially introduced an error in the PIMS architecture by excising the link between the "Data Access" and "Loader" components. In introducing this error, our expectation was that the walkthrough of the "Create portfolio" scenario would succeed while the "Get the current prices of shares" scenario would fail.

We performed the walkthroughs of the two scenarios manually. The "Create portfolio" 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.

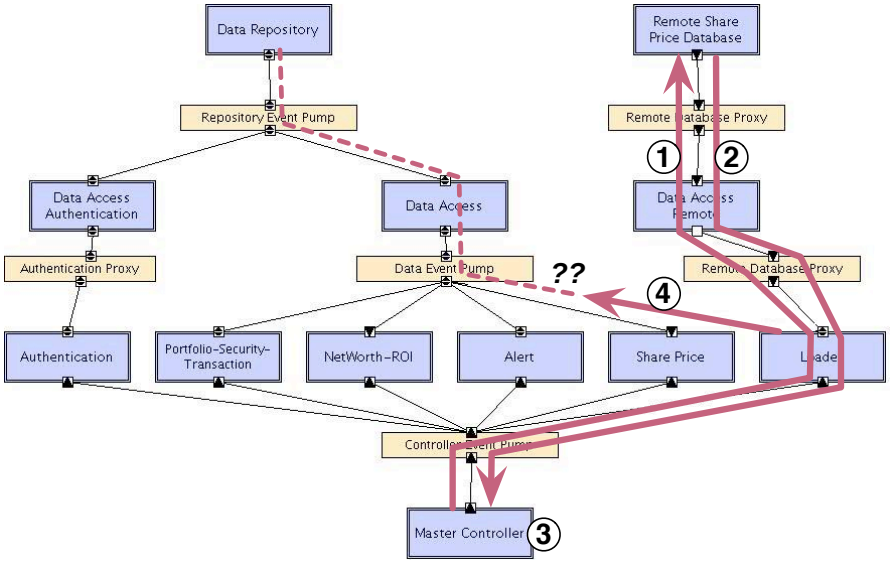

Fig. 4. Failed walkthrough of "Get the current prices of shares" scenario

The "Get the current prices of shares" 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 scenario does not succeed in the modified architecture due to the excised link between the "Data Access" and "Loader" components. Figure 4 illustrates the walkthrough of this scenario. The first event sends a request from the "Master Controller" component through intervening connectors and components to the "Remote Share Price Database". The second event transfers data back to the "Master Controller", and the third event displays it there. The fourth event would transfer specific data from the "Loader" through "Data Access" to the "Data Repository" to be saved. Since

the necessary first link along this path between the "Loader" and "Data Access" was excised, and other paths do not support transfer of this data, the current prices of shares cannot be sent to the "Data Repository" to be saved. Therefore, the modified architecture does not cover this scenario.

4.2 CRASH System

The second system we chose to illustrate our approach is CRASH (Crisis Response and Situation Handling), a decentralized and distributed system developed in our department for case studies [27]. CRASH models a collection of governmental and non-governmental organizations cooperating in response to emerging situations in order to make decisions. The system contains the following decision-making organizations: Police Department, Fire Department, Search and Rescue, Red Cross, St. Elsewhere Hospital, a Charitable Organization, and the Department of Public Works. Each CRASH peer is divided into three subsystem classes: Display, Information Gathering Sources, and Command and Control. The Display sub-system is responsible for visualizing the information currently known to the organization such as deployment of resources and other vital information. Information Gathering Source sub-systems provide feedback and information to the entity's Command and Control sub-system, for example by relaying reports from the public. These sub-systems are connected to the entity's Command and Control through internal ad hoc networks. Additionally, each entity's Command and Control center is also connected to the Command and Control centers of other organizations, perhaps, again, through ad hoc networks. An entity's Command and Control center is then responsible for aggregating data received from its information sources as well as information from other organizations. Ultimately, the Command and Control system is responsible for making decisions on behalf of the entity and conveying information and instructions to its affiliated resources. A high-level architecture of the system is illustrated with two peers in Figure 5.

For the CRASH system to be dependable, it needs to be available, reliable and secure. Since the system is intended to be used to manage crisis in critical times, its availability is crucial. The continuity of correct service of the system is necessary to successfully coordinating the activities of the different organizations during the execution of the operations. Secondly, the CRASH system needs to be reliable so that all the services it delivers during the operations are correct. Finally, the system also needs to be secure to ensure that malicious entities cannot join the network and perform malicious behavior.

In this section, we focus on two scenarios that illustrate several dependability issues of CRASH.

The first scenario in concerned about the availability of the system. In the CRASH system, it is important to know what are the available entities at any given time. The availability of a system can be compromised by hardware and software failures. In this example scenario, we focus on the software failure. The following scenario operationalizes the availability requirement by showing how the system handles the failure of a component.

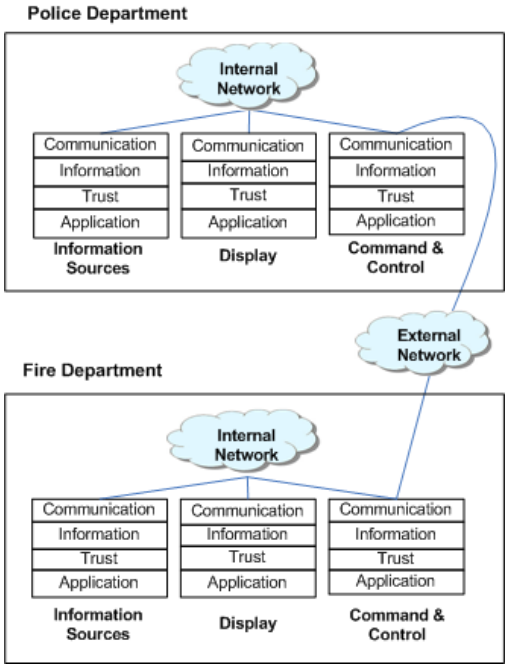

Fig. 5. CRASH High-Level Architecture

Entity Availability scenario: (1) The Police Department shuts down its Command and Control entity. (2) The Fire Department's Command and Control sends a request message to the Police Department's Command and Control. (3) The Network sends a failure message to the Fire Department. (4) The Fire Department receives the failure message.

The second scenario focusses on the reliability of the system. In order for the CRASH system to deliver reliable services, the communication between the entities in the system need to be effective. One aspect of this is the sequence in which messages are received; messages received out of order can create a mistaken understanding of what has occurred. The following scenario shows how the reliability requirement can be verified by testing whether the messages sent by a peer are received by other peers in the same sequence they are sent.

Message Sequence scenario: (1) Fire Department's Command and Control sends a request message to the Police Department's Command and Control. (2) Fire Department's Command and Control sends a second request message to the Police Department's Command and Control after 5 seconds. (3) The Police Department's Command and Control receives the first message. (4) The Police Department's Command and Control receives the second message.

The architecture style used to design the CRASH system is C2 [28]. A C2 architecture is composed of components and connectors that are organized into layers. Components in a layer are only aware of components in the layers above

and have no knowledge about components in layers below. Components communicate with each other using two types of asynchronous event-based messages, requests and notifications. Request messages travel up the architecture while notification messages move down the architecture.

CRASH ScenarioML Scenarios In this step, an ontology and scenarios for the non-functional requirements were developed for the CRASH system. The principle actors of the system are "User", "System", "Entity", and "Network". Based on the non-functional requirements 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 and for a minimal set of event types. The description in ScenarioML of the ""Entity Availability" scenario is presented in Figure 6, and that of the ""Message Sequence" is presented as part of Figure 8.

```
SEQUENCE
  1. TYPEDEVENT [ac1]
     COMMENT The Police Department shuts down its Command and Control entity.
  : #shutdownEntity (•entity='Police Department' , •component='Command and Control' )
  2. TYPEDEVENT [ac2]
     COMMENT The Fire Department's Command and Control (from the UI component)
          sends a request message to the Police Department's Command and Control.
  : #sendMessage (•sender='Fire Department' , •message='Request Message'
  , •receiver='Police Department' )
  3. TYPEDEVENT [ac3]
     COMMENT The Network (Multicast Manager component) sends a failure message
          to the Fire Department (UI Component).
  : #sendMessage (•sender='Network (Multicast Manager)' , •message='Failure Message'
  , •receiver='Fire Department (UI)' )
  4. TYPEDEVENT [ac4]
     COMMENT The Fire Department (UI Component) receives the failure message.
  : #receiveMessage (•sender='Network (Multicast Manager)' , •message='Failure Message'
  , •receiver='Fire Department (UI)' )
```

Fig. 6. "Entity Availability" scenario

CRASH Architectural Description in xADL In this step, we again used Archipelago to define the CRASH architecture. The full architecture is too large to show here; as an illustration, we show the internal architecture of the Police Department's Command and Control center in Figure 7.

Mapping CRASH Ontology Elements to CRASH Architecture Components Figure 8 gives a general overview of the relationships between ontology, scenarios, and architecture in our approach. It illustrates the mapping between the event types in the CRASH ontology and the components in the CRASH architecture. For example, the event type "sendMessage" is mapped to three

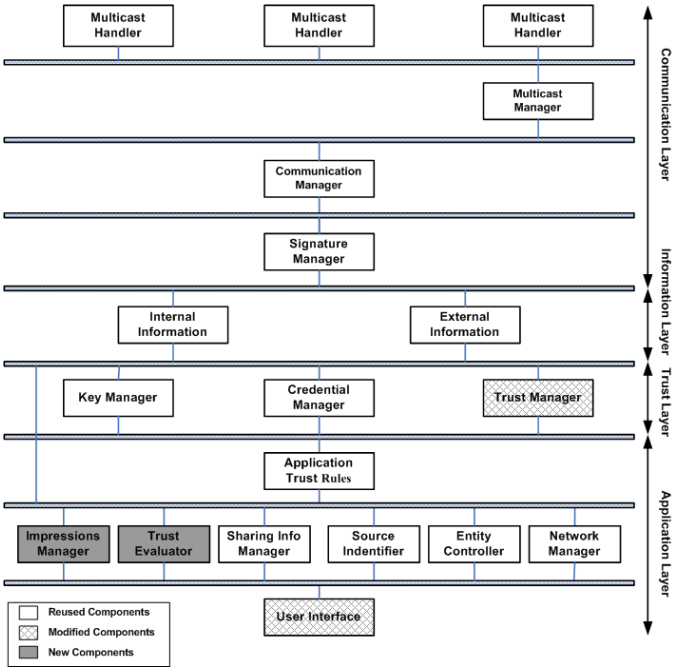

Fig. 7. Architecture of each CRASH Entity

components: "User Interface", "Sharing Info Manager", and "Communication Manager". It also shows how event types in the ontology are instantiated as typed events in the scenarios.

CRASH Scenarios Walkthrough The two selected scenarios, "Entity Availability" and "Message Sequence", are concerned with availability and reliability respectively. In general, static walkthroughs have limited effectiveness for evaluating satisfaction of quality attributes by an architecture. These two quality attributes can be determined effectively only at run-time. Since we have not implemented our tool for supporting the execution of the architecture with scenarios, we demonstrate the concept by describing what could have happened when the execution of the scenarios on the architecture is simulated.

The "Entity Availability" main scenario contains four simple events in a chain that match a number of components in the architecture. The result of the walkthrough of this scenario is as follow. If the architecture provides a mechanism for detecting the availability of the entities, than the User Interface component of the Fire Department's Command and Control, which initiated the first event in the scenario, will receive an error message alerting the unavailability of the Police Department's Command and Control. Otherwise, Fire Department's Command and Control will not receive any alert about the unavailability of Police Depart-

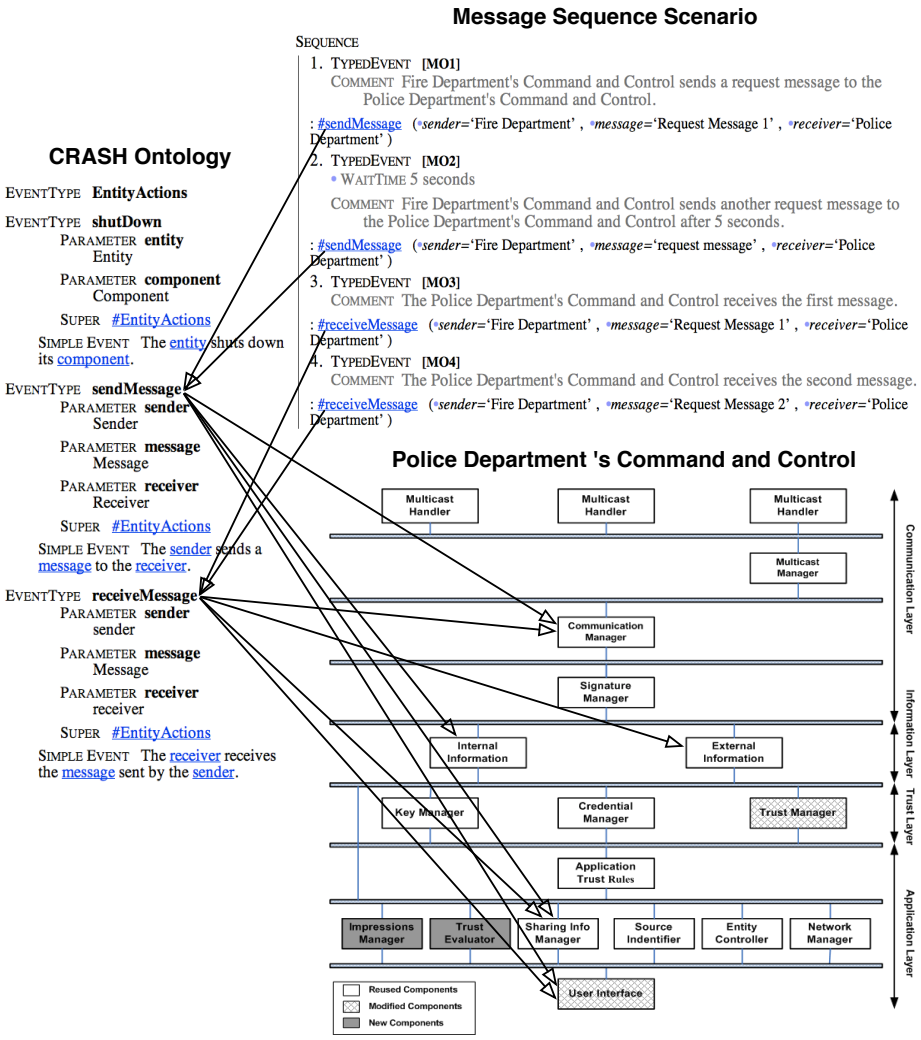

Fig. 8. CRASH ontology, scenario, and architecture mapping

ment. This scenario is effective in determining if the an entity is available at any given time.

The "Message Sequence" main scenario is also composed of four simple events in a chain that match a number of components in the architecture. The main concern in this scenario is the preservation of the messages order in the architecture. The result of the walkthrough is as follow. If first message sent by the Fire Department's Command and Control arrives first in the Police Department's Command and Control, then the order is preserved; otherwise the order not preserved.

5 Discussion

A challenge of using scenarios to evaluate architecture is finding the necessary behavioral information in the architecture in order to exercise the architecture convincingly with a requirements scenario. This difficulty is highlighted when one tries to match events in scenarios to appropriate components and services in architectures. One aspect of the challenge is that requirements events are typically described at a finer granularity than architectural components, with several successive events often mapping to a single component. A requirements-level ontology can support generalization and specialization of the actions performed by actors in the scenario events, giving more flexibility in the conceptual level of the events. For instance, in the CRASH system, the message for saving, updating, and deleting information, can be generalized under one more-abstract message action, especially if the architecture handles all three similarly. Then the more-abstract *eventType* for that action can be mapped more straightforwardly and dependably to the architecture. The method by which it is specialized for the three cases (by parameters, or by event subtypes, or a combination) then provides a specific direction from which to evaluate how the architecture supports each of the cases.

Another challenge for scenario-based architecture evaluation methods is selecting the most important scenarios to evaluate the architecture. The number of possible scenarios can be very large for even small systems, which makes it impractical to check all scenarios. ScenarioML supports this task with its ontology that allows reuse of a single event type in several scenarios, or several times in a single scenario, resulting in fewer distinct events that may need to be covered.

A third challenge for scenario-based architectural evaluation methods is the difficulty of determining and expressing the non-functional requirements. Ideally, quality requirements are written completely and unambiguously in a requirements document prior to architecture design. Too often, however, quality requirements are not written or poorly written. Kazman *et al.* argue that quality attribute requirements for both existing and planned systems are often missing, vague, or incomplete [24]. Even if the quality attributes exist, they are limited to simple statements. For our approach, these quality attributes need to be described using scenarios, as in our previous work [30]. The scenarios need

to describe specific examples (or counterexamples) of current and future uses of the system. Since these quality attributes need to be gathered for the most part from the stakeholders, designers need to work closely with stakeholders during requirements elicitation, specification, and analysis. Once identified, our approach facilitates their expression and management.

Traceability between requirements and architecture is a key condition for software maintainability with a reduced negative impact on software quality. Software evolves to meet users' new requirements, to correct defects, and to cope with changes to the environment in which it operates, among other reasons. Management of maintenance tasks demands some effort on the part of the developers, depending on the nature of the change and on available tool support. Tools that support requirements traceability (e.g. DOORS and Requisite Pro) manage source code and/or architecture versions [19], and supporting effective regression testing can alleviate the burden of software maintenance. When evolution of stakeholder requirements creates software maintenance tasks, traceability assists developers in locating other artifacts that also need modifications. Hence, traceability increases maintainability.

One benefit of our approach is the traceability links that are established between requirements and architecture, which ease maintenance involving these artifacts. Our approach explores ontology-based requirements-level scenarios to trace requirements to architecture. It explicitly maps event types in the ontology to components in the structural architectural description, and uses the ontology to simplify and minimize the mapping.

6 Related Work

Software architecture specifies the high-level structure, behavior, and characteristics of a system intended to satisfy software product requirements. Evaluating an architecture against requirements during architecture design is important because it is faster and cheaper to fix defects early. However, evaluating an architecture is also challenging because it is not possible to guarantee absolutely that the architecture meets its requirements [16]. So, evaluating an architecture can give only an estimate of the likelihood that it satisfies its requirements.

Scenarios have been proposed as a means for analyzing and evaluating architectures by many researchers. Barber and Holt proposed using a scenario space for evaluating architectures [5]. 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) [23]. A number of other scenario-based architecture analysis methods are also geared toward evaluating architectures against the desired quality attributes described using scenarios. These include Software Architecture Level Modifiability Analy-

sis (ALMA) [7], Performance Assessment of Software Architecture (PASA) [15], Architecture Level Usability Assessment (SALUTA) [15], ART-SCENE [31], and Architecture Trade-off Analysis Method (ATAM) [4]. 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. We build on their work in the ways we use scenarios to evaluate an architecture.

Grunbacher *et al.* proposed the CBSP (Component-Bus-System-Property) approach for bridging requirements and architecture [18]. CBSP uses intermediate models to systematically reconcile requirements and architecture. Unlike our approach, CBSP is not intended for the evaluation of architecture, but instead is used in the design process. It helps designers develop architectures based directly on the requirements. Another method that aids the design of software architectures is the integrated decision-making framework [20]. The framework aids in systematically determining architecture alternatives from negotiated requirements among stakeholders. It facilitates requirements elicitation, architecture alternatives exploration, and reaching agreement. However, it does keep any mapping between requirements and architectures.

Ontologies have been proposed as way of representing knowledge on the Semantic Web [8]. An ontology is a data model representing a set of concepts within a domain and the relationships between those concepts. On the Semantic Web, ontologies facilitate interoperability and allow autonomous agent interaction. The idea of ontologies for knowledge representation that facilitates the sharing of information between agents has been used in traditional software development, in particular, in requirements engineering. Kaiya and Saeki developed a method for analyzing requirements based on ontologies [22]. In this method, an ontology is used as a semantic domain for detecting defects in requirements such as inconsistencies and incompleteness. The ontology is not used to express part of the requirements, as is the case in ScenarioML. Breitman *et al.* use ontologies to formalized services specifications in multi-agent systems [10]. The role of the ontologies in this work is to enhance the communication protocol to allow software agents to exchange meaningful information. Ontologies capture the semantics of the operations and services provided by agents, allowing interoperability and information exchange in a multi-agent systems. Again, the ontologies are not used to express the requirements. Breitman and Leite considered an ontology of a web application as a sub-product of the requirements engineering activity [11]. From this viewpoint, they proposed a requirements engineering based process for the construction of ontologies. The process is based on language extended lexicon (LEL) and provides a way to implement ontologies using the application lexicon. Our research builds on this work.

7 Summary

In this paper, we proposed an approach for evaluating software architectures against requirements-level scenarios. The approach comprises four steps: (1)

user requirements specification in the form of scenarios using ScenarioML [1], (2) architectures design using an architectural description language, (3) mapping the requirements scenarios into the architecture components through an ontology, and (4) architectures evaluation against the original requirements scenarios. While ScenarioML constitutes the basis for specifying the requirements, any ADL that supports structural and behavioral specification of architectures can be used to specify the architecture. In this paper, we focused on xADL [12].

As a proof of concept, we applied our approach to two example applications, where we used xADL as the language for describing the architectures. We used the first example, PIMS, to illustrate how our approach checks the consistency of an architecture in regard to functional requirements. We used the second example to show that our approach can be used to check the fitness of an architecture in regard to non-functional requirements including dependability. Furthermore, the second example demonstrates the applicability of our approach to distributed and decentralized systems.

The next step in evaluating the effectiveness of our approach is to use our tool supporting the approach, currently under development. With the tool, we will be able to automatically check all the considered scenarios, which will lead to better results.

This approach differs from other scenario-based architectural evaluation methods in 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.

Another benefit of our approach is the traceability between requirements and architectures made possible by the mapping. By explicitly mapping event types in the ontology to components in the architectural description, requirements changes in the scenarios can be traced to the architecture and vice versa. As a consequence, requirements can evolve while the pre-established mapping assists developers in locating impacted components in the architecture.

8 Future Work

We are currently developing a tool called SOSAE (Scenario and Ontology-based Software Architecture Evaluation) to support this approach. SOSAE is an Eclipse plug-in tool that facilitates the mapping between the ontology elements of the requirements and components of the architecture. Furthermore, SOSAE provides the mechanism for automatically "executing" the scenarios on the architecture. For the description of the scenarios, we plan to integrate SOSAE with the Scenario Workbench, an Eclipse plug-in tool for editing and working with ScenarioML scenarios. In this way, the scenarios will be described in the Scenario Workbench and automatically loaded in SOSAE. For the description of the architecture, we will provide a means for integrating SOSAE with an appropriate architectural description language. The description of the architecture will

be also automatically loaded in SOSAE. The first version of the tool is focusing on xADL.

We plan to generalize SOSAE to work with a range of ADLs. Our choice for supporting this is the generic ADL Acme [17], a simple ADL that can be used as a common interchange format for architecture design tools and/or as a foundation for developing new architectural design and analysis tools. Acme is also attractive in this context because it incorporates both structure and behavior description mechanisms. Our approach will then make use of AcmeStudio [26], an Eclipse plug-in tool that facilitates editing and visualization of software architectural designs based on the Acme architectural description language (ADL).

In Section 4 we applied our approach to two example systems as an illustration and proof of concept. We are planning a more detailed and convincing evaluation of our approach. An analogous evaluation using real-world systems would also require subject systems with both detailed requirements and detailed architectures. However, our earlier study [13] found that such real-world systems are hard to find, and in practice may not exist. After our SOSAE tool is completed, we plan a study in which we observe the use of our approach by an industrial partner in developing a real system.

We are strongly interested in connecting stakeholders into the architectural phase of development. The present work describes using scenarios to evaluate an architecture; we also envision using scenarios to communicate stakeholder goals and needs forward to software architects, and also using stakeholder-level scenarios from the architects to communicate the results of architectural analysis and evolution back to the stakeholders. These in turn could be used to derive implied scenarios from the combined stakeholder and architectural scenarios, using the approach of Uchitel *et al.* [29], in order to identify possibly undesired implied scenarios.

We will also explore the capability of an ontology to improve the efficiency of the mapping between requirements and architecture. Currently, our approach uses the domain ontology to define equivalent event types that can be reused when expressing the scenarios. The mapping is performed through event types to the architectural components, which results in a simpler mapping if event types are reused several times or in several scenarios. We hypothesize that the mapping can be further simplified, facilitated, and made more evolvable through use of other ontology features supported in ScenarioML: specialization/generalization among events, relationships among instances of an event type with different arguments, and references from events to domain classes and individuals. For example, the events that map to a specific component can be determined by the domain entities that appear in those events, rather than the actions the events describes. In such cases, defining the mapping links in terms of finer-grained elements such as domain classes shows promise to provide mappings that can adapt under evolution more naturally and efficiently, and thus (among other benefits) help the requirements and architecture to evolve coherently together.

Our future work also includes using the full potential of the domain ontology to further reduce the complexity of requirements to architecture mapping. The

version of ScenarioML used here has it own domain ontology sublanguage. We are moving toward the use of the OWL web ontology language [6] in order to make use of existing OWL tools and reasoners.

9 Acknowledgments

The authors would like to thank the ROSATEA research group at the Donald Bren School of Information and Computer Science of the University of California, Irvine, and the anonymous reviewers of an earlier version of this paper, for their valued suggestions and insights.

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. T. A. Alspaugh, S. E. Sim, K. Winbladh, M. Diallo, H. Ziv, and D. J. Richardson. The importance of clarity in usable requirements specification formats. In *5th Intl. Wkp. on Comparative Evaluation in Requirements Engineering (CERE'07)*, 2007.
3. M. A. Babar and I. Gorton. Comparison of scenario-based software architecture evaluation methods. In *APSEC*, pages 600–607. IEEE Computer Society, 2004.
4. 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.
5. K. S. Barber and J. Holt. Software architecture correctness. *IEEE Software*, 18(6):64–65, 2001.
6. S. Bechhofer, F. v. Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL web ontology language reference. Technical report, W3C, 2004. `http://www.w3.org/TR/2004/REC-owl-ref-20040210/`.
7. P. Bengtsson, N. Lassing, J. Bosch, and H. v. Vliet. Architecture-level modifiability analysis (ALMA). *J. Syst. Softw.*, 69(1-2):129–147, 2004.
8. T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5), May 2001.
9. 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.
10. K. K. Breitman, A. H. Filho, E. H. Haeusler, and A. von Staa. Using ontologies to formalize services specifications in multi-agent systems. In *3rd Intl. Workshop on Formal Approaches to Agent-Based Systems (FAABS 2004)*, pages 92–110, 2004.
11. K. K. Breitman and J. C. S. d. P. Leite. Ontology as a requirements engineering product. In *11th IEEE Joint International Conference on Requirements Engineering (RE'03)*, pages 309–319, 2003.
12. E. M. Dashofy, A. v. d. 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.
13. M. H. Diallo, S. E. Sim, and T. A. Alspaugh. Case study, interrupted: The paucity of subject systems that span the requirements-architecture gap. In *First Workshop on Empirical Assessment of Software Engineering Languages and Technologies (WEASELTech'07)*, 2007.

14. J. Dick. Rich traceability. In *International Workshop on Traceability in Emerging Forms of Software Engineering*, Edinburgh, UK, 2002.
15. E. Folmer, J. v. Gurp, and J. Bosch. Scenario-based assessment of software architecture usability. In *ICSE Workshop on Bridging the Gaps Between Software Engineering and Human-Computer Interaction*, pages 61–68, 2003.
16. C. Fox. *Introduction to Software Engineering Design*. Addison-Wesley, 2007.
17. D. Garlan, R. T. Monroe, and D. Wile. Acme: An architecture description interchange language. In *CASCON'97*, pages 169–183, 1997.
18. P. Grünbacher, A. Egyed, and N. Medvidovic. Reconciling software requirements and architectures with intermediate models. *Software and System Modeling*, 3(3):235–253, 2004.
19. A. v. d. Hoek, M. Rakic, R. Roshandel, and N. Medvidovic. Taming architectural evolution. In *Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, 2001.
20. H. In, R. Kazman, and D. Olson. From requirements negotiation to software architectural decisions. In *1st Intl. Workshop on From Software Requirements to Architectures*, 2001.
21. P. Jalote. *An Integrated Approach to Software Engineering*. Springer, 2006.
22. 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.
23. 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.
24. R. Kazman, M. Klein, and P. Clements. ATAM: Method for architecture evaluation. Technical Report CMU/SEI-2000-TR-004, Soft. Eng. Institute, 2000.
25. 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.
26. B. Schmerl and D. Garlan. AcmeStudio: Supporting style-centered architecture development. In *26th Intl. Conf. on Softw. Eng. (ICSE'04)*, pages 704–705, 2004.
27. G. Suryanarayana, M. H. Diallo, J. R. Erenkrantz, and R. N. Taylor. Architectural support for trust models in decentralized applications. In *28th Intl. Conf. on Softw. Eng. (ICSE'06)*, pages 52–61, 2006.
28. R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., and J. E. Robbins. A component- and message-based architectural style for GUI software. In *17th Intl. Conf. on Softw. Eng. (ICSE'95)*, pages 295–304, 1995.
29. S. Uchitel, J. Kramer, and J. Magee. Detecting implied scenarios in message sequence chart specifications. In *Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, pages 74–82, Sept. 2001.
30. L. Xu, H. Ziv, T. A. Alspaugh, and D. J. Richardson. An architectural pattern for non-functional dependability requirements. *Journal of Systems and Software*, 79(10):1370–1378, Oct. 2006.
31. X. Zhu, N. Maiden, and P. Pavan. Scenarios: Bringing requirements and architectures together. In *2nd Intl. Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, 2003.