Using Scenarios to Support Traceability

Leila Naslavsky Thomas A. Alspaugh Debra J. Richardson Hadar Ziv
Donald Bren School of Information and Computer Sciences
University of California, Irvine
Irvine, CA 92697-3425 USA
{lnaslavs,djr,alspaugh,ziv}@ics.uci.edu

ABSTRACT
Software traceability is a recognized challenge in software development that can be ameliorated with requirements management tools. Traceability information can be used in a number of different software engineering activities such as software change impact analysis and testing One main challenge in the automation of software testing is mapping modeling concepts to code concepts. The level of granularity and the semantics supported by available requirements management tools do not, however, fully support such mapping, or more sophisticated requirement change impact analysis. 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. This paper argues for exploring scenarios as one means for tracing requirements to code, and using this information to leverage automation of activities that benefit from traceability such as change impact analysis and software testing.

Categories and Subject Descriptors
D.2.1 **[Requirements/Specifications]**: Tools, D.2.5 **[Testing and Debugging]**: Testing tools (e.g., data generators, coverage testing)

General Terms
Design, Management, Measurement, Verification.

Keywords
Scenarios, model-based testing, requirements, traceability.

1. INTRODUCTION
Software evolves as a result of modifications to users' original requirements, modifications to the environment in which it operates, and repairs that fix reported errors. Management of changes demands some effort on the part of the developers, depending on the nature of the change and on tool support. Tools that support requirements traceability, manage software versions, and support effective regression testing alleviate the burden of software maintenance. In particular, requirements traceability tools such as DOORS [19] and Requisite Pro [20] support management of traces from requirements to other artifacts. Indeed, these tools are widely adopted and accepted as satisfactory for manually establishing and — to some extent — supporting maintenance of traces [16].

Requirements managed by these tools are expressed in natural language sentences, each of which is given a unique identifier. This identification is used to relate requirements to other artifacts, so one can find the possibly affected artifacts associated with each change to a requirement. Nevertheless, relevant information that supports software evolution, such as what part of the code might need modification due to a requirement change, is not readily available. On the other hand, the level of granularity and the way requirements are expressed and structured can be chosen to aid developers in obtaining more precise information. Scenarios have been used as an alternative and complementary way to express requirements, as well as to express the behavior of the system throughout the software development life cycle. However, they are used with different representation and semantics across different software phases. Generally, they can describe a system at different levels of abstraction, can be linked to software architecture [17], and be used for testing ([9],[6]). Scenarios can be expressed in many forms, including prose. Approaches such as ScenarioML [2] support description of scenarios in a syntactically structured way (e.g. XML); this kind of structuring opens the door for modeling finer grained traceable entities. Finer grained tracing can leverage automation of activities benefiting from traceability, such as change impact analysis and software testing (which includes test-case generation, evaluation and regression testing). In this paper, we are only concerned with scenarios already considered as a requirement specification, and reused downstream in the software life cycle (post-traceability rather than pre-traceability). We believe that if appropriate support is provided to stakeholders (e.g., users, designers, testers) to refine the high-level requirements scenarios to low-level design scenarios or code scenarios, this information can be maintained and leverage aforementioned activities. It is worth noting that such scenarios describe functional requirements. This does not preclude traceability of non-functional requirements when they are also expressed as scenarios. However, traceability of non-functional requirements described through scenarios is part of a separate discussion, not pertinent to this paper.

This paper explains scenario semantics used in the different phases of the software life cycle (section 2); presents one way to relate scenarios across phases (section 3); describes one way to

TEFSE 2005, November 8, 2005, Long Beach California, USA.

relate scenarios to other artifacts (section 4); discusses related work (section 5) and concludes by explaining our future work.

2. SCENARIO USES

In software engineering, scenarios are used to express system requirements, analyses, component interactions [17], and test cases [6],[9]. To establish a relationship among them, we need to understand their purpose in each software phase, and their commonalities and differences across phases. A scenario is a sequence of events. What constitutes an event is variously described in the literature: something that happens, an action, an interaction, a step, or in some usages only an instantaneous change from one state to another. Our view is that any of these may constitute an event, but that an event is fundamentally something occurring over a time interval, not instantaneously. While a number of commonly-used formalisms assume non-overlapping events for simplicity (often expressed as time points or restricted to be state changes), in the real world closer examination shows that no event is truly indivisible. Rather, an event that appears to be instantaneous at one level of detail, abstraction, or concepts, is revealed at a finer-grained level to be composed of smaller sub-events or to be better described by concepts that distinguish its time span and evolution over that time. Thus, we relate scenarios through their *events* and *concepts* and the *level of abstraction* at which they are used, across the different kinds of scenarios and different development phases.

Requirements engineering is one of the areas that most heavily explores the uses of scenarios ([1],[3],[8],[18]). For our purposes, ***requirements scenarios*** are used to describe requirements after a preliminary discovery phase (as distinguished from those used during the process of discovering the requirements). A requirements scenario is closer to stakeholders' comprehension than scenarios used in later phases, and describes sequences of events that take place at the system-wide level of abstraction. These events occur in the real world (only) or are interactions between a user and the system-to-be. Information about the system's internal events is not pertinent to this kind of scenario. An example of this kind of scenario is seen in UML use cases, in which the individual traces (if present) are scenarios. Use cases are represented primarily in prose, which makes it harder to map them to lower and more detailed levels of abstraction. Since this mapping is needed from a traceability perspective, as well as from a testing perspective, these scenarios should be structured and expressed in a way that allows a lower level of granularity. The structuring would support linking and managing scenario-related pieces at the different levels of abstraction, and thus alleviate the mapping problem. For that purpose, we consider ScenarioML – a language for describing scenarios using XML – to be well suited. XML has been widely adopted by academia and industry, so using it to express scenarios has advantages such as availability of tools to analyze and work with it. Moreover, the scenario schema provides guidance on the information needed to be filled in each phase, and can be extended with some formalism to support testing-related needs.

Analysis scenarios add detail and some system internals perspective to requirements scenarios. Here, the top-level entities that appear in the internals of the system are identified, and the ontology of the concepts used in the requirements scenarios is more precisely defined - hierarchies of sub-concepts and instance-part hierarchies are elaborated. In addition, the events formerly described through prose and preliminary identified domain-specific concepts, can now be described in terms of more exact ontological concepts. The events in this phase should be interactions among system concepts. In case an object-oriented analysis approach such as UML is used, an example of this kind of scenario would be a UML sequence diagram, or a message sequence chart in which agents that exchange messages are at a level of abstraction similar to classes in an analysis class diagram. Those concepts are entities expected to appear in the system, but it is still not specified if they will become implementation components or objects. We believe ScenarioML should be used to describe analysis scenarios as well. We expect extensions to be needed in the schema to support a more structured way to express hierarchies of sub-concepts and instance-part hierarchies.

Architecture scenarios encompass scenarios that involve even more detail. In a manner analogous to architecture structure definitions, in this phase of scenario definition decisions are made on which concepts will become components of the systems architecture. So, the scenarios are at the architecture-level of abstraction, in which events are described as message exchanges among architectural components, or among components and connectors, depending on the architectural style chosen. In this case, the analogy is to an equivalent UML abstraction level that makes use of extensions to UML that support architecture descriptions [12]. An example of this kind of scenario is a UML sequence diagram, or a message sequence chart where the agents that exchange messages are architecture-level components and connectors, those being described by ADLs such as xADL [4]. The same descriptive structure as for earlier scenarios can be reused and provide further support for describing events in terms of interactions among components. In fact, if lower level scenarios are to satisfy the requirements scenario, concepts used in the latter will appear in the lower levels as well, and should thus be related to that level's specific concepts. Not all concepts will be projected across all scenario levels, and the exact relation among concepts is not fully defined, but it is very likely that hierarchies of sub-concepts and instance-part relations will predominate. Since structural architectures and architecture scenarios are both described in terms of components, we envision a relation among them to be easily established through this concept.

Design scenarios are scenarios described in more detail than architecture scenarios. The concepts used in this case are object-parts of the architectural components. The design scenarios are thus scenarios at the design abstraction level, where events are message exchanges among objects. A design scenario is an architecture-scenario that has been opened and elaborated to show the internals of the components.

Code scenarios describe the flow of events at a system implementation level. They are in fact runtime trace information corresponding to statements executed or method invocations, depending on the level of abstraction at which they appear. So requirements, analysis, architecture, or design scenarios, when executed, result in some code scenario. In this case, events are method invocations that describe object interactions. Design scenarios can be linked to code through the concepts in the ontology as well as through events refinements. Events from the design scenarios are message exchanges among objects, and message exchanges become, in turn, method calls. Objects from design scenarios also appear as implementation concepts at the code level, and thus both concepts are related.

The scenarios defined above for each of the software life cycle phases can in fact be used as test scenarios. ***Test scenarios*** appear in each phase of the development life cycle, and are used to test system implementations or other system artifacts. Requirements scenarios are the closest to the users, and are indeed used as guidance for black-box testing. Analysis, architecture, and design scenarios have internal information about the system, and thus can be used to simulate its behavior and guide white-box testing. Scenarios at each level of abstraction are used to test that the behavior of a lower level scenario conforms to the expected behavior described by the higher-level one. Since one higher-level scenario can lead to multiple lower level scenarios, there is a need to test their conformity. For instance, suppose there are two related scenarios, one an analysis scenario and the other an architecture scenario. An event (and related concepts) in the analysis scenario is related to several events (and related components) in the architecture scenario. Simulation and correlation of lower level events to higher-level events check for conformity across those two scenarios.

3. SCENARIOS RELATIONS

There are many ways to express tracing among scenarios. In this section, we describe one possible approach using ScenarioML [2], which is a scenario description language that can describe whole families of scenarios as scenario schemata, and also express temporal relations between event intervals using Allen's interval algebra. ScenarioML scenarios are described using XML — an extensible markup language which can be used to provide structure to scenarios. XML has been widely adopted by academia and industry and offers many advantages such as availability of tools to parse, analyze and visualize. Clearly, using the same language to describe all kinds of scenarios facilitates the establishment and maintenance of scenario relationships, which in turn aids in maintaining traceability among software artifacts that are related to scenarios. Due to space constraints, we do not discuss the ScenarioML grammar, semantics, or schema details here; rather we provide a brief example using ScenarioML to relate scenarios through their events and refer the reader to [2] for more detailed information about ScenarioML.

In ScenarioML, an event can be a simple event, a grouping of events, or an operator over events, all having attributes. We used the *level* attribute of an event to capture the level of abstraction of the scenario where the event appears. Depending on the complexity of the system, a scenario of one kind can still have many levels of abstractions, and thus, many levels of events and concepts. So, the levels of abstraction of an event should not be limited to the kind of scenario (and its level of abstraction) in which it is used. Since the ScenarioML language is still evolving, it is expected and intended to support more sophisticated solutions such as descriptions of events at multiple levels of abstractions and references among them — an alternative to using unique names to trace them. Nevertheless, we used ScenarioML "as is", and figures 1, 2 and 3 shows how events and terms can be traced through different scenarios.

The name is used as a unique identifier of events, allowing us to trace events refined from one phase to the other. The simple event *simpleEvent-select-new-channel* in Figure 1 is traced to the catenation (consecutive sequence of events) with same name in Figure 2. This catenation added detail to the simple event from the requirements level, so it was broken into two other events (*simpleEvent-new-channel-to-remote* and *simpleEvent-new-channel-to-tv*). Events *simpleEvent-get-remote-control* and *simpleEvent-channel-change*, and terms *the-tv* and *the-remote-control* did not change from requirements to analysis, so they are repeated in figure 2. Figure 3 in turn shows that for the architecture scenario, the event *simpleEvent-channel-change* that appeared originally in the requirement scenario in Figure 1, was further detailed into a catenation of six events to show that the channel (*the-new-channel*) is in fact transmitted to a tuner (*the-tv-tuner*), and revealing architectural aspects of the system: *the-tv-tuner* and *the-tv-display* are components of *the-tv*. This information is explicitly shown in Figure 3, where it is said that the term *the-tv* is composed by *the-tv-tuner* and *the-tv-display*. This is an example of how to describe related concepts the scenario where they appear.

This example illustrates how traces between scenarios can be established through events and concepts, and how these traces relate scenarios at different levels of abstraction, developed in different phases of the software life cycle. If support is provided for their maintenance and management, they can be used as basis for more sophisticated impact analysis as well as testing activities. For instance, a change to event *simpleEvent-select-new-channel* in the requirements scenario in Figure 1, allows the identification of the catenation with same name in the analysis scenario in Figure 2 as two possibly impacted events. Additionally, a technical decision can be made determining that channel changes will not happen any more using *the-remote-control*. Thereby, one can identify the event *simpleEvent-get-remote-control* as an impacted event, because its description refers to *the-remote-control* term. For the same reason, the catenation *simpleEvent-select-new-channel* on the analysis scenario (figure 2) will also be identified as possibly needing a change. These traces can also identify parts of the scenario that should not be impacted by a change. As is the case of the event *simpleEvent-channel-change*, which will not be impacted by the change to the remote control, since it does not refer to *the-remote-control* in its descriptions (not in the requirement scenario nor in the architecture scenario).

```
<scenario>
<title>Change Channel</title>
<summary>A user uses a remote control to change the channel.</summary>
<category>requirements</category>
<term name="the-tv">TV</term>
<term name="the-remote-control"/>
...
<catenation>
   <simpleEvent name="simpleEvent-get-remote-control" level="requirements">
      The user points the <ref type="term">the-remote-control</ref>
       to the <ref type="term">the-tv</ref>.
   </simpleEvent>
   <simpleEvent name="simpleEvent-select-new-channel" level="requirements">
      The user selects the <ref type="parameter">the-new-channel</ref>
      he wants to watch.
   </simpleEvent>
   <simpleEvent name="simpleEvent-channel-change" level="requirements">
      <ref type="term">the-tv</ref> shows
      <ref type="parameter">the-new-channel</ref> program.
   </simpleEvent>
</catenation>
</scenario>
```

Figure 1 - requirement scenario

```
<scenario>
<title>Change Channel</title>
<summary>A user uses a remote control to change the channel.</summary>
<category>analysis </category>
<term name="the-tv"/>
<term name="the-remote-control"/>
...
<catenation>
   <simpleEvent name="simpleEvent-get-remote-control" level="requirements">...
   </simpleEvent>
   <catenation name="simpleEvent-select-new-channel" level="requirements">
      <simpleEvent name="simpleEvent-new-channel-to-remote" level="analysis">
         The user presses the <ref type="parameter">the-new-channel</ref>
         on the <term name="the-remote-control"/>
      </simpleEvent>
      <simpleEvent name="simpleEvent-new-channel-to-tv" level="analysis">
         the <term name="the-remote-control"/> transmits
         <ref type="parameter">the-new-channel</ref> to
         <ref type="term">the-tv</ref>
      </simpleEvent>
   </catenation>
   <simpleEvent name="simpleEvent-channel-change" level="requirements">...
   </simpleEvent>
</catenation>
</scenario>
```

Figure 2 - analysis scenario

```
<scenario>
<title>Change Channel</title>
<summary>A user uses a remote control to change the channel.</summary>
<category>architecture</category>
<term name="the-tv">A TV is composed by a
<ref type="term">the-tv-display</ref>
<ref type="term">the-tv-tuner</ref>
</term>
<term name="the-remote-control"/>
<term name="the-channel"/>
...
<catenation>
   <simpleEvent name="simpleEvent-get-remote-control" level="requirements">...
   </simpleEvent>
   <catenation name="simpleEvent-select-new-channel" level="requirements">...
   </catenation>
   <catenation name="simpleEvent-channel-change" level="requirements">
      <simpleEvent name="simpleEvent-stop-program-display"
          level="architecture">
         <ref type="term">the-tv-tuner</ref>sends request for
         <ref type="term">the-tv-display</ref>to
         stop program display
      </simpleEvent>
      <simpleEvent name="simpleEvent-acknowledge-stop-program-display"
         level="architecture">
         <ref type="term">the-tv-display</ref> acknowledge request to stop
         displaying program to <ref type="term">the-tv-tunner</ref>
      </simpleEvent>
      <simpleEvent name="simpleEvent-blank-tv" level="analysis">
         <ref type="term">the-tv</ref> is blanked
      </simpleEvent>
      <simpleEvent name="simpleEvent-tune-channel" level="architecture">
         <ref type="term">the-tv-tuner</ref>tunes to
         <ref type="parameter">the-new-channel</ref>
      </simpleEvent>
      <simpleEvent name="simpleEvent-start-program-display"
         <ref type="term">the-tv-tuner</ref>sends request for
         <ref type="term">the-tv-display</ref> to start program display
      </simpleEvent>
      <simpleEvent name="simpleEvent-acknowledge-start-program-display"
       level="architecture">
         <ref type="term">the-tv-display</ref>
         acknowledge request to start displaying program to
         <ref type="term">the-tv-tunner</ref>
      </simpleEvent>
   </catenation>
</catenation>
</scenario>
```

Figure 3 - architecture scenario

4. RELATIONS TO OTHER ARTIFACTS

In addition to establishing and maintaining relationships of scenarios to each other within and across phases, it important to establish and maintain their relationships to other artifacts; a software architecture is one instance of such artifacts. To fully explore the potential for analysis and testing, integration between both structural and behavioral description needs to be exist. This behavioral description role is fulfilled by the architecture scenarios described above (note that architecture scenarios can have many levels of abstraction and thus be yet more detailed than the example above). In this section we take as an example the description of the scenarios' architecture and show how we relate an architectural description of the system, using xADL — an XML-based Architectural Description Language (ADL). In previous work [14] we suggested an extension to xADL that adds behavioral description to the component type by extending the type with a new element named *behavior* of type statechartType (a type described in that paper [14] and based on UML statecharts). Since one scenario describes the interaction among many components, and one component can appear in many scenarios, we take a different approach here and suggest the use of a separate schema to relate terms in a scenario in ScenarioML and components in xADL.

Thus, given a change made directly to a component in the architectural description, one can easily locate the architecture scenarios possibly impacted by such change. In consequence, test cases derived from such scenarios and possibly impacted by a change to a component, can also be located. The same is true if the relationship is followed backwards: a change is made to a term in a scenario and this term is related to an architectural component, this component possibly impacted by a change, can be identified as well. Figure 4 illustrates this idea with an example based on the scenarios and terms described above in Figures 1, 2 and 3.

```
<ComponentType id="tv-display-id">
  <Description>This representes the tv display</Description>
  <Signature id="tv-display-signature-id">
    ...
  </Signature>
</ComponentType>
```

Figure 4 - one component in xADL

```
<term name="the-tv" Identifier="the-tv-id">A TV is composed by a
  <ref type="term" Identifier="the-tv-display-id">the-tv-display</ref>
  <ref type="term" Identifier="the-tv-tuner-id">the-tv-tuner</ref>
</term>
```

Figure 5 - the TV term from scenario in figure 3

```
<ArchScenarioRelationship
   ComponentID="tv-display-id"
   ScenarioTermID="Identifier="the-tv-display-id"/>
```

Figure 6 - relation between scenario and component

Figure 4 shows one component (the TV display component) that is part of a complete TV architectural description in xADL. This component is uniquely identified by the attribute id ("*tv-display-id*"). In Figure 5 is a part of the architecture scenario from Figure 3 that describes the TV term. Here we extended the ScenarioML schema by adding that a unique identifier to the *namedMUT* type (the complex type used to describe terms). For more details, we refer the reader to [2]. Figure 6 shows one instance of the element

that relates components and scenarios through their id's: the term the-tv-display (id *the-tv-display-id*) from the architecture scenario in figure 3, is related to the TV display component (id *tv-display-id*) from the architecture in Figure 4.

In this section we showed one way to describe relationships between architecture scenarios and architectures. This information can be used to perform analysis such as ensuring that the events described in the scenarios that use certain terms are in fact supported by their related components. Additionally, this information can also be used to derive test cases in the level of architectural components. Analogous solutions can be taken to relate design scenarios to design, code scenarios to code and so forth.

5. RELATED WORK

In their work, Breitman and Leite [11] recognize the use of scenarios throughout the software life cycle, and the need for managing them. They treat scenarios as one kind of artifact that evolves throughout the software life cycle; they provide a framework for capturing traces among scenarios and among scenarios and other software artifacts. In addition, they propose a model for relating and versioning scenarios. Nevertheless, they do not get into the details of comparing different uses of scenarios, nor the possible different levels of abstraction that exist, and how these scenarios are in fact traced through different levels of abstraction down to the code level. The links among the levels of abstraction are key facets for supporting software testing activities such as measuring requirements testing coverage (by using requirement scenarios), solving the mapping need for test case generation, and performing finer-grained scenario-based regression testing. In his book [1], Alexander explains different uses of scenarios throughout software life cycle systems, but still maintains a high level view of each of the presented solutions – the approaches he presents mostly concentrate on requirements elicitation. Moreover, a detailed analysis of the relationship among the approaches is not presented. Our work goes a step further toward evaluation of the uses of scenarios in software engineering activities, and the possibility of describing them in a way to support traceability among scenarios and between scenarios and other scenario-related artifacts.

The TraceM [15] system provides an infrastructure to relate artifacts from different tools, using translation from artifacts to a common infrastructure that is based on an open hypermedia system. In addition to the infrastructure, their main contribution is being able to find implicit relationships from explicit relationships. This infrastructure could be used to manage the kinds of scenarios described in this paper as well as other artifacts, and thus provide more meaningful information to users, such as support for impact analysis across different levels of scenario abstraction and scenario-based test case generation. While they provide an infrastructure to connect artifacts, we are suggesting the use of scenarios as a means to semantically relate the concepts and events identified across scenarios and other artifacts such as architectures as explained in Section 4. Knethen [10] presents a conceptual trace model and tool that classifies entities into documentation entities and logical entities. A single documentation entity is represented by one or more logical entities; logical entities are related to the system model and documentation entities to the documentation model. Also, this approach classifies relationships among artifacts into three types. Unfortunately, this model uses use cases as the lower level entity, thereby compromising the quality of the impact analysis performed due to the constraints imposed by the structure of a use case and by the use case "includes" relationship. ScenarioML structures scenarios in a lower level of granularity, allowing us to express other kinds of relationships among this kind of entity, although those relationships are yet to be defined. Additionally, scenarios appear across different phases and thus depending on their level of granularity, so they can be considered as 'documentation' or 'logical' entities. In his work [5] Dick presents one approach to rich link traceability, applies it to requirements expressed in prose and exemplifies how these kinds of richer relationship enriches impact analysis. Our work does not describe richer relationships than identifying which concepts and events changed from one kind of scenario to another. However, we can benefit from such an idea by using rich links to relate scenarios and annotating them with more formal information to support automation of test case generation based on different kinds of scenarios, test coverage measurement, and insights on selective regression testing. Yet another approach is that of Egyed [6] who observes scenarios execution to recover traceability information between the software and its models. The scenarios recovered are what we called design scenarios, so there are still some levels of abstraction to be traced up to requirements scenarios. Therefore, we believe our ideas can complement this work.

6. CONCLUSIONS AND FUTURE WORK

This paper builds upon the fact that scenarios are artifacts used throughout software development, and explores their use as a means for artifact traceability. The tracing expressed across scenarios and between scenarios and other artifacts exemplified in Sections 3 and 4 are not the only possible ways, and we will continue exploring the full range of possibilities. As discussed, scenarios have slightly different definitions in different phases of the software life cycle, but with commonly used terms such as *events* and *concepts*. These terms were used to trace scenarios in a single phase or across phases.

We still need to further investigate the feasibility of maintaining these traces and its impact to the scalability of an infrastructure like that. Additionally, we are aware of the fact that establishing such traces might be a burden that developers are not willing to take. What should be noted though is that in addition to other benefits well know from traceability, these traces can provide richer change impact analysis and support software testing activities with gains in terms of automation of test case generation, coverage measurement and regression testing. We aim at proposing a framework that will support scenario-based testing activities based on the traceability among scenarios and other artifacts that must be established and maintained to enable automated or semi-automated support for testing activities. Thereby, this framework should describe (1) the information needed in the scenario specification; (2) kinds of relationships among scenarios (as well as information that should go into these relationships); and (3) other ways to relate scenarios to other artifacts. This work will be based on ScenarioML, which is still evolving, and we plan to extend it to address testing needs.

We also recognize the fact that a major problem in the traceability world is maintaining the traces among artifacts. Traces from code to design scenarios can be recovered for instance, by monitoring execution of test scenarios [6]. For that reason, we will also explore the composition of two approaches: One is to recover the traceability by using reverse engineering, or 'trace dependency

analysis', and the other is maintain the links down to the level recoverable by the first approach.

7. ACKNOWLEDGMENTS

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

8. REFERENCES

[1] Alexander, I. F. and Maiden, N., "Scenarios, Stories, Use Cases Through the Systems Development Life-Cycle", John Wiley & Sons, 2004.

[2] Alspaugh, T. A., "Temporally Expressive Scenarios in ScenarioML", Institute for Software Research Technical Report UCI-ISR-05-06, University of California, Irvine. May 2005.

[3] Ant´on, A. I. and Potts, C. , "A Representational Framework for Scenarios of System Use", Requirements Engineering Journal, 3(3–4):219–241, Dec. 1998.

[4] Dashofy, E.M., van der Hoek, A. , and Taylor, R.N., "A Highly-Extensible, XML-Based Architecture Description Language" , Working IEEE/IFIP Conference on Software Architecture, 2001.

[5] Dick, J., "Rich Traceability", 1st International Workshop on Traceability in Emerging Forms of Software Engineering, at the 17th IEEE Conference on Automated Software Engineering (ASE), Sept. 28.

[6] Egyed, A. F, "A Scenario-Driven Approach to Trace Dependency Analysis", IEEE Transactions on Software Engineering, 2003. 29(2):116-132.

[7] Hartman, A. and Nagin, K., "The AGEDIS Tools for Model Based Testing" in International Symposium on Software Testing and Analysis , 2004.

[8] Jarke, M., Tung Bui, X. and Carroll, J. M. . "Scenario management: An interdisciplinary approach". Requirements Engineering Journal, 3(3-4):155-173, 1998.

[9] Jeremiah, W. and Frank, M., "Using UML to Partially Automate Generation of Scenario-Based Test Drivers", 7th International Conference on Object Oriented Information Systems, 303-306.

[10] Knethen, A. von, "Automatic Change Support based on a Trace Model", 1st International Workshop on Traceability in Emerging Forms of Software Engineering at the 17th IEEE Conference on Automated Software Engineering, Sept. 28.

[11] Leite, J. C. S. P. and Breitman, K. K., "Experiences Using Scenarios to Enhance Traceability", 2nd International Workshop on Traceability in Emerging Forms ofSoftware Engineering at the 18th IEEE Conference on Automated Sofware Engineering, Oct.6-10.

[12] Medvidovic, N., Rosenblum, D. S., Redmiles, D.F. and Robbins, J.E., "Modeling software architectures in the Unified Modeling Language", ACM Transactions on Software Engineering and Methodology, Jan 2002. 11(1).

[13] Muccini,H., Bertolino, A. and Inverardi, P. "Using Software Architecture for Code Testing", IEEE Transactions on Software Engineering, March 2004. 30(3):160-171.

[14] Naslavsky, L., Xu, L., Dias, M., Ziv, H., and Richardson, D., "Extending xADL with Statechart Behavioral Specification", Proceedings of the Twin Workshops on Architecting Dependable Systems at International Conference of Software Engineering, May 25, 2004.

[15] Sherba, S. A., Anderson, K. M. and Faisal M., "A Framework for Mapping Traceability Relationships," 2nd International Workshop on Traceability in Emerging Forms of Software Engineering at 18th IEEE International Conference on Automated Software Engineering, Oct., 2003.

[16] G. Spanoudakis and A. Zisman, "Software Traceability: A Roadmap", Advances in Software Engineering and Knowdledge Engineering, (ed) S.K Chang, World Scientific Publishing, 2005.

[17] Uchitel, S., Chatley, R., Kramer , J. and Magee, J., "System Architecture: the Context for Scenario-based Model Synthesis", 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering, 33-42.

[18] Weidenhaupt, K., Pohl, K. Jarke, M., Haumer, P., "Scenarios in System Development: Current Practice", IEEE Software, March/April 1998. 15(2).

[19] http://www2.telelogic.com/products/doorsers/doors/

[20] http://www-306.ibm.com/software/awdtools/reqpro/