A Pattern System for Tracing Architectural Concerns

Mehdi Mirakhorli
DePaul University, School of Computing
Chicago, IL 60604, USA
mehdi@cs.depaul.edu

Jane Cleland-Huang
DePaul University, School of Computing
Chicago, IL 60604, USA
jhuang@cs.depaul.edu

ABSTRACT

A software architecture is carefully designed to satisfy the quality concerns of its stakeholders, and as such, represents a systematic and intricately balanced set of design decisions which deliver required qualities such as performance, reliability, and safety. In practice, architectural degradation tends to occur over the lifetime of the software system, as developers make ongoing and incremental maintenance changes to the system without knowledge of its underlying design decisions. Fortunately, this problem can be alleviated by establishing traceability between concrete elements in the architecture and their associated design decisions, and then using these traceability links to keep developers informed of relevant architectural tactics, styles, and design patterns throughout the development process. This paper focuses on the task of creating and using such traceability links. We present six trace creation patterns describing techniques and supporting structures for creating architecturally significant traceability links, and two usage patterns describing techniques for using the created links to help preserve qualities in the architectural design. The patterns described in this paper emerged from our experiences and observations of tracing architectural concerns in safety critical systems.

Keywords

Architecture, traceability, tactics, patterns

1. INTRODUCTION

The architectures of complex software systems are designed and implemented to satisfy a wide variety of competing goals related to qualities such as reliability, dependability, safety, security, performance, and usability [1]. These quality goals are realized through a series of carefully evaluated architectural decisions [24, 4, 13] which work together to shape the structure, behavior, properties, processes, and governance of the delivered solution [28].

Unfortunately, despite significant efforts that go into delivering a high quality architectural solution, its quality can be slowly eroded by ongoing maintenance activities which are inevitably undertaken to correct faults, improve performance or other quality concerns, and to adapt the system in response to changing requirements [38, 22]. Even seemingly innocuous changes made to design or code, can often inadvertently lead to degradation of architectural qualities [13, 16]. Such degradation can be partially prevented by making design decisions visible to software engineers so that as they perform maintenance activities they are fully informed of the relevant underlying design patterns, tactics, and constraints [3]. This requires the use of traceability links to establish relationships between design decisions and architectural elements in visual models or implemented code. However, individual architectural decisions are often cross-cutting in nature and therefore impact numerous architectural components, exhibit complex interdependencies, and contribute to satisfying multiple quality goals [25, 34, 10]. This introduces a dilemma. On one hand it can be difficult and costly to trace quality concerns into the architectural design and the end result may involve creating and maintaining an almost impossible number of traceability links. On the other hand, failing to trace architectural concerns, leaves the system vulnerable to problems such as architectural degradation.

To address these problems we present a set of traceability patterns, each of which captures a reusable solution for creating and using traceability links in the architectural context. In software engineering, patterns are a relatively popular concept for capturing existing, proven practices and solutions in software development. They support various software engineering tasks by providing solutions for common occurring problems within a clearly defined context. The notion of patterns first emerged with the seminal work on design patterns[17], and has since been extended into different areas of software engineering. In the context of software system traceability, a pattern refers to a general solution that can be applied to a common, recurring problem related to creating and using traceability links in support of various tasks such as architecture design reasoning, and change impact analysis.

In the pattern system presented in this paper, traceability patterns are categorized according to whether they describe *trace creation* or *trace usage*. *Trace creation* patterns present solutions for establishing traceability links between quality goals and related architectural concerns, while *trace usage* patterns present techniques for utilizing the created links to support specific architectural tasks. All of the patterns are designed for tracing architecturally significant requirements that are satisfied through one or more architec-

PLoP 2011

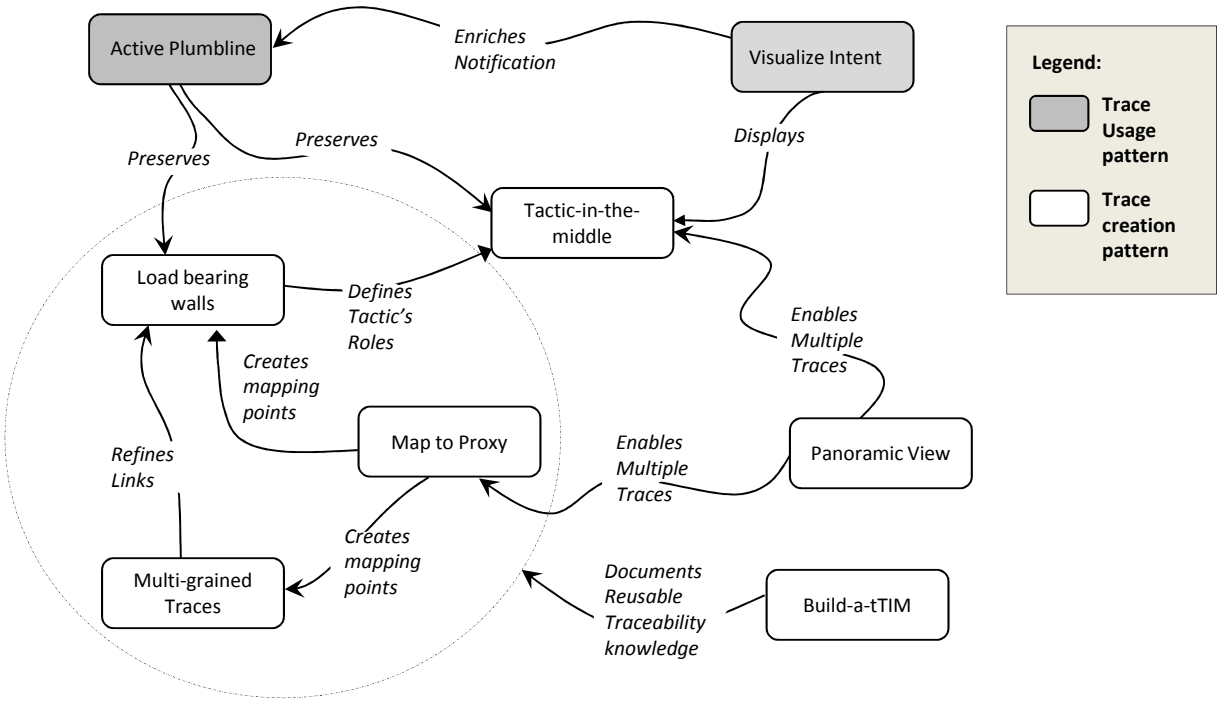


Figure 1: Pattern System: Traceability Patterns and their relationships

tural decision. Although such decisions may include the use of architectural styles, frameworks, design patterns, or tactics, we use the term *tactic* to refer to all of them throughout the remainder of the paper.

2. TRACEABILITY PATTERNS

In this section we provide a brief overview of the eight patterns presented in this paper. There are six different trace creation patterns which are explained further in section 4, and two different trace usage patterns which are explained in section 5. *Trace creation* patterns are designed to minimize the cost and effort of creating and maintaining traceability links while maximizing their potential benefits. *Trace usage* patterns provide solutions for utilizing the created traceability links to support activities such as change impact analysis, architectural knowledge sharing and software program comprehension, all of which provide support for architectural preservation. The patterns are as follows.

Tactic-in-the-Middle: This pattern describes a tactic-centric approach in which all traces are centered around a tactic. Pre-tactic traces extend back to requirements and quality goals, while post-tactic traces extend forward to the architectural elements in which the tactic is realized. Placing an architectural decision, such as a tactic in the center of the traceability graph reduces complexity and groups similar-links together.

Load Bearing Walls: This pattern describes a solution in which a tactic (or other architectural decision) is modeled as a traceability information model (TIM). The key roles, i.e. the load bearing walls, of the tactic are captured and modeled in the TIM. The TIM is then used as a reference model by project stakeholders as they establish semantically typed traceability links between the tactic and related architectural elements.

Map to Proxy: This pattern describes how to create a proxy for each traceable element in the TIM, so that traceability links can be created by simply mapping concrete architectural elements to a proxy. The use of trace proxies has two primary benefits. First, it transforms the more complex task of traceability to a simple mapping task, and secondly, it enables re-use of the TIM's previously defined traceability links, thereby reducing the number of project-specific traces which need to be created and maintained.

Panoramic View: This pattern describes how to establish traceability links across different architectural and code views of the system. This is necessary when a given tactic is visible across multiple views. The solution to this problem is to allow a single proxy in the TIM to be mapped to heterogeneous elements, across multiple views.

Multi-grained Traces: This pattern describes how to explicitly differentiate between fine and coarse-grained traceability links in the TIM, so that trace creators can generate traces at either level. This is useful as there is a definite trade-off in the costs and benefits of creating coarse versus fine-grained links.

Build-a-tTIM: Applying the first three patterns of *Tactic-in-the-Middle*, *Load Bearing Walls*, and *Map to Proxy* for a specific tactic or a group of related tactics, results in what we refer to as a Tactic Traceability Information Model (tTIM). This *Build-a-tTIM* pattern describes the process for creating re-usable tTIMs, which can represent significant cost and effort savings when the tTIMs are reused in future projects.

Active Plumbline: Although many organizations invest significant cost and effort into the traceability process, the traceability links are not always used effectively during maintenance activities. As a result, a Software Engineer

may modify a component without fully understanding the underlying architectural decisions. This pattern describes how traceability links can be used to register and monitor architecturally significant elements, so that maintainers working on registered components can be kept informed of underlying architectural decisions.

Visualize Intent: Software architectures often include very complex interdependencies and trade-offs, and are developed according to an extensive set of carefully considered design decisions. These complexities are difficult to adequately portray in a text-only format. This pattern therefore describes how tTIMs can be used to effectively visualize the underlying architectural decisions.

These eight patterns interact with each other to form the pattern system depicted in Figure 1. The interrelationships between patterns are discussed in greater depth throughout the remainder of this paper.

3. GUIDE TO NOTATION

We use two different kinds of diagram to describe the patterns and to illustrate them with concrete examples. Patterns are described using an informal Line-and-Box drawing style, while the concrete examples are presented using a UML profile which defines the design elements, implementation classes, variables and other artifacts. This section describes these two notations.

Pattern descriptions: Each pattern is presented using a simple box and line notation in which boxes and their textual labels represent traceable elements while lines represent traceability links. Traceable elements can include tactics, qualities, goals, design, or implementation elements. Design rationales associated with each architectural tactic are depicted using the standard document notation. Proxy elements, used in the *Map to Proxy* pattern, are depicted with dashed borders.

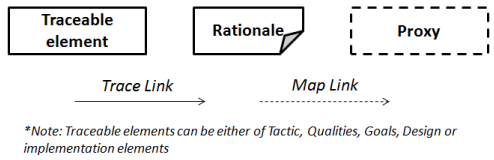


Figure 2: Schematic notation to present patterns

Concrete examples: Concrete examples of each tactic traceability pattern are presented using a customized UML profile in which each type of traceable element is represented with its own symbol. The following legend describes this profile.

Elements such as Requirements, Business Goals, Rationales, and Tactics are used in the same way across all the examples; however Roles and Parameters, which represent the internal structure of a tactic, use a variety of different stereotypes. Some of these stereotypes are unique to a given tactic, while others are shared across multiple tactics.

The profile includes two types of connectors, the first type is depicted as a solid line, and represents a relationship internal to the tactic, while the second type is depicted as a dashed line, and represents an external association. Both types of links can be stereotyped to represent the purpose of the link. For example *Maps* indicates that an external element is mapped to an element of the TIM, while *Helps* indicates that a tactic contributes towards satisfying a goal.

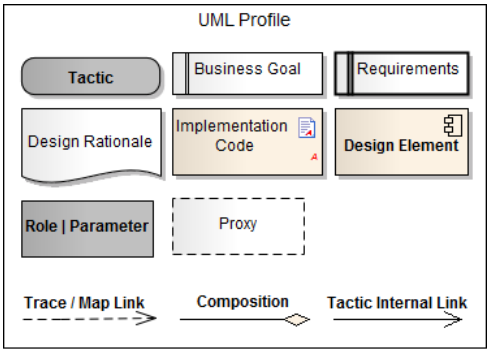


Figure 3: UML-Based notation used to illustrate each pattern

4. TRACE CREATION PATTERNS

In this section we provide a more detailed explanation for each of the six trace creation patterns. Each of the patterns is illustrated using an example taken from the NASA Crew Exploration system (CEV) from NASA's Constellation System of Systems. This system is an exploratory vehicle designed to provide round trip transportation for human crews between Earth and Space. The CEV is designed to coordinate with transfer stages, landing vehicles, and surface exploration systems in order to support manned voyages to the Moon and beyond. Requirements, architectural decisions, and architectural models of CEV were obtained from publicly available documentation manuals [20], [37]. Throughout this paper we describe all of the patterns with respect to the following architectural decision to use semantic based scheduling with task sequencing in the CEV System.

CEV_Tactic1: Semantic Based Scheduling with Task sequencing.
CEV_Quality_Goals: CEV_Performance, CEV_Reliability.
CEV_Requirement1: When the CEV is performing automated maneuvers during any phase of the mission, if it experiences unexpected and significant loss of communication with Earth, it will return the crew to Earth.
CEV_Rationale1: This tactic guarantees the real-time accomplishment of both nominal and trajectory missions, task by task based on mission phases and situations.

4.1 Tactic-in-the-Middle

Context: Any highly dependable, safety critical system which incorporates requirements management, architectural reasoning, and design maintenance.

Problem Statement: Most safety critical domains mandate traceability to support activities such as compliance verification, requirements validation, and impact analysis [8] [2] [5] [34]. It is especially important to understand the way in which critical quality concerns such as reliability, performance, and security, are realized in the design so that these decisions can be carefully maintained throughout the lifetime of the system [1] [29]. Unfortunately, tracing quality concerns into architectural designs or code, can be quite challenging, primarily because many quality concerns im-

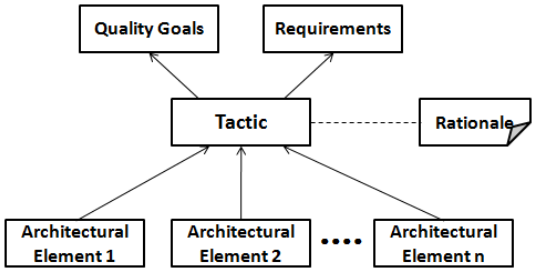


Figure 4: Structure of *Tactic-in-the-Middle* Pattern

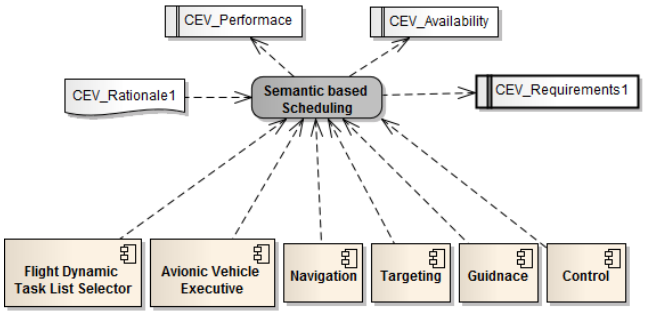


Figure 5: "Semantic Scheduling Decision'' implemented as *Tactic-in-the-Middle*

pact the architecture in broad and diverse ways, and therefore result in a proliferation of traceability links [9]. Furthermore, rationales of trace relationships are often lost, inhibiting future reasoning and comprehension of the system [31] [33].

Solution: Quality requirements are primarily satisfied through a series of architectural decisions which lead to the implementation of specific frameworks, design patterns, or tactics (referred to collectively as tactics in the remainder of this pattern). Traceability links should therefore be established between quality requirements and architectural elements via these tactics. As depicted in Figure 4, this requires establishing pre-tactic traces between tactics and quality requirements, and post-tactic traces between tactics and architectural decisions.

Example: Figure 5 depicts an example from the NASA Crew Exploration system showing how traceability links are established between the *semantic scheduling* tactic and performance and availability goals, the rationale behind this decision, and an associated quality requirement. Furthermore, it shows how downstream traceability links are established to scheduled tasks including the *Flight Dynamic Task List Selector* and the *Avionic Vehicle Executive*. The tactic is also traced to a previously documented rationale and requirement.

Related Patterns: *Tactic-in-the-Middle* is often used in conjunction with *Load Bearing Walls*. *Load Bearing Walls* allows an analyst to semantically type the traceability links generated by the *Tactic-in-the-Middle* pattern.

4.2 Load Bearing Walls

Context: Traceability links are created between architectural elements (visible in either code or design views)and quality goals and constraints. The semantics of each link are either defined manually by the user at the time of trace creation, or else are inferred through examining the source and target of each trace link.

Problem Statement: In practice, users tend to create trace links without explicitly defining the semantics of the link [36] [6]. This makes it difficult for trace links to be used in an automated way, and also reduces their understandability. This is especially true in tracing architectural concerns where a single quality concern may trace to multiple architectural elements, and a single architectural element may trace back to multiple quality concerns [9]. The lack of semantic typing reduces the usefulness of such links [26][34].

Solution: Identify the "Trace Link Types'' of each tactic. Create a Traceability Information Model (TIM) that clearly depicts key roles of a tactic which can be traced into the design and implementation. Also identify the set of external links which connect the tactic to its environment. External links are semantically typed according to the target of the link, for example a link from the tactic to a related quality goal might be semantically typed as *helps*, while a link from a rationale to the tactic might be semantically typed as *justifies*. The TIM is used to guide the creation of trace links and to provide information about the relationships between various architectural components. The structure of this pattern is depicted in Figure 6.

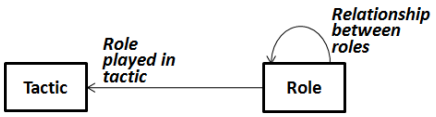


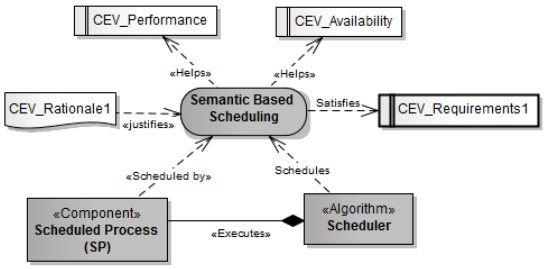
Figure 6: Structure of *Load Bearing Walls*

Example: Figure 7(a) shows how the *semantically based scheduling* tactic, previously depicted in Figure 5 as a black-box tactic, is decomposed into specific roles and attributes, and then augmented to show semantically typed traceability links. A TIM is informational only. It shows that the key roles in the tactic are *scheduled processes* and the *scheduler*. The trace user can utilize this information to make informed traceability decisions.

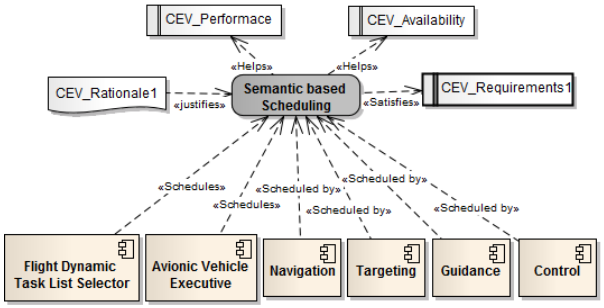
Related Patterns: *Load Bearing Walls* requires use of the *Tactic-in-the-Middle* pattern. It adds semantics to the trace links. *Map to Proxy* allows an analyst to create proxies for each of the key roles identified in *Load Bearing Walls*, thereby transforming the tracing task to a simpler mapping task. This pattern can be refined by *Multi-grained Traces* to differentiate between coarse-grained and fine-grained traceability links. Finally, *Active Plumbline* can be used in conjunction with *Load Bearing Walls* to keep developers informed of underlying architectural decisions during the maintenance process.

4.3 Map to Proxy

Context: A project manager has determined that traceability links must be established between quality goals and architectural elements. Typically, such links are created at coarse-grained levels and stored in a trace matrix [14]. Even when a traceability information model is present, the trace creator must make a series of decisions to determine the source and target artifacts for each link, and must also doc-



(a) Constructing a TIM to define the link semantics and guide trace creation



(b) Final traceability links after using the pattern

Figure 7: Example Implementation of *Load Bearing Walls*

ument the meaning of each link [18] [6] [23] [36]. Additional information such as rationales, trade-offs, and decisions etc must be documented separately and then integrated into the trace matrix [32] [33].

Problem Statement: The effort needed to plan, create, and maintain traceability links can serve as a prohibitive implementation barrier [23] [8]. As a result, trace users often create a very minimal set of untyped traceability links, which provide only limited benefits for ongoing software engineering activities [36].

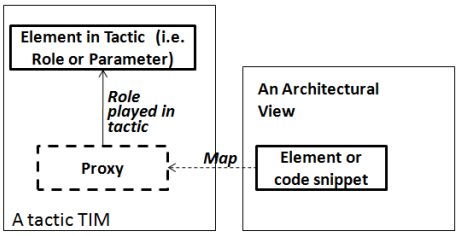


Figure 8: Structure of *Map to Proxy*

Solution: Instead of using a tactic's TIM (described in the *Load Bearing Walls* pattern) for informational purposes only, instantiate the TIM and re-use its internal and external links as active traceability links within the current project. Furthermore, for each role defined as a load bearing wall, create a proxy element in the TIM to represent the concrete architectural element that actually satisfies that role. Replace the tracing task with a simple mapping task in which the project stakeholder establishes a traceability link by mapping an architectural element to a relevant proxy in

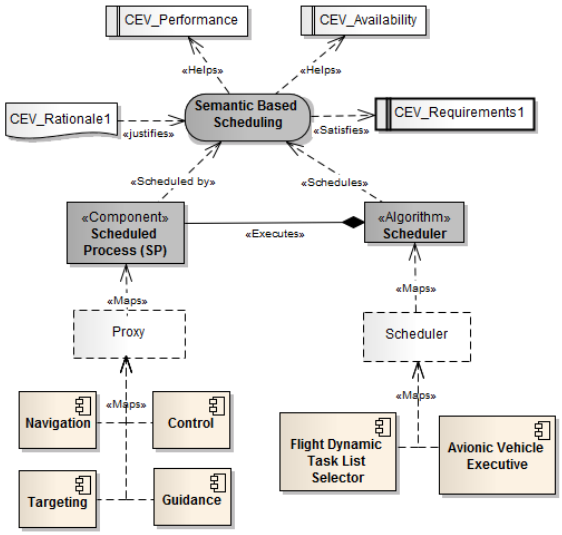


Figure 9: Architectural elements mapped to proxies in the Semantic based Scheduling tactic

the TIM. The primary benefits of this approach are simplicity, and effort reduction. Once an architectural element is mapped to a proxy, it inherits all of the semantically typed traceability links defined for that tactic, meaning that fewer project specific links need to be created and maintained.

Example: Figure 9 depicts two proxies modeled for the *semantic based scheduling* TIM. These proxies are for tracing *scheduled processes* and for *schedulers*. The figure also depicts actual architectural elements mapped to the proxies. For example, the *Flight Dynamic task list selector* and *Avionic Vehicle Executive* components are mapped to the *scheduler* proxy while the *Navigation* component is mapped to the *Scheduled Process (SP)* Proxy. In each of these cases, establishing the mapping delivers traceability all the way from architectural elements, via the tactic, to quality goals.

Related Patterns: *Map to Proxy* requires the use of *Load Bearing Walls* and creates mapping points for the underlying tactic's key roles. *Panoramic View* is often used with *Map to Proxy* to allow a single proxy to map to elements in multiple architectural views. *Multi-grained Traces* can be used on top of *Map to Proxy* to facilitate the creation of fine grained traces.

4.4 Panoramic View

Context: Architectural decisions are visible across multiple views including both architectural and code views. For example, a particular decision might be traced to a single component in the decomposition view, and also to a set of concurrent threads in a runtime view. Almost every architectural decision needs to be mapped to both code and one or more architectural design views.

Problem Statement: Architectural documentation is notoriously incomplete. Although practitioners may follow a standard documentation approach such as Kruchten's 4+1 Views [27], actual architectural decisions are inconsistently documented. Some decisions do not appear explicitly in any architectural views, while others appear across multiple architectural diagrams. A single element in a tactic often

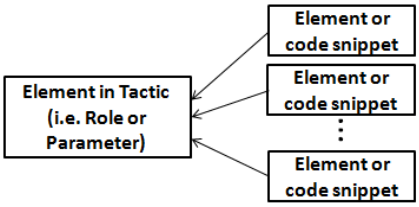


Figure 10: Structure of *Panoramic View*

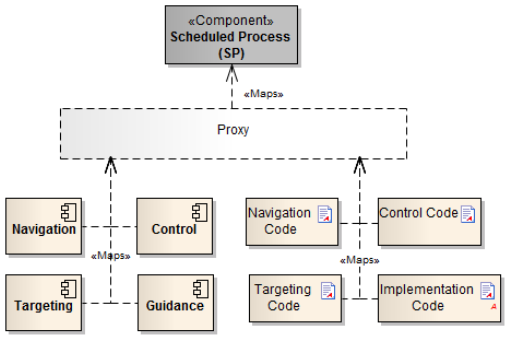


Figure 11: Mapping the Scheduled Process role to a deployment and code view

needs to be traced to multiple components [9] [31].

Solution: The *Panoramic View* pattern allows an architectural decision (i.e. the entire tactic, a role in the tactic, or a proxy for a role) to be traced to multiple architectural views. This is depicted in Figure 10. A special case occurs when the architectural decision represents tacit knowledge, or is so dispersed across the architectural design and the code, that it is not possible to physically map the role to any specific components or views. In this case traceability is considered implicit.

Example: Figure 11 provides an example in which the *Scheduled Process* component from the *semantically based scheduling* tactic is traced to both the *runtime component view* in the architectural documentation, and also the *code view*. It should be noted that these maps do not establish traceability directly between associated architectural and code views. In other words, no explicit traces are established between the navigation component in the deployment view and the navigation code. However, this can be accomplished either implicitly through naming conventions, through an additional trace matrix, or through refining the link types in each of the individual traces. Furthermore, most architectural modeling tools will automatically track cases in which multiple elements appear in multiple views.

Related Patterns: *Panoramic View* is often used with Map to Proxy, as it allows a single proxy to map to elements in multiple architectural views. *Panoramic View* can also be used with *Tactic-in-the-Middle* to trace the tactic to elements in multiple architectural views.

4.5 Multi-grained Traces

Context: There is a well known trade-off between the costs and benefits of tracing at coarse versus fine-grained levels of granularity [21] [15] [11]. Coarse-grained traceability links are easier to create and maintain, but provide less accuracy when the traceability link is actually used. Conversely, fine-grained links require greater effort to create and maintain, but provide greater accuracy in pinpointing impacted elements. Project stakeholders make traceability decisions that take these costs and benefits into consideration.

Problem Statement: Trace users often do not understand how to create fine-grained traces to a tactic without generating a superfluous number of links. As such they tend to create only high-level untyped links, which are not very useful for supporting the automation of critical software engineering tasks such as impact analysis or architecture preservation. **Solution:** Differentiate between the primary roles and supporting attributes of a tactic and model them in the TIM. Coarse-grained traces are generally established by tracing the primary roles of a tactic, while fine-grained links are established by tracing finer grained elements such as attributes or configuration files. Therefore, clearly differentiate between elements that should be traced in coarse-grained strategies, versus those additional links which should be traced if finer-grained traces are desired. Make these decisions visible to trace users so that they can choose to create either coarse-grained or multi-grained traceability links. The structure of this pattern is depicted in Figure 12.

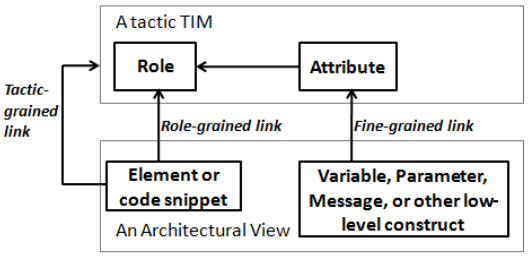


Figure 12: Structure of *Multi-grained Traces*

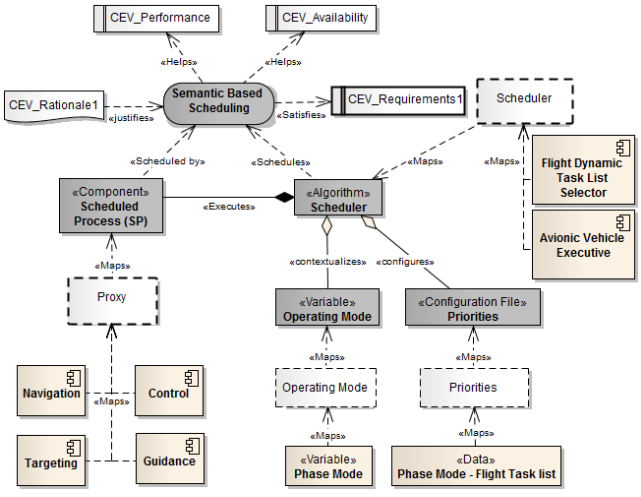


Figure 13: Semantic based Scheduling tactic showing coarse-grained traces with fine-grained proxies left unmapped

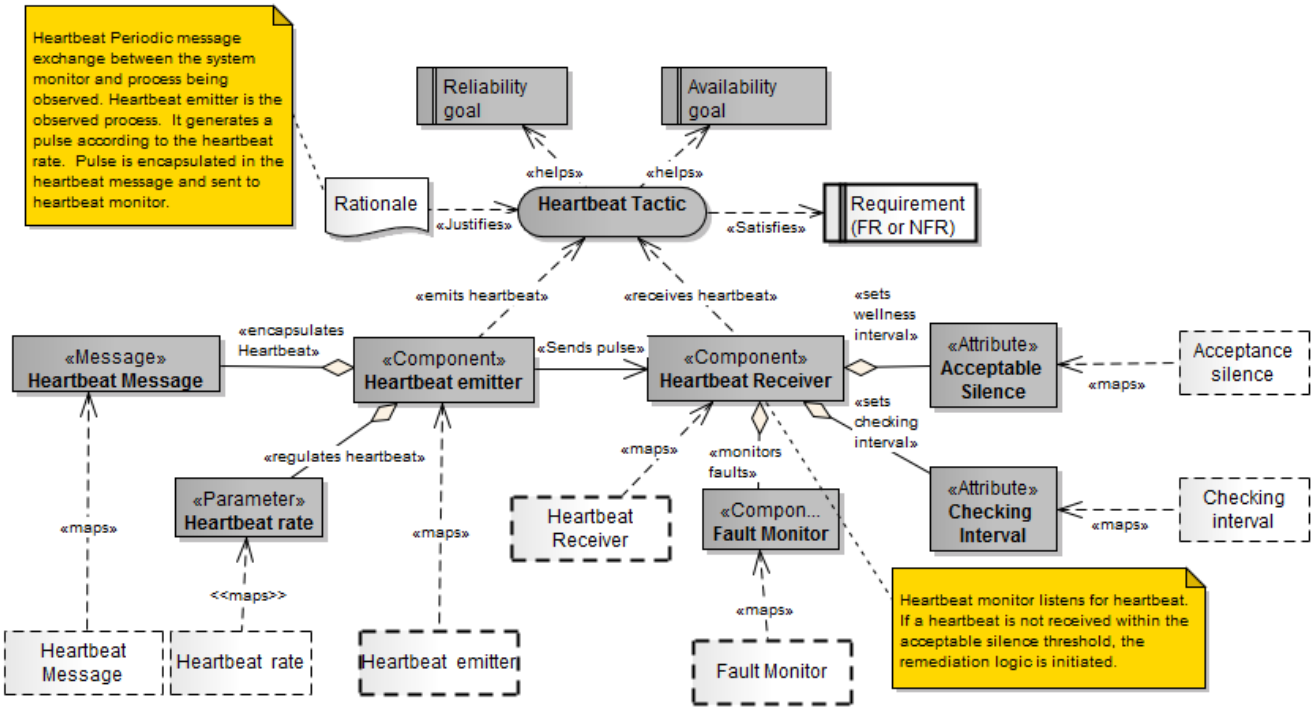


Figure 14: Tactic Traceability Information Model for Heartbeat Tactic

Example: Figure 13 shows how the traceable elements (depicted as proxies) are divided into coarse and fine-grained trace targets. The *scheduler* is configured by a configuration file and contextualized according to the current operating mode. Therefore in this example, the user has chosen to trace at the coarse-grained level to the *Scheduler* and *Scheduled Processes* proxies and trace at the fine-grained to the *operating modes* and *priorities* proxies. Coarse-grained trace proxies are assigned darker borders than fine-grained ones.

Related Patterns: *Multi-grained Traces* can be used with either *Tactic-in-the-Middle*, *Map to Proxy* or *Load Bearing Walls* to support traceability at different levels of granularity.

4.6 Build-a-tTIM

Context: Architectural frameworks, styles, design patterns, and tactics tend to be reused in similar ways across multiple projects [19]. This is particularly true within a single organization, in which similar architectural decisions are often made across families of solutions.

Problem Statement: Tracing architectural concerns using tactics requires significant planning. The use of patterns such as Tactic-in-the-Middle, *Map to Proxy*, *Load Bearing Walls*, and *Multi-grained Traces* requires skill and domain knowledge. Project stakeholders may lack these skills or may be unwilling or unable to invest the necessary effort to implement them in an individual project.

Solution: Create a set of reusable TIMs for commonly adopted tactics, store them in a reusable traceability knowledge base, and reuse them across projects. We refer to reusable tactic-specific TIMs as *Tactic Traceability Information Models* (tTIMs) [35].

Example: The *semantically based scheduling* tTIM shown in Figure7(a) and and the *heart beat* tTIM shown in Figure14 provide illustrations of reusable tTIMs. A tTIM can be reused as-is and mapped to specific architectural components found in the particular application in which it is applied.

Related Patterns: *Build-a-tTIM* is always used on top of *Load Bearing Walls*, *Map to Proxy*, and *Multi-grained Traces* patterns in order to create a reusable and instantiable traceability information model for a particular tactic.

5. TRACE USAGE PATTERNS

In this section we describe two additional patterns which utilize the created traces to support tasks related to architectural preservation.

5.1 Active Plumbline

Context: A software architecture has been carefully designed to satisfy a set of stakeholder quality goals. The architecture incorporates a varied set of architectural decisions, some of which are more visible to developers than others.

Problem Statement: During the long-term maintenance of a software system, it is easy for architectural quality to gradually erode as changes are applied to the design and code [38, 25, 4, 13]. This problem is exacerbated when developers making the changes are unaware [30, 12, 3]of underlying architectural decisions and their rationales. While some of these decisions, such as the use of layers, may be clearly evident in the design, other decisions may be quite obscure. Unless developers are kept informed of underlying design decisions, they may inadvertently make a change which degrades the overall quality of the system. Furthermore, although many projects document architectural de-

cisions in an informal way, this documentation is typically only available if the developer pro-actively searches for it [33].

Solution: Register all critical architectural elements with a central coordinator, monitor those elements for modifications, and generate informational messages to keep the maintainer informed of underlying architectural decisions [35, 7, 9]. This pattern is typically implemented through automatically registering any architectural elements that are traced to elements in a tactic. The pattern is constrained by the degree of automated support provided for event monitoring in the modeling or development environment. The sequence of these actions is depicted in Figure 15.

Example: Consider the case of the *semantic based scheduler* tactic. All of the traced components are registered with the central observer. If a human maintainer modifies the *Phase Mode - Flight task list* he or she will be notified that this list configures the scheduler, which in turn schedules the navigation, control, targeting, and guidance systems.

As a second example, Figure 14 shows the tTIM for the Heartbeat tactic used in a Lunar Robot application, and Figure 16 shows a snapshot of design models in which the heart-beat emitter role is mapped to the *Obstacle Detection* component, the heart-beat monitor to the *Path Planning 1* component, and the *fault monitor* to the Navigation Health Management component. As a byproduct of the mappings, each of these three components are registered with the central coordinator. When the developer makes changes to the component named "Obstacle Detection" within the Enterprise Architect's IDE. The component is immediately annotated to indicate it is load-bearing. All underlying architectural decisions are then displayed.

Related Patterns: *Active Plumbline* can be used with *Tactic-in-the-Middle* and *Load Bearing Walls*, to help maintain a tactic in the architectural design by monitoring the tactic's critical elements. *Visualize Intent* can be used with *Active Plumbline* to visualize warnings and notifications displayed to the maintainer.

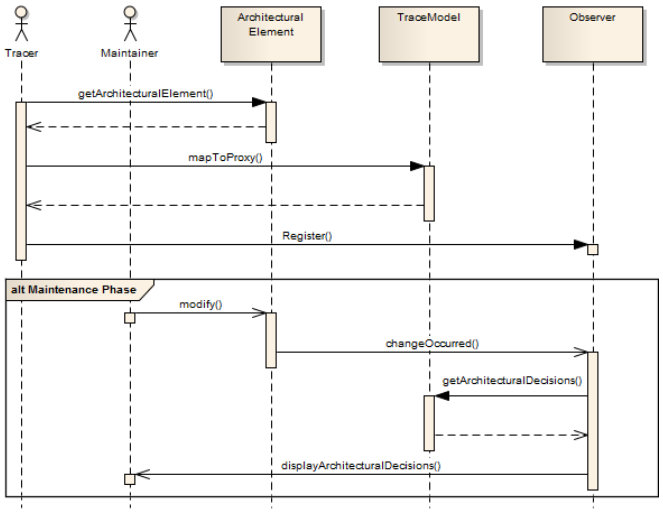


Figure 15: Monitoring Critical Architectural Element during Maintenance

5.2 Visualize Intent

Context: Architectural decisions are often far-reaching and exhibit numerous trade-offs and dependencies. Certain load-bearing elements therefore contribute towards realizing multiple architectural decisions. Such decisions need to be available to developers as they construct and maintain software systems.

Problem Statement: Most existing techniques document and present architectural decisions in a textual format, creating a disconnect between the decision and its implementation in a design. Textual lists of architectural decisions often fail to fully communicate the complex interactions between decisions, and also fail to display the full impact of the decision upon various elements of the architectural design [30, 12, 3].

Solution: Use the infrastructure of the tTIMs and their traces to concrete architectural elements to visualize architectural decisions [35]. Present relevant architectural decisions to developers using the visual format of the developed tTIM. Map this onto the actual architectural components.

Example: Figure 16 not only depicts the way critical architectural elements are monitored, but also illustrates the visualization of the heart-beat tactic during a change event. In the main screen of the Enterprise Architect IDE, when the user modifies the obstacle detection component, a "load bearing" symbol (i.e. a pyramid) is displayed on the modified component, and the underlying heart-beat tactic is visualized in a special panel on the right-hand side of the screen. Both sections of the screen are color coordinated to make it easier for the user to understand the underlying role of each architectural component.

Related Patterns: *Visualize Intent* can be used with *Tactic-in-the-Middle* and *Active Plumbline* to visualize information presented to the maintainer in order to keep him or her informed of underlying design intent.

6. CONCLUSIONS

This paper has a presented a pattern system for tracing architectural concerns. Each of the patterns was discovered through an extensive study of architectural decisions in safety critical systems, combined with our own experiences of establishing and utilizing traceability links in such systems.

The *trace creation* patterns describe reusable solutions for establishing and maintaining traceability between quality goals, tactics, and concrete architectural elements. These patterns are designed to improve the quality of the established traceability links in order to maximize their benefits, and to minimize the effort needed to create those links. As described in the "related patterns" sections of each pattern description, a trace user can opt to use only a single pattern, such as *Tactic-in-the-Middle* or *Multi-grained Traces*, or else could utilize a set of interdependent patterns in order to establish a more sophisticated traceability infrastructure. All of these individual trace creation patterns provide useful guidance for improving the effectiveness of the traceability effort; however longer term benefits can be realized if traceability knowledge is documented and reused through implementing the *Build-a-tTIM* pattern. Our current work in this area involves creating and validating a generic library of tTIMs for commonly used tactics [35].

The *trace usage* patterns focus upon architectural preser-

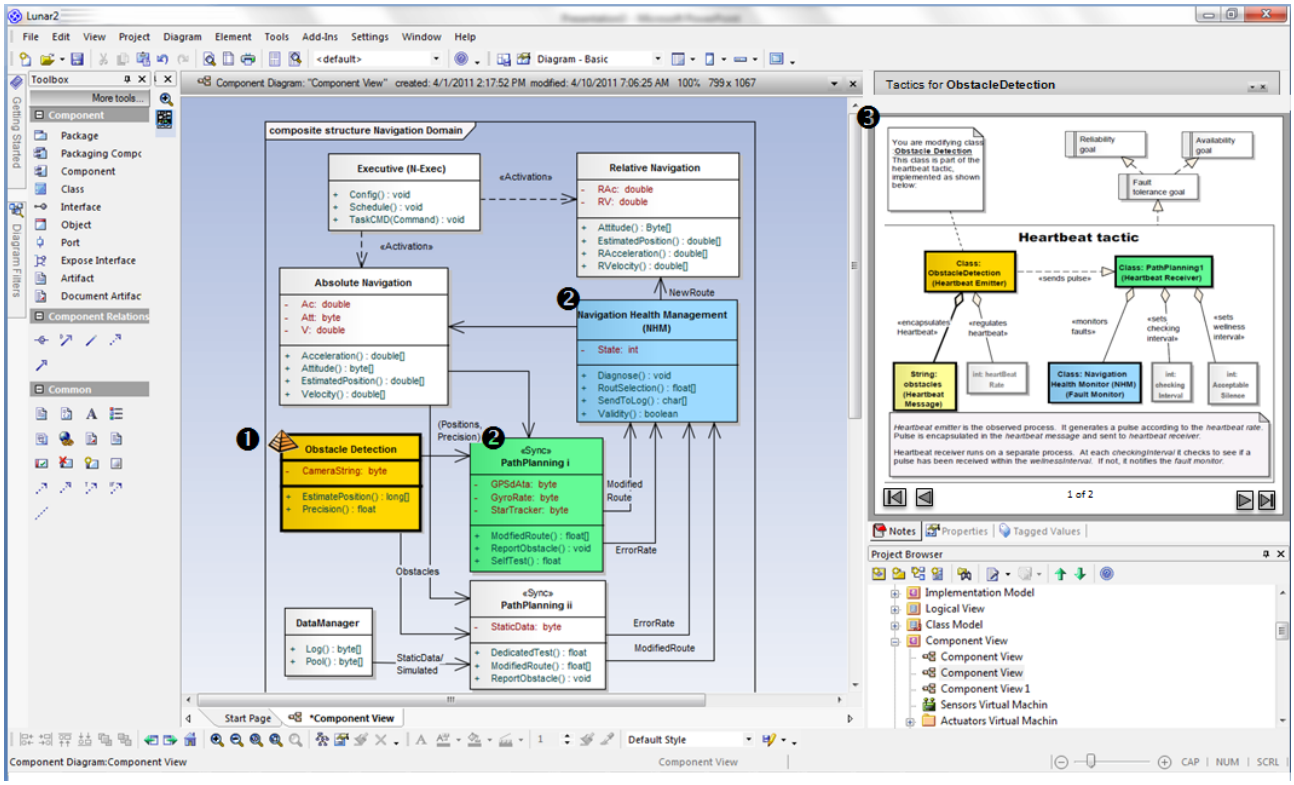


Figure 16: Visualizing architectural tactics within Enterprise Architect

vation during long-term maintenance activities. They utilize the created traceability links to support critical activities such as impact analysis, compliance verification, and keeping maintainers fully informed of underlying architectural concerns. This paper documents the two trace usage patterns we have found to be most effective; however in future work we plan to identify and document a more extensive set of usage patterns.

7. ACKNOWLEDGMENTS

The work described in this paper is partially funded by the National Science Foundation under grants CCF-0447594 and CCF-0810924. We thank Bob Hanmer for initially challenging us to document our work on architectural traceability in the form of patterns. We also thank Sam Suppakul, who, in his role as shepherd to this paper, provided extensive and insightful comments and suggestions, which helped us to more fully understand our patterns and to document them more effectively.

8. REFERENCES

[1] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Adison Wesley, 2003.

[2] B. Berenbach, D. Gruseman, and J. Cleland-Huang. Application of just in time tracing to regulatory codes. In *Proceedings of the Conference on Systems Engineering Research*, 2010.

[3] G. Booch. Draw me a picture. *IEEE Software*, 28:6–7, 2011.

[4] J. Bosch. Software architecture: The next step. In *EWSA*, pages 194–199, 2004.

[5] T. Breaux and A. Antón. Analyzing regulatory rules for privacy and security requirements. *IEEE Trans. Softw. Eng.*, 34(1):5–20, 2008.

[6] J. Cleland-Huang, B. Berenbach, S. Clark, R. Settimi, and E. Romanova. Best practices for automated traceability. *Computer*, 40(6):27–35, 2007.

[7] J. Cleland-Huang, C. K. Chang, and M. J. Christensen. Event-based traceability for managing evolutionary change. *IEEE Trans. Software Eng.*, 29(9):796–810, 2003.

[8] J. Cleland-Huang, M. Czauderna, Adam fand Gibiec, and J. Emenecker. A machine learning approach for tracing regulatory codes to product specific requirements. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 155–164, New York, NY, USA, 2010. ACM.

[9] J. Cleland-Huang, W. Marrero, and B. Berenbach. Goal-centric traceability: Using virtual plumblines to maintain critical systemic qualities. *IEEE Trans. Software Eng.*, 34(5):685–699, 2008.

[10] J. Cleland-Huang, R. Settimi, O. B. Khadra, E. Berezhanskaya, and S. Christina. Goal-centric traceability for managing non-functional requirements. In *ICSE*, pages 362–371, 2005.

[11] J. Cleland-Huang, G. Zemont, and W. Lukasik. A heterogeneous solution for improving the return on investment of requirements traceability. In *RE*, pages

230–239, 2004.

[12] R. C. de Boer, P. Lago, A. Telea, and H. van Vliet. Ontology-driven visualization of architectural design decisions. In *WICSA/ECSA*, pages 51–60, 2009.

[13] D.E.Perry and A.L.Wolf. Foundations for the study of software architecture. *SIGSOFT Software Eng. Notes*, 17(4):40–52, 1992.

[14] J. Dick. Design traceability. *IEEE Software*, 22(6):14–16, 2005.

[15] A. Egyed, S. Biffl, M. Heindl, and P. Grünbacher. Determining the cost-quality trade-off for automated software traceability. In *ASE*, pages 360–363, 2005.

[16] S. Eick, T. Graves, A. Karr, J. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *Software Engineering, IEEE Transactions on*, 27(1):1 –12, jan 2001.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[18] O. Gotel and A. Finkelstein. Contribution structures (requirements artifacts). In *RE*, pages 100–107, 1995.

[19] R. Hanmer. *Patterns for Fault Tolerant Software*. Wiley Series in Software Design Patterns, 2007.

[20] J. Hart, E. King, P. Miotto, and S. Lim. *Orion GN&C Architecture for Increased Spacecraft Automation and Autonomy Capabilities*, August 2008.

[21] C. Ingram and S. Riddle. *Cost-benefits of Traceability.* In: Andrea Zisman, Jane Cleland-Huang and Olly Gotel. Software and Systems Traceability., Springer-Verlag., 2011.

[22] C. Izurieta and J. M. Bieman. How software designs decay: A pilot study of pattern evolution. In *ESEM*, pages 449–451, 2007.

[23] A. Z. Jane Cleland-Huang, Olly Gotel. *Software and Systems Traceability*. Springer Verlag, 2011.

[24] A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*, pages 109–120, Washington, DC, USA, 2005. IEEE Computer Society.

[25] A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *WICSA*, pages 109–120, 2005.

[26] W. Jirapanthong and A. Zisman. Xtraque: traceability for product line systems. *Software and System Modeling*, 8(1):117–144, 2009.

[27] P. Kruchten. The 4+1 view model of architecture. *IEEE Softw.*, 12:42–50, November 1995.

[28] P. Kruchten. An ontology of architectural design decisions, 2004.

[29] P. Kruchten, P. Lago, and H. van Vliet. Building up and reasoning about architectural knowledge. In *QoSA*, pages 43–58, 2006.

[30] L. Lee and P. Kruchten. Visualizing software architectural design decisions. In *ECSA*, pages 359–362, 2008.

[31] M. Mirakhorli and J. Cleland-Huang. A decision-centric approach for tracing reliability concerns in embedded software systems. In *Proceedings of the Workshop on Embedded Software Reliability (ESR), held at ISSRE10*, November 2010.

[32] M. Mirakhorli and J. Cleland-Huang. *Tracing Non-Functional Requirements.* In: Andrea Zisman, Jane Cleland-Huang and Olly Gotel. Software and Systems Traceability., Springer-Verlag., 2011.

[33] M. Mirakhorli and J. Cleland-Huang. Transforming trace information in architectural documents into re-usable and effective traceability links. In *Proceedings of the Sixth Workshop on SHAring and Reusing architectural Knowledge*, May 2011.

[34] M. Mirakhorli and J. Cleland-Huang. Tracing architectural concerns in high assurance systems (NIER track). In *Proceedings of the 33th International Conference on Software Engineering, New Ideas and Emerging Results Track, ICSE*, 2011.

[35] M. Mirakorli and J. Cleland-Huang. Using tactic traceability information models to reduce the risk of architectural degradation during system maintenance. In *ICSM*, 2011.

[36] B. Ramesh and M. Jarke. Toward reference models for requirements traceability. *IEEE Trans. Softw. Eng.*, 27:58–93, January 2001.

[37] S. Tamblyn, H. Hinkel, and D. Saley. *Crew Exploration Vehicle (CEV) Reference Guidance, Navigation, and Control (GN&C) Architecture*, February 2007.

[38] J. van Gurp, S. Brinkkemper, and J. Bosch. Design preservation over subsequent releases of a software product: a case study of baan erp: Practice articles. *J. Softw. Maint. Evol.*, 17:277–306, July 2005.