Using Tactic Traceability Information Models to Reduce the Risk of Architectural Degradation during System Maintenance

Mehdi Mirakhorli and Jane Cleland-Huang
School of Computing
DePaul University
Chicago, IL, 60604
m.mirakhorli@acm.org; jhuang@cs.depaul.edu

***Abstract*—The software architectures of safety and mission-critical systems are designed to satisfy and balance an exacting set of quality concerns describing characteristics such as performance, reliability, and safety. Unfortunately, practice has shown that long-term maintenance activities can erode these architectural qualities. In this paper we present a novel solution for preserving architectural qualities through the use of Tactic Traceability Information Models (tTIMs). A tTIM provides a reusable infrastructure of traceability links focused around a commonly implemented architectural tactic, as well as a set of mapping points for tracing the tactic into the architectural design and the implemented code. The use of tTIMs significantly reduces the effort needed to create and maintain traceability links, provides support for visualizing the rationale behind various architectural components, and delivers timely information to maintainers so that they can preserve critical architectural qualities while implementing modifications. Our approach is described and evaluated within the context of a mission-critical software-intensive system.**

***Keywords*-Software architecture; tactics; traceability**

I. Introduction

Complex software systems, especially safety, and mission-critical ones, are designed to satisfy a number of competing quality goals. Such goals typically describe concerns related to reliability, dependability, safety, security, performance, usability, and other important qualities [1]. Designing a system to satisfy such concerns, involves devising and comparing alternate solutions, understanding their trade-offs, and ultimately making a series of interrelated design decisions with the intention of optimizing the degree to which each of the quality concerns is satisfied. Design decisions are often based on well-known architectural *tactics*, defined as re-usable techniques for achieving specific quality concerns. For this reason, instead of viewing a software architecture in terms of its physical structure of components and connectors, it is helpful to view it as a set of interrelated architectural design decisions [2] which work together to shape the structure, behavior, properties, processes, and governance of the delivered solution [3]. From this perspective, the quality of the system is therefore dependent upon the quality of the decisions by which it is shaped.

Unfortunately, despite significant efforts that go into delivering an 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 [4]. Even seemingly innocuous changes can often inadvertently lead to degradation of architectural qualities [2]. 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 [5]. 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 [6]. Tracing architectural decisions can therefore require an excessive number of traceability links.

Despite these challenges, several researchers have developed processes and models for tracing design elements to their rationales [7] or tracing requirements to the components and subcomponents to which they have been allocated [8]. These approaches establish traceability links directly between rationales, design decisions, and elements in the architectural design. Although this is conceptually simple, in practice it can result in a large number of traceability links, which are difficult and arduous to create and maintain [8]. Furthermore, it is not always obvious where to create traceability links in order balance the costs versus benefits of tracing architectural concerns.

To address these problems we present a set of Tactic Traceability Information Models (tTIMs), designed to help architects establish strategic traceability links which can be used during the maintenance phase to visualize underlying architectural tactics and tradeoffs, and to keep maintainers fully informed of underlying decisions and rationales. In earlier work [9], [10], [11] we proposed the idea of using tTIMs to trace design decisions to architectural components, and presented initial ideas for modeling tTIMs. In this paper we present an augmented set of tTIMs which explicitly

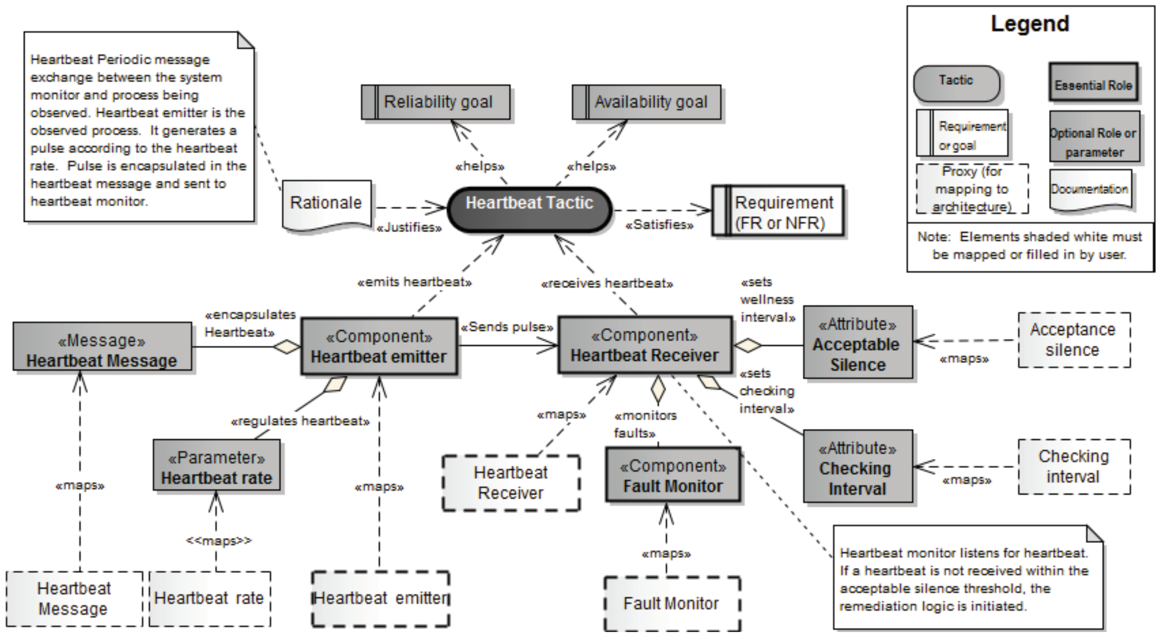

Legend
Tactic
Essential Role
Requirement or goal
Optional Role or parameter
Proxy (for mapping to architecture)
Documentation
Note: Elements shaded white must be mapped or filled in by user.
Heartbeat Periodic message exchange between the system monitor and process being observed. Heartbeat emitter is the observed process. It generates a pulse according to the heartbeat rate. Pulse is encapsulated in the heartbeat message and sent to heartbeat monitor.
Reliability goal
Availability goal
«helps»
«helps»
Rationale
«Justifies»
Heartbeat Tactic
«Satisfies»
Requirement (FR or NFR)
«emits heartbeat»
«receives heartbeat»
«encapsulates Heartbeat»
«Message» Heartbeat Message
«Component» Heartbeat emitter
«Sends pulse»
«Component» Heartbeat Receiver
«sets wellness interval»
«Attribute» Acceptable Silence
«maps»
Acceptance silence
«sets checking interval»
«Attribute» Checking Interval
«maps»
Checking interval
«regulates heartbeat»
«Parameter» Heartbeat rate
«maps»
«maps»
<<maps>>
«maps»
Heartbeat Receiver
«monitors faults»
«Component» Fault Monitor
«maps»
Heartbeat Message
Heartbeat rate
Heartbeat emitter
Fault Monitor
Heartbeat monitor listens for heartbeat. If a heartbeat is not received within the acceptable silence threshold, the remediation logic is initiated.

Figure 1. Tactic Traceability Information Model for Heartbeat Tactic

differentiate between reusable traceability links i.e. those links internal to the tactic, which can be used across multiple projects, versus project specific traceability links established as mappings from concrete elements of the architectural design to components of the tTIM. We also describe the role of the tTIM in the software maintenance effort, and evaluate the efficacy of our approach based on a case study of a Lunar Robot.

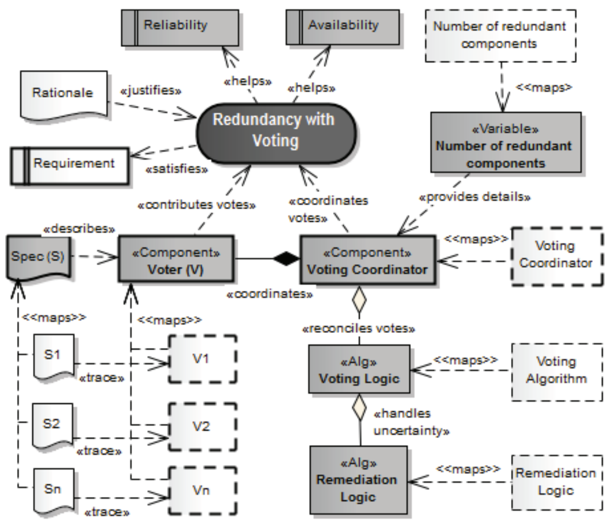
The remainder of this paper is laid out as follows. Section II describes the concept of a tTIM in more detail. Section III explains how tTIMs help to preserve architectural qualities during maintenance activities. Section IV introduces a case study of a Lunar-Robot which is then used to explain and validate our approach. Section V reports on two experiments we conducted to evaluate whether the use of tTIMs reduces the number of traceability links and facilitates software maintenance. Finally, sections VI and VII discuss related work and propose directions for future research.

II. Tactic Traceability Information Models

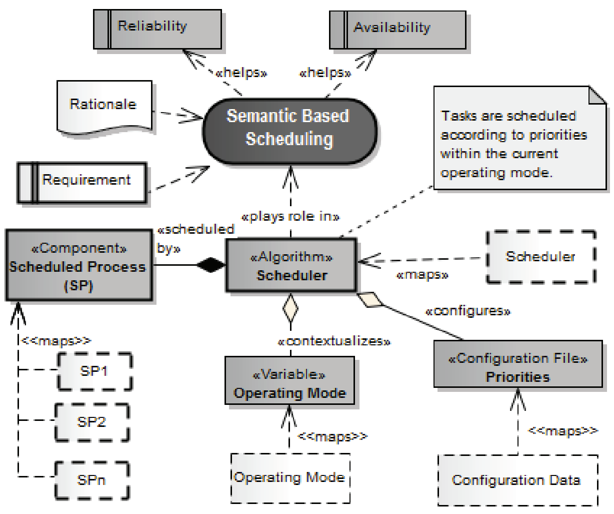
A tTIM provides the infrastructure for tracing an architectural tactic into the design. It is most useful when building families of systems which implement similar tactics, or when using very common tactics that tend to be implemented in similar ways across different products. Each of the tTIMs described in this paper were identified through observing the use of architectural tactics across the specifications of several high-assurance software systems including the Airbus A320/330/340 family, Boeing 777, Boeing 7J7, NASA robots, and the Google Chromium OS [10].

Each tTIM is centered around a specific tactic and defines backward traceability to the tactic's driving requirements and rationales, and forward traceability to the architectural elements in which it is realized. Backward traces include links to goals and rationales associated with the tactic, however if necessary, these can be customized for a specific project. The tTIM also defines the primary roles and parameters of the tactic, relationships between the roles, and proxy elements which are used to map architectural elements in design models and code to the tTIM. Traceability links between tactics and architecture are therefore established as mappings.

Figure 1 depicts the tTIM associated with the *heartbeat* tactic. This tactic is commonly used to monitor the availability of a critical process in order to increase qualities such as *reliability* and *availability*. The *heartbeat* tactic includes the *receiver*, *emitter*, and *fault monitor* roles, and additional supporting roles of *heart beat rate*, *heart beat message*, *checking interval*, and *acceptable silence threshold*. *Heart beat rate* establishes the frequency of the heart beat, *heart beat message* represents the data structure in which the heart beat is transmitted, *checking interval* represents the frequency with which the receiver checks for the heartbeat, and finally the *acceptable silence threshold* establishes the maximum amount of time that can elapse between observed heartbeats before a fault is generated. Each of these roles and parameters are modeled in the tTIM along with their relevant proxies connected via ≪maps≫ links and used for creating traceability links to the architecture. The tTIM also models structural relationships between roles such as the *pulse* sent

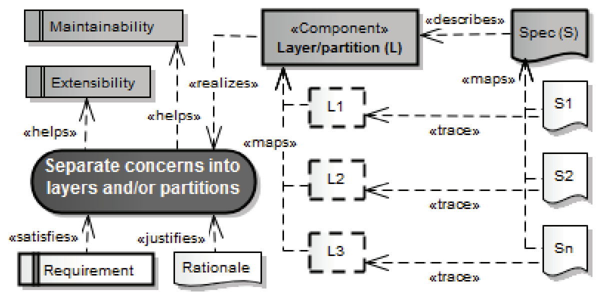


(a) Redundancy with Voting



Tasks are scheduled according to priorities within the current operating mode.

(b) Semantic Based Scheduling

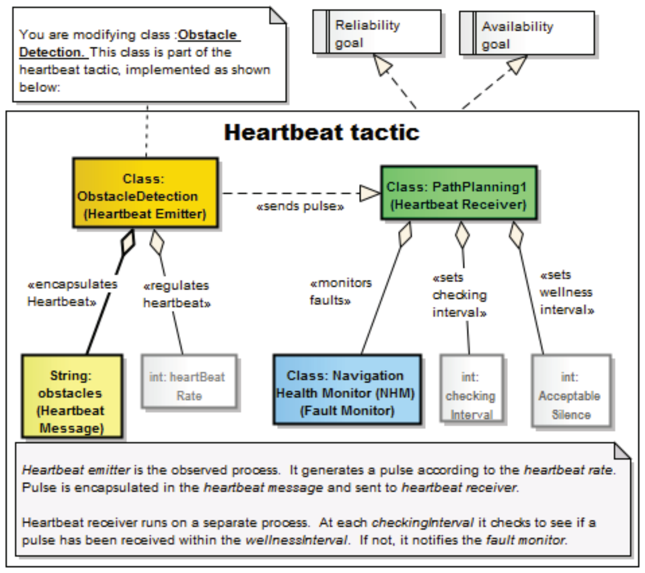


(c) Partitioning/Layers

Figure 2. Tactic Traceability Information Models for three additional Tactics

from the *heart beat emitter* to the *heartbeat receiver*.

Figure 2 provides examples of three additional tTIMs representing *redundancy with voting*, *semantic based scheduling*, and *layers and partitions*. Each of these tTIMs includes quality goals, rationales, requirements, and roles, as well as proxies for mapping the tTIM to architectural elements. As previously stated in this paper, each of these tTIMs was derived from an extensive study of tactics used in several mission critical systems. However, the contribution of this paper, is not in providing a fully validated set of tTIMs, but rather in showing how a tTIM can be used to support effective software maintenance while reducing the likelihood of architectural degradation.


You are modifying class :**Obstacle Detection.** This class is part of the heartbeat tactic, implemented as shown below:

Heartbeat emitter is the observed process. It generates a pulse according to the *heartbeat rate*. Pulse is encapsulated in the *heartbeat message* and sent to *heartbeat receiver*.

Heartbeat receiver runs on a separate process. At each *checkingInterval* it checks to see if a pulse has been received within the *wellnessInterval*. If not, it notifies the *fault monitor*.

Figure 3. Design rationale displayed to user when they modify the heartbeat emitter component

III. Preserving Architectural Quality

tTIMs are designed to minimize the potential for architectural degradation during the software maintenance process. This is accomplished through the two tasks of link creation and link usage. Link creation involves identifying all of the tactics used in the architectural solution, instantiating a tTIM for each identified tactic, and then mapping proxies in the tTIM to the relevant elements in the architecture and code. Although the established tTIMs can be used to support manual inspections of the architecture, greater value can be realized through instrumenting an IDE to automate the use of the traceability links. An automated tool should provide support for registering architectural elements which are mapped to tTIM proxies, monitoring those elements for maintenance activities such as move, delete, duplicate, modify [12], and finally utilizing the infrastructure of the tTIM to generate timely notifications to software maintainers to keep them informed of architectural decisions related to any components they are currently modifying.

Table I
LUNAR ROBOT: A SUB-SET OF HIGH-LEVEL REQUIREMENTS

ID	Requirement	Type
1.	The Lunar-Robot (LR) shall have the capability to operate in two modes: manual and autonomous.	Functional
2.	At any time, the LR shall be either inactive, waiting, or active.	Functional
3.	When the LR is inactive, it will check periodically for activation commands.	Functional
4.	The LR shall perform the following functions: move, plan, collect, analyze, record, and transmit functions.	Functional
5.	When the LR is in manual mode it will respond to specific commands received from MCC.	Functional
6.	When the Lunar-Robot is in autonomous mode it will determine and execute the correct steps needed to perform a higher level function defined by MCC. Examples of such functions include reaching a specified location, or gathering atmospheric samples.	Functional
7.	If the Lunar-Robot is in manual mode and and receives no commands from MCC for over *maximum_outOfRange_period* it shall switch to *autonomous_mode* and return to the geographical coordinates of *last_point_of_known_contact*.	Availability
8.	The Lunar-Robot will have a probability of uncorrectable failure of 0.00001 or less during 90 days of its mission on the surface of the moon.	Reliability
9.	Commands sent from MCC to LR will be acknowledged within 3 seconds when communication is feasible.	Responsiveness
10.	Transmissions between MCC and the LR shall be secure.	Security

Table II
LUNAR ROBOT: PRIMARY ARCHITECTURAL DECISIONS

ID.	Decision
1.	The Lunar Robot control system will be replicated using active redundancy with graceful degradation. There will be replications.
2.	Each instance of the Guidance, Navigation and Control subsystems(GN&C) will run on a separate process, and will be configured uniquely.
3.	The tasks to be performed at various phases of the mission will be managed in individual configuration files by the robot executive and domain executive modules. The task sequencer reference model is used to achieve real-time task distribution and execution
4.	Task scheduling will be performed using the semantic-based scheduling algorithm. This approach takes priorities of various tasks, and the current mode of operation into consideration. It allows task priorities to be established according to the current operating mode.
5.	Each of the three domains will have a dedicated health management module which will implement two-self checking for each running thread (i.e. two separate threads will run the same task and must be in agreement). In case of a mismatch, the result which most closely matches the data produced by the simulation engine will be used.
6.	Critical modules in each of the Guidance, Navigation and Control (GN&C) domains will be redundantly coded in three different programming languages by three independent teams.
7.	Thread execution will be scheduled according to predefined priorities using preemptive scheduling.
8.	All sensors will be monitored using the heartbeat tactic.
9.	All data received from sensors will be tested for validity i.e. using CRC.
10.	A majority voting schema will be used to select the most accurate data from a set of replicated sensors.
11.	Two separate buses will be used for conveying sensor data and control data respectively. Note: This allows different scheduling strategies to be applied to each bus.

An example of such a notification is provided in Figure 3. It shows the notification message that would be displayed if a maintainer modifies an architectural component mapped to the heartbeat tactic's emitter component. Visualizing tactics in this way abstracts out the important factors of the underlying design rationale, and makes design decisions understandable to software maintainers. We have currently developed functioning prototypes of the event monitoring and design notification process in both the StarUML modeling tool and in Visual Studio.

IV. CASE STUDY: LUNAR ROBOT

To illustrate and evaluate the use of tTIMs in the maintenance process, we introduce a case study of a Lunar Robot system, compiled from an extensive set of publicly available NASA documents [13], [10]. The robot's primary mission is to autonomously traverse the lunar surface, collect sample data related to comets, dust, and celestial objects, record temperatures, perform scientific experiments, and send results back to the earth-based Mission Control Center (MCC). From a maintenance perspective, our interest is in ensuring that quality concerns, as realized through the various architectural decisions, are maintained throughout the lifetime of the Lunar-Robot software system. Such systems often live far beyond a single lunar mission, as individual components and sometimes entire architectures are often reused in future implementations. Maintenance activities therefore include initial modifications made during the early development phase, modifications made to the system after deployment (which in the case of the lunar robot often means transmitting new software after the robot has been launched into space), and finally modifications made to the system if it is re-used in a new robot system.

A. Architecture

A selection of the Lunar Robot's functional and non-functional requirements are shown in Table I. An initial analysis of these led to a series of design decisions including the ones depicted in Table II. In turn, these decisions resulted in the high-level architectural design shown in Figures 4 and 5. The Lunar-Robot architecture is structured around a *control system (CS)*, *Integrated Vehicle Health Management*

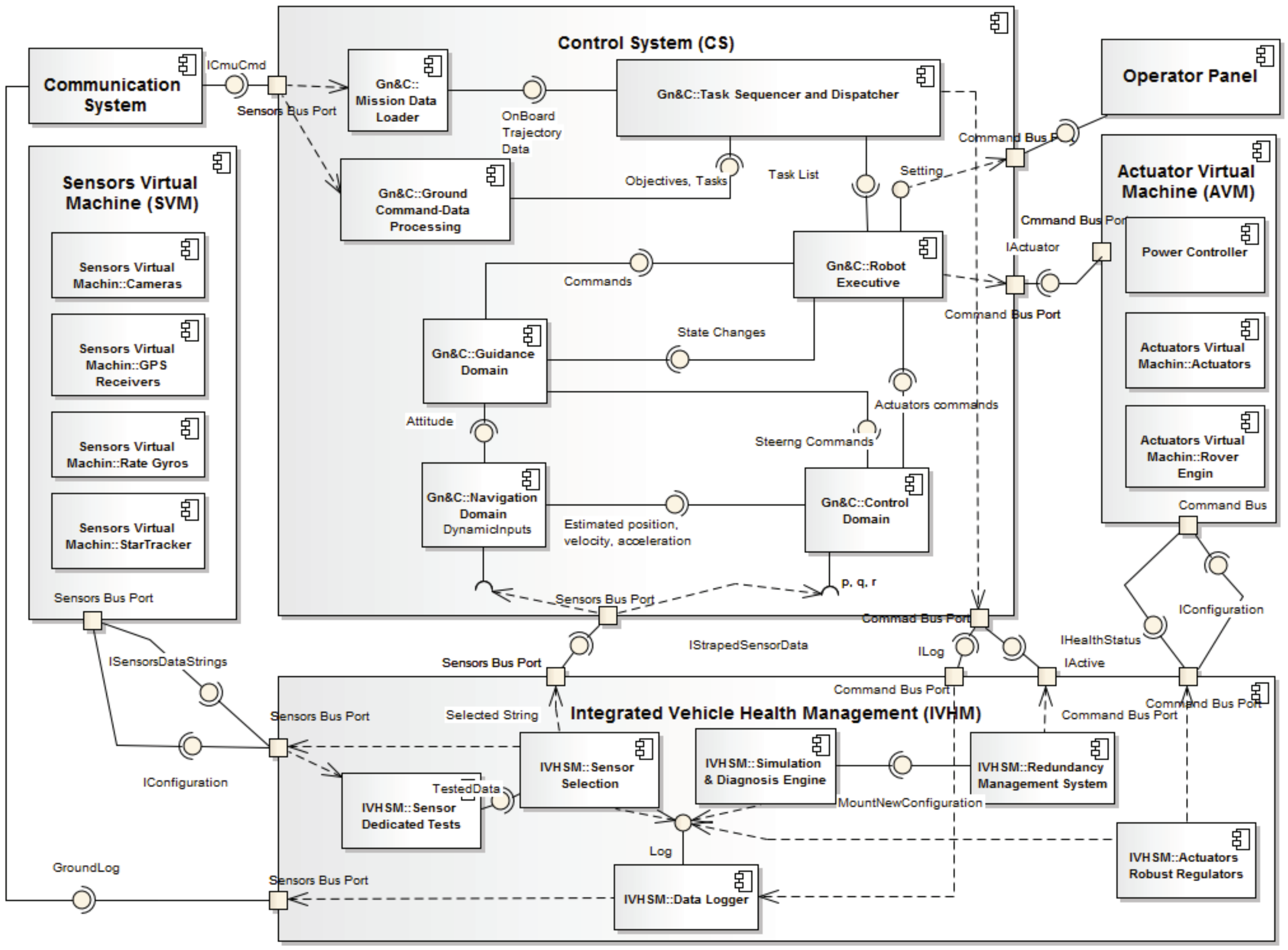


Figure 4. Lunar Robot: High Level Component and Connector View

(IVHM) system, the *Sensors Virtual Machine* and *Actuators virtual machine*, a *communication system*, and an *operator panel*. Data from the sensors is first passed through the IVHM component for correctness checking and is then forwarded to the CS. The CS uses the data to make decisions and to process high level commands. It then sends lower level commands to the actuators. One of the most interesting components is the integrated vehicle health management system (IVHM) responsible for monitoring the health of the robot, and when necessary, performing dynamic reconfigurations to maintain functionality. The Lunar Robot received inputs from cameras, GPS receivers, rate gyros, and star trackers, and issues commands to mechanical devices such as the power controller, wheels, and scientific instruments.

It is worth noting that the component and connector view of Figure 4 provides only limited visual clues concerning the architectural decisions behind the design. Certain architectural decisions are clearly visible, such as the use of partitions to separate different types of functionality. Other decisions are partially visible, such as the decision to include redundant components (given the plural descriptions for GPS receivers and Rate Gyros). However, many of the important architectural decisions listed in Table II are not visible at all, either in this diagram or in lower level diagrams. This lack of visibility is one reason that maintenance efforts often result in architectural degradation.

B. A Maintenance Scenario

To illustrate how tTIMs can address this problem, we provide a step-by-step example of a specific change request executed within the context of Sparx Enterprise Architect. It should be noted that event monitoring and basic notifications are fully functioning in our tool; however some of the GUIs depicted in this change scenario are still under development.

In this maintenance scenario, high resolution stereo cameras are added to the system to work in conjunction with existing forward looking infrared cameras and laser sensors. The new data is processed by modifying an existing algorithm in the *obstacle detection (OD)* component from the *navigation domain*. The OD component utilizes infrared cameras and other resources to identify obstacles in the planned path. It sends *obstacle* messages to two *Path Planning (PP)* components. However, an underlying architectural decision to use the heartbeat tactic to monitor the OD component is not explicitly documented, even though the design calls for the OD component to embed heartbeats into the obstacle messages sent to the *PPi* component. This means that obstacle messages must be sent regularly, regardless of whether an obstacle is detected or not.

In the change scenario, the developer starts modifying the OD component to interface with the infrared camera, and to integrate data from the camera into the obstacle detection logic. As the OD component is a monitored element (i.e. registered as playing a role in the heartbeat pattern), an 'architectural' icon is displayed on the screen. This is labeled as (1) in Figure 6 and shown as a pyramid over the OD component on the left hand side of the Enterprise Architect (EA) screen. Once an architectural event is detected, the related architectural tactics are visualized in the IDE. In our prototype we display the tactic in a separate pane on the right hand side of the screen (labeled (3)), and color-coordinate specific roles in the primary architectural view (labeled (2))with those of the tactic.

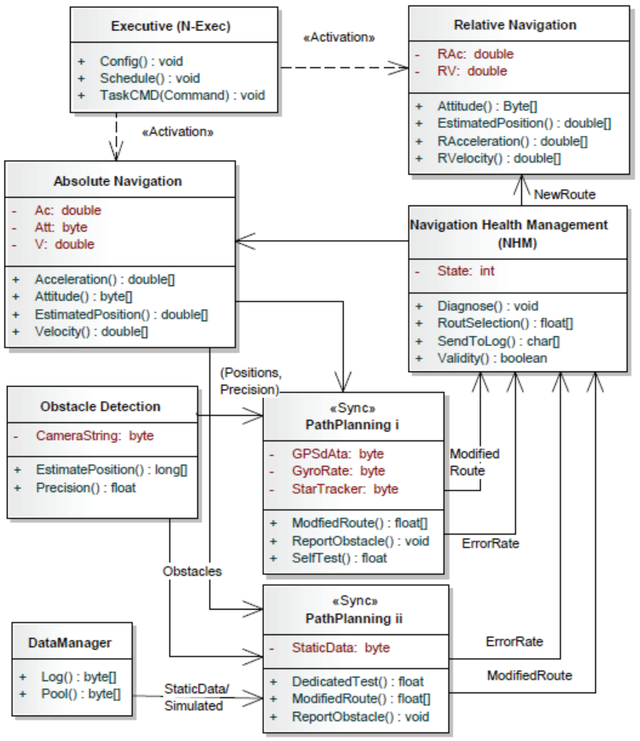
In this way, the developer modifying the OD component is informed of architectural tactics that impact the OD component. The tactical roles of the OD are clearly visible in the visualized tactic, and in this case show that the OD plays the role of a *heartbeat emitter*, and that the heartbeat is packaged into the *obstacle* message sent to the *PPi* component. Although the *PPii* component also receives the *obstacle* message, it does not monitor the heartbeat and is therefore not involved in the tactic. This example illustrates how tTIMs are used during the maintenance process.

V. Evaluation

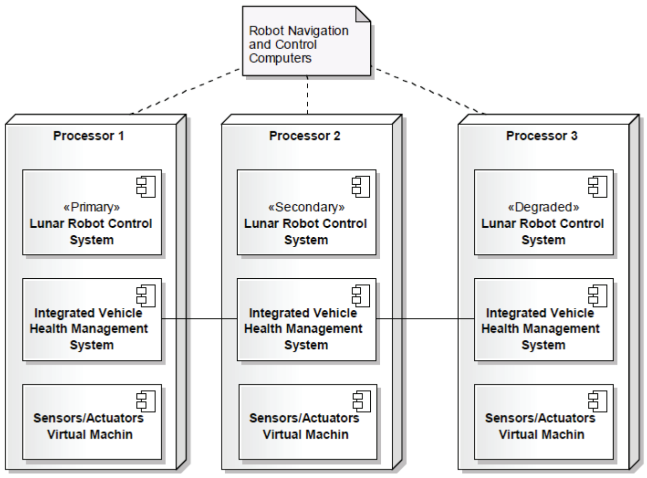
We conducted two experiments to investigate the use of tTIMs in the software maintenance process. The first evaluated our claim that using tTIMs significantly reduced the cost and effort of establishing and maintaining traceability links, while the second evaluated the usefulness of tTIMs to help preserve architectural qualities during a series of simulated maintenance activities in the Lunar Robot.

A. Experiment #1: Reducing Cost and Effort

The first experiment evaluated the number of traceability links needed to trace tactic-related design decisions in the Lunar-Robot both with and without the use of tTIMs. It did not include traces for functional requirements, as these generally have less impact on architectural quality, and are also unaffected by the use of tTIMs. A subset of the traces created by mapping architectural elements to tTIM proxies are depicted in Table III.



(a) Composite Structure Navigation Domain


Robot Navigation and Control Computers
Processor 1
«Primary» Lunar Robot Control System
Integrated Vehicle Health Management System
Sensors/Actuators Virtual Machin
Processor 2
«Secondary» Lunar Robot Control System
Integrated Vehicle Health Management System
Sensors/Actuators Virtual Machin
Processor 3
«Degraded» Lunar Robot Control System
Integrated Vehicle Health Management System
Sensors/Actuators Virtual Machin

(b) Deployment View

Figure 5. Additional Architectural Views for the Lunar Robot

To evaluate the effectiveness of using tTIMs for traceability purposes we constructed two different traceability matrices. The first established traceability by mapping architectural elements to tTIM proxies, while the second established traceability in a more traditional way without the use of tTIMs. In both cases the final traceability matrix provided forward and backwards traceability between architectural elements, tactics, rationales, goals, and requirements as recommended in Ramesh's traceability metamodels [8]. Two different granularity levels were explored which included coarse-grained traces to components and/or partitions, and also fine-grained traces to the method and/or parameter level.

Traceability links were established for the 18 different instances of tactics found in the Lunar Robot including several cases of *heartbeat*, two cases of *redundancy*, three cases of *N Self-Checking Programming*, and several additional tactics shown in Table IV. Trace links were counted for each implemented tactic and are also reported in Table IV. The table shows the number of times a specific tactic was used in the Lunar-Robot architecture, the number of links needed to support coarse grained traceability, both with and without the use of tTIMs, and finally the number of additional links needed to achieve fine-grained traceability to the method or parameter level using the tTIM. Fine-grained links were only counted for scenarios which used the tTIM, as our experience suggests such traces would be unlikely without the guidance of the tTIM.

These results highlight some interesting trends. In all cases, the use of tTIMs reduced the number of traceability links that needed to be created and maintained for an individual project. Each tactic traced using a tTIM required an average of 5.4 links, versus 9.28 for tactics traced without tTIMs. On the other hand, our results also showed that the sum of reusable links for a tactic plus the project-specific links can sometimes be greater than the number of links created without benefit of a tTIM. This observation highlights the need to use tTIMs strategically across multiple projects, as opposed to creating them individually on a project-by-project basis. The additional links occur because the tTIM provides a larger and richer set of semantically typed traceability links that are not included in Ramesh's metamodel [8], and which are unlikely to be created without the benefit of the tTIM. In fact, this is one of the significant benefits of using a tTIM. Instead of relatively ad-hoc typing of traceability links created by multiple stakeholders, a simple mapping of an architectural element to a proxy in the tTIM provides a rich set of consistently typed traceability links.

The results show that the amount of additional effort needed to achieve fine-grained traceability is often very minimal given the existing infrastructure of each tTIM. We should also point out that we assume that equal effort is required to establish a traceability link using a tTIM versus creating a link without the tTIM. However, our informal observations have shown that the tTIM simplifies the traceability task and therefore reduces the cost and effort of creating a traceability link. Although we leave a more rigorous evaluation of this phenomenon to a later user study, it implies that simply counting the number of traceability links does not fully represent the cost and effort savings realized through using a tTIM.

In summary, this experiment demonstrated that for our Lunar-Robot case study, the use of tTIMs significantly reduced the number of project-specific traceability links. We leave a discussion of the overall costs and benefits of using tTIMs to the concluding section of this paper.

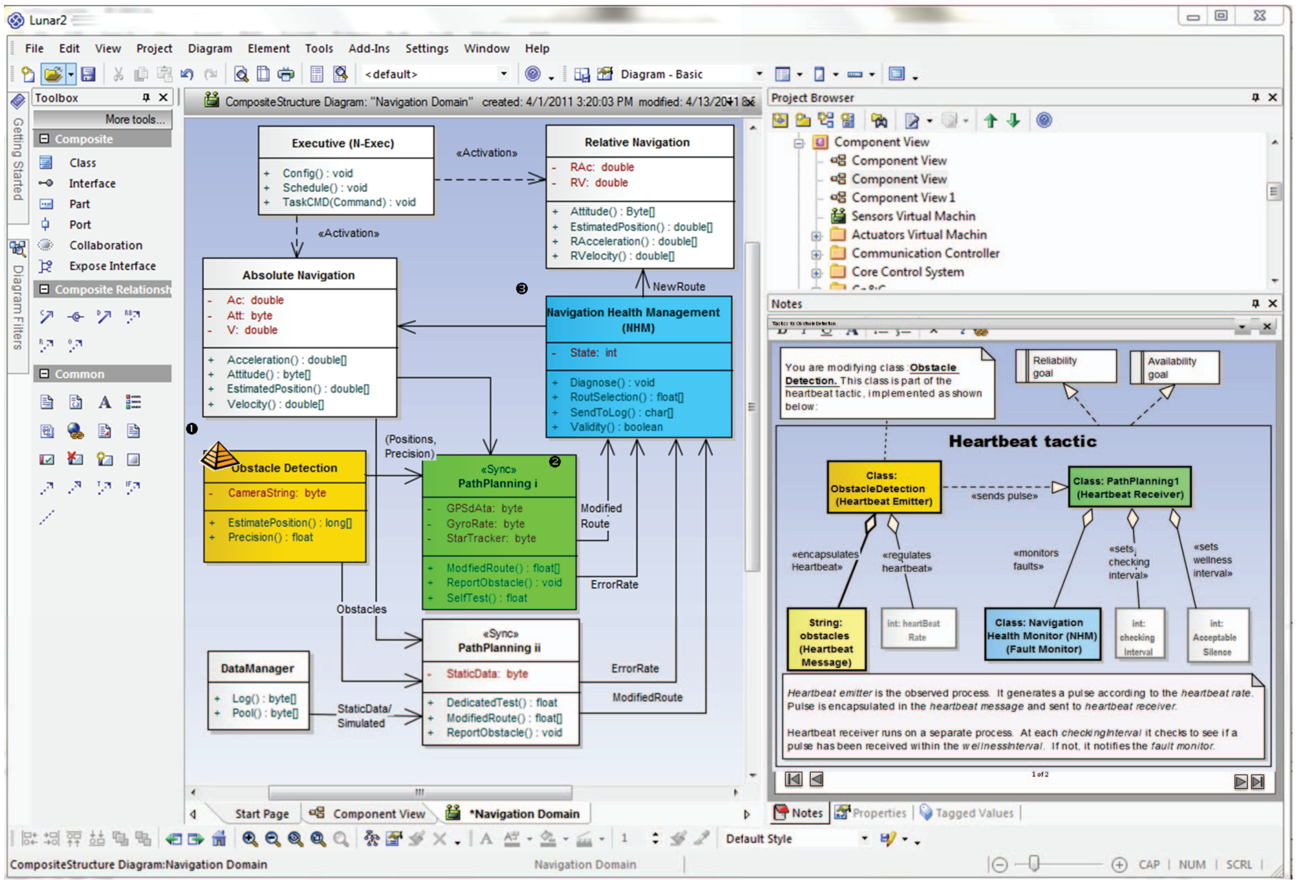
Table III
SUBSET OF TACTIC TRACES FOR LUNAR ROBOT

ID	tTIM	Role	Mapping
T1	Heartbeat including piggy backing and CRC	Emitter	SensorVM:Lidar Driver
		Monitor	IVHM: Sensor Selection
T2	Heartbeat including piggy backing and CRC	Emitter	SensorVM:GPS Receiver
		Monitor	IVHM: Sensor Selection
T3	Layers and partitions	Partition	Navigation Domain
		Partition	Guidance Domain
		Partition	Control Domain
T4	Active redundancy with degradation, deployed separately	Coordinator	Redundancy Management System
		Replica	Lunar Robot CS (Primary)
		Replica	Lunar Robot CS (Secondary)
		Replica	Lunar Robot CS (Degraded)
T5	N Self-checking programing with acceptance tests	Component & test	Navigation Domain: PathPlanning, PathTest1
		Component & test	Navigation Domain: PathPlanning2, Pathtest2
		Coordinator	Navigation Domain: Navigation health management

B. Evaluation #2: Maintenance Scenarios

The second experiment was designed to evaluate whether tTIMs provided adequate support for the goal of keeping developers informed of underlying architectural decisions during the maintenance process. We therefore developed 18 different maintenance scenarios based on a software architecture change characterization scheme defined by Williams and Carver [14]. The eighteen scenarios are depicted in Table V. Each one was executed using our prototype tool which first captured the change event, and then generated appropriate information displays. Table V also depicts the category of change covered by each scenario, the number of expected notification messages (as determined through a manual analysis), the number of successfully generated notifications (true positives), the number of unnecessary notifications (false positives), the number of missed notifications (false negatives), and the number of correctly ignored maintenance tasks (true negatives).

Nine of the scenarios affected architectural decisions, and our prototype tool correctly recognized these potential impacts and generated appropriate notifications. The remaining nine scenarios represented either micro-changes or functional changes that did not affect architectural decisions. Of these, our tool correctly filtered out three scenarios, but generated notifications for the remaining six. These six notifications represented false positives, meaning that tactic information was displayed, even though the actual change did not impact the tactic. Clearly, displaying unnecessary information may desensitize the developer to the importance



You are modifying class : **Obstacle Detection.** This class is part of the heartbeat tactic, implemented as shown below:

Heartbeat emitter is the observed process. It generates a pulse according to the *heartbeat rate*. Pulse is encapsulated in the *heartbeat message* and sent to *heartbeat receiver*.

Heartbeat receiver runs on a separate process. At each *checkingInterval* it checks to see if a pulse has been received within the *wellnessInterval*. If not, it notifies the *fault monitor*.

Figure 6. Visualizing architectural tactics within Enterprise Architect

of such messages; however it should be noted that changes made to the large percentage of classes and components that are entirely unrelated to any tactic will never result in architectural notifications. Furthermore, false positives do not provide unrelated information, but do in fact provide information related to the general context of the change. Nevertheless, future work in this area will focus on minimizing false-positives through capturing and interpreting additional information about the current maintenance task.

C. Threats to Validity

Evaluating a new process is always challenging. Ideally such an evaluation would be conducted within the context of a real project over a period of time as the project progresses from early design decisions to deployment and finally on into the maintenance phase. As this was not feasible in the initial phases of our research, we evaluated the approach through the use of a case study and two quantitative experiments.

The case study of the Lunar Robot created a realistic environment for evaluating the utility of tTIMs for preserving architectural qualities, as its architectural decisions are typical of those found in other safety critical software systems. Counting the number of traceability links represents a commonly adopted technique for estimating traceability effort, although given our observations that less effort is required to create traceability links using a tTIM, this approach likely underestimates the realized savings. Furthermore, the maintenance scenarios were constructed using a standard framework, and were therefore representative of a broad spectrum of maintenance tasks. The results from this study therefore suggest that our findings will be applicable across a broad range of systems which utilize some of the architectural tactics we have specified as tTIMs. The primary threat to validity of our results is that our approach was studied only in a controlled environment, and has not yet been used within the context of a real project. This will be the focus of future work.

VI. Related Work

Lee and Kruchten explored four different techniques for visualizing architectural design decisions. These included a simple tabular view, a structural view of the decision structure, a chronological view showing decisions over time, and a view which visualizes the impact various decisions have on each other [15]. The structural view, which is most similar to

Table IV
TRACE LINK COUNTS PER TACTIC IN THE LUNAR ROBOT

	Tactic	Number of occurrences of tactic	Coarse Grained Trace Links					Additional Fine-Grained Links	
			without tTIM		with tTIM			with tTIM	
			per tactic	Total	per tactic	Total	re-used from tTIM	Per Tactic	Total
1.	Heartbeat, Piggy backing and CRC check	6	9	54	4	24	6	4	24
2.	Redundancy with voting	1	7	7	3	3	6	6	6
3.	Active redundancy with degradation	1	17	17	9	9	6	5	5
4.	Multi Threading	1	12	12	9	9	4	1	1
5.	Separate processes with configuration files	1	19	19	9	9	5	2	2
6.	Layers (1)	1	9	9	7	7	4	2	2
7.	Layers (2)	1	11	11	9	9	Shared with Layers (1)	2	2
8.	Transaction	2	5	10	2	4	4	0	0
9.	N Self-Checking Programming w. Acceptance Tests	3	7	21	6	18	6	3	9
10.	N-Version Programming	1	7	7	6	6	4	3	3
		TOTALS:		167		98	45		

Table V
MAINTENANCE SCENARIOS

	Change Scenario			Motivation	Impact			Features
	NG	Imp	Description		Arch	Des	Code	
1.	Y	Y	New communication channel established directly between IVHM::Logger and the Robot Executive and with the CS::Task Sequencer & Dispatcher is removed	Refactoring, Performance Enhancement	*			Data transfer: Flow of data from system to external systems
2.	N	N	IVHM::Actuators Robust Regulators module communicates with CS::Robot Executive module instead of directly with the AVM.	Refactoring, Performance Enhancement	*			
3.	Y	N	IVHM::Simulation and Diagnosis Engine retrieves information from the shared repository instead of communicating with the Logger.	Refactoring: Performance Enhancement	*		*	
4.	Y	Y	Component Data Manager is moved from Navigation Domain process to Control System process and combined with component Mission Data Loader	Architecture refactoring	*	*	*	Data access: Receipt of data from external systems/repositories
5.	Y	Y	A security enhancement changes the encryption algorithm used to send mission information to agents on the LEO.	Secure data transmission		*	*	
6.	Y	N	Communication module is changed to eliminate encryption from messages sent between the Lunar Robot and the MCC whilst on the moon.	Performance enhancement			*	
7.	Y	Y	Rover's processor is upgraded to provide greater processing power. A mistake occurs and memory is accidentally reduced.	Enhancement	*			Devices: Hardware devices used by the system
8.	Y	Y	High resolution stereo cameras added to work with existing forward looking infrared cameras and laser sensors. The new data will be processed by modifying an existing algorithm in the obstacle detector component	Hardware enhancement	*	*	*	
9.	Y	Y	Multi-threading is used instead of multi-process for all the three domain of Navigation, Guidance and Control	Architecture refactoring	*	*	*	
10.	Y	N	Communication module is modified to remove a defect	Defect removal		*	*	System interface: Software interfaces with external systems
11.	N	N	Change in Operator Panel, modification of user interface, new menu item, changes look and feel of dialog	UI enhancement		*	*	human computer interaction interfaces
12.	Y	N	Change in protocol for communication with Mission Control Center (MCC) at the earth	Adaptive enhancement	*		*	Communication: Interfaces to other systems/data
13.	Y	Y	Mission data and operations are logged following completion of a task instead of following each simple action	Performance Enhancement			*	Computation:algorithm functions and modification of data
14.	N	N	Component Relative Navigation is modified to fix bugs	Defect removal			*	
15.	Y	Y	Change in functionality in a performance critical module (PPOD)	Addition of new functionality	*	*	*	
16.	Y	Y	Obstacle Detection component is merged into both Path Planning threads	Corrective maintenance/self checking	*	*	*	
17.	Y	N	Data stored by the IVHM::Logger in the Robot's blackbox, is modified to be stored in encrypted format	Security			*	I/O format of information processed by system
18.	Y	N	IVHM Simulation and Diagnosis Engine is modified to decode data stored in encrypted format in the blackbox	Security	*	*	*	

*'NG' = Notification generated due to potential impact of the change; 'Imp' = a true risk of affecting an underlying architectural decision. Hence (Y,Y)= True positives, (N,N) = True negatives, (N,Y)=False negatives, and (Y,N) = False positives.

our approach, makes design decisions visible to the architect without cluttering the architecture. Decisions are represented visually as graphs depicting the interrelationships between architectural decisions and business goals. However, while this approach is a natural representation for design decisions and their interdependencies such graphs can quickly become cluttered and difficult to understand. Boer et al. therefore visualize design decisions in a multi-tabular format instead of a graph [16]. However, like Lee and Kruchten's approach, they focus on visualizing design decisions and tradeoffs, and do not address the problem of tracing decisions to concrete architectural elements. This creates a disconnect between architecture and design decisions, that our approach is intended to address. Jansen et al. developed a tool named Archium for capturing architectural decisions and integrating them into the development process [17]. Archium provides support for various scenarios such as checking that requirements are addressed in the design, and identifying architectural components related to specific design decisions. Unlike our approach, Archium is an integrated development environment, which means that the approach is not generalizable outside the Archium environment.

There are also several techniques for capturing architectural knowledge [18], [19]. However, from a traceability perspective, these approaches focus primarily on capturing architectural knowledge but fail to provide guidance for creating and managing the potentially large number of related traceability links. Furthermore, while many of the prescribed techniques capture architectural knowledge in a seemingly beneficial and systematic way, they provide little support for actually utilizing this knowledge during software maintenance.

VII. Conclusions

This paper has proposed a novel, yet simple approach for tracing architectural tactics through the use of tTIMs. As our results have indicated, tTIMs reduce the cost and effort of traceability through providing a set of re-usable traceability links. However, upfront effort is required to create the tTIMs. Clearly our approach is constrained by the extent to which similar tactics are reused across projects. As an integral part of our ongoing work we are engaging in an extensive study of the use and implementation of tactics in open source and safety critical projects. However, even without this study, it is clear that organizations which release families of related products do in fact implement similar tactics across different applications [20]. Furthermore, the preliminary observations we have already made of tactics in safety-critical systems also support this claim.

In ongoing work we will extend our existing prototypes and conduct a longer-term study to refine the tTIMs and the underlying traceability techniques in order to more rigorously evaluate the use of tTIMs within the context of a variety of software development projects. Furthermore we will investigate the feasibility of reducing false positive notifications by making architectural notifications more context-specific.

VIII. Acknowledgments

Partial funding for this work was provided by the National Science Foundation grant #CCF-0810924.

References

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

[2] D.E.Perry and A.L.Wolf, "Foundations for the study of software architecture," *SIGSOFT Software Eng. Notes*, vol. 17, no. 4, pp. 40–52, 1992.

[3] P. Kruchten, "An ontology of architectural design decisions," pp. 55–62, 2004.

[4] 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.*, vol. 17, pp. 277–306, July 2005. [Online]. Available: http://portal.acm.org/citation.cfm?id=1077863.1077864

[5] G. Booch, "Draw me a picture," *IEEE Software*, vol. 28, pp. 6–7, 2011.

[6] A. Jansen and J. Bosch, "Software architecture as a set of architectural design decisions," in *WICSA*, 2005, pp. 109–120.

[7] P. Kruchten, R. Capilla, and J. C. Dueas, "The decision view's role in software architecture practice," *IEEE Software*, vol. 26, no. 2, pp. 36–42, 2009.

[8] B. Ramesh and M. Jarke, "Toward reference models for requirements traceability," *IEEE Trans. Softw. Eng.*, vol. 27, pp. 58–93, January 2001. [Online]. Available: http://portal.acm.org/citation.cfm?id=359555.359578

[9] M. Mirakhorli and J. Cleland-Huang, "Tracing architectural concerns in high assurance systems (NIER track)," in *Proc. of the 33rd International Conf. on Software Engineering, New Ideas and Emerging Results Track, ICSE*, 2011.

[10] 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.

[11] ——, "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.

[12] J. Cleland-Huang, C. K. Chang, and M. J. Christensen, "Event-based traceability for managing evolutionary change," *IEEE Trans. Software Eng.*, vol. 29, no. 9, pp. 796–810, 2003.

[13] NASA's and Robots, *Online at: http://prime.jsc.nasa.gov/ROV/nlinks.html*, 2008.

[14] B. J. Williams and J. C. Carver, "Characterizing software architecture changes: A systematic review," *Information & Software Technology*, vol. 52, no. 1, pp. 31–51, 2010.

[15] L. Lee and P. Kruchten, "A tool to visualize architectural design decisions," in *QoSA*, 2008, pp. 43–54.

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

[17] A. Jansen, J. S. van der Ven, P. Avgeriou, and D. K. Hammer, "Tool support for architectural decisions," in *WICSA*, 2007, p. 4.

[18] R. Capilla, F. Nava, S. Pérez, and J. C. Dueñas, "A web-based tool for managing architectural design decisions," *SIGSOFT Softw. Eng. Notes*, vol. 31, Sept. 2006. [Online]. Available: http://doi.acm.org/10.1145/1163514.1178644

[19] A. Tang, Y. Jin, and J. Han, "A rationale-based architecture model for design traceability and reasoning," *Journal of Systems and Software*, vol. 80, no. 6, pp. 918 – 934, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/B6V0N-4M6SB7T-1/2/82ea7eabc2c4947aee2168fd536cc9f3

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