Architectural Dependency Analysis to Understand Rework Costs for Safety-Critical Systems

Robert L. Nord, Ipek Ozkaya
Software Engineering Institute[1]
Carnegie Mellon University
Pittsburgh, PA, USA
{rn, ozkaya}@sei.cmu.edu

Raghvinder S. Sangwan[1, 2]
School of Graduate Professional Studies[2]
Pennsylvania State University
Malvern, PA, USA
rsangwan@psu.edu

Ronald J. Koontz
The Boeing Company
Mesa, AZ, USA
ron.j.koontz@boeing.com

ABSTRACT
To minimize testing and technology upgrade costs for safety-critical systems, a thorough understanding and analysis of architectural dependencies is essential. Unmanaged dependencies create cost overruns and degraded qualities in systems. Architecture dependency analysis in practice, however, is typically performed in retrospect using code structures, the runtime image of a system, or both. Retrospective analysis can miss important dependencies that surface earlier in the life cycle. Development artifacts such as the software architecture description and the software requirements specification can augment the analysis process; however, the quality, consistency, and content of these artifacts vary widely. In this paper, we apply a commonly used dependency analysis metric, stability, and a visualization technique, the dependency structure matrix, to an architecture common to safety-critical systems that was re-engineered to reduce safety testing and upgrade cost. We describe the gaps observed when running the analysis and discuss the need for early life-cycle dependency analysis for managing rework costs in industrial software development environments.

Categories and Subject Descriptors
D.2.8 [Software Engineering]: Metrics – complexity measures; D.2.9 [Software Engineering]: Management; D.2.11 [Software Engineering]: Software Architectures

General Terms
Management, Measurement, Design

Keywords
Architecture views, architecture analysis, dependency analysis, DSM, stability, testing, evolution.

1. INTRODUCTION
In this paper, we analyze the software architecture of a generalized, industrial, safety-critical system that was re-engineered to reduce rework cost associated with safety testing and technology upgrade. By walking through the motivations and the engineering decisions of the enhancement of the architecture, we demonstrate the kinds of architecture dependencies that cannot be detected with today's tools and techniques but that are necessary to capture the value of the effort. Software engineers have been inundated with promising tools in architecture, conformance, and quality analysis, especially to manage software evolution and maintenance [31]. While this is a much anticipated positive development, almost all developments in this area have been variations on providing static analysis metrics at the code and module views of the architecture.

These tools often rely on dependency analysis, an assessment of the degree to and the nature of which a software element syntactically relies on another element (a module is a part of another module, a module is a refinement of another module, a module uses another module, etc.). There is little evidence of how this analysis helps improve decision making when evolving a system, especially when these decisions involve semantic transformations (for instance, understanding the impact of splitting a module to reduce cost related to safety-critical testing or the impact of using a module as a standard interface intermediary that provides a generalized publish-subscribe capability between modules).

Our previous work built on tools and analysis techniques, such as dependency structure matrices (DSM), to experiment with metrics to better estimate and understand the impact of rework [19]. The challenge remains to establish practical application of these techniques across industry and to feed findings back into applied research to further improve the analysis techniques and associated tool support. Detecting architecture dependencies requires expert judgment and more elaborate qualitative techniques. Our goal is to highlight these categories of dependencies so that others can gain insight and provide similar examples.

We discuss our experience investigating these dependencies using DSMs and architectural views. In describing our analysis and experience, we followed an exploratory case study protocol with the intent to address two questions:

Q1: Does the dependency structure extracted from the system's static code structure provide insight into rework costs associated with testing and technology upgrade?

Q2: Can additional architectural information be represented in the dependency structure to provide insight into rework costs associated with testing and technology upgrade?

From a practitioner's point of view, answers to these questions establish the rationale for and usefulness of dependency analysis as well as provide insights into developing an approach for conducting this analysis.

The case study is conducted with one industrial software project for which one of the authors has served as the architect. Our unit of analysis is change propagation based on structural dependency

ICSE Companion'14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2768-8/14/05...$15.00
http://dx.doi.org/10.1145/2591062.2591185

information. We use a narrative analytical approach, relying on the architecture artifacts described by the architect in four documents on module structure, design approaches, architectural decisions, and evolution of the system. We analyze the structural dependencies quantitatively using tools and techniques and then augment the dependencies with type information to conduct a qualitative analysis with feedback from the architect.

We do not limit our analysis to the module view but also investigate dependencies observed from the component-and-connector (C&C) view of the architecture, which documents the system's units of execution [5]. We observe that while dependencies such as change propagation or uses are straightforward to see from the static module view and code structures, those dependencies observed from other views have an impact on the life-cycle cost for operations and evolution. Through this case study we identify tradeoffs, especially those related to safety-critical testing versus cost, that take advantage of rigorous dependency analysis tools and techniques that not only provide input for static module organization but also guide decision making from the C&C and deployment views.

In Section 2, we review dependency analysis techniques and their relevance to architectural tradeoffs. In Section 3, we describe the shared-computing resource architecture and the significance of different types of dependencies to the evolution goals of reducing safety testing and technology upgrade costs, generalized from observing the evolution of safety-critical systems. Section 4 revisits the architecture using knowledge of behavior and design guidelines to capture additional dependencies that are critical from runtime and deployment perspectives. Sections 5 and 6 are devoted to our conclusions and discussion of future work, respectively. Our study confirms that existing dependency analysis tools fall short of capturing quality attribute tradeoffs for architecting and suggests recommendations for integrating dependency analysis tools in model-based development environments.

2. ARCHITECTURAL DEPENDENCY

The motivation to understand complex system architecture-level dependencies is often rooted in controlling the cost of change. While the cost of change is a significant concern that affects the follow-on development effort, a closer look at architecture-level dependencies and the tradeoffs reveals that the dependencies drive more than the development effort; they also have a significant impact on costs related to the life cycle and operational environment. Separating such concerns encourages opportunistic decisions that may fall short of an elegant optimal architecture yet satisfy key quality concerns and business goals. Closer study of the issues in a long-lived safety-critical system that has undergone several evolution cycles revealed significant tradeoffs when analyzing dependencies:

Change impact on development effort: Change impact often focuses on dependencies at the requirements level and the module structure of the system, including package, component, and class or code dependencies. Many existing dependency analysis metrics—such as coupling, cohesion, fan-in, fan-out, and stability—are used to assess the impact of change during system evolution. Data sharing, information input-output, send/receive variables, and uses dependencies are often categorized as dependencies that influence development-time cost of change.

Configuration management: While many development and design-related dependencies apply in configuration management, dependency analysis focused on the binding time of elements is critical when assessing configuration management issues. Deployment and C&C views are essential in such analysis.

Runtime environment: It is difficult to effectively reason about scheduling, availability, and response time with the module views that most dependency analysis techniques and tools utilize. These dependency types and concerns become more tightly related to the hosting environments or isolation of failure modes, which require understanding the patterns of the runtime architecture.

Testing: Security and safety-critical systems—such as aviation, health care, nuclear, and situated information awareness—all have specialized verification, validation, and assurance requirements. In such systems, the kinds and levels of testing significantly affect the cost of both development and deployment. Engineers often overlook an important tradeoff when they examine only dependencies related to development or configuration management. While the cost of changing a module may be small, the cost of testing it may be significant. Separation of concerns focusing on testing may drive overall costs down significantly.

Visual representation of system dependencies can help architects focus on significant areas that require further attention and reduce the cognitive overload in decision making. Development tools like Sonargraph for Java [22], Lattix [16], and Structure 101 [30] provide assistance for visualizing the dependencies and associated metrics such as cycles, coupling, and cohesion. These tools also focus on design rules of the architecture. Such analysis is, however, limited to calculating the impact of change at design time. Work in visualization techniques and design rules has assisted in bridging some code and architecture conformance gaps [8]. For instance, such work has been used to empirically associate the dependency profile of a system with its risks for defects or evolution profiles [28].

Visualization approaches can also increase insights into both the status and the quality of a project. These approaches include software maps and graphs in addition to DSMs. The software map is structured according to the modularity of the system. Complex files, as indicated for example by their McCabe complexity measure [18], are highlighted in three dimensions and by map color. Some challenges in visualization techniques include integrating time and making the technique fully interactive. The objective is to provide early warnings to detect costs and risks (e.g., "watch out for this class; it might be growing too big") [3].

Dependency metrics associated with these visualizations typically aim for managing software quality and favor managing modifiability. Optimizing design quality has largely meant detecting and avoiding defects rather than strategically managing key infrastructure decisions, especially in the context of architecture. Examples of metrics-based dependency analysis for defect detection, defect avoidance, and software maintainability can be found in static analysis tools that provide measures such as cohesion, coupling, fan-in, fan-out, and cyclomatic complexity [10][23][24][25][32]. A recent and more extensive study examining architecture-level software metrics revealed that many rely on some kind of dependency analysis but are limited to encapsulation, extendibility, and acyclicity [13]. Callo Arias and colleagues provide advice on additional dependency sources that can help augment such architecture-related dependency analysis [4].

3. SHARED RESOURCE ARCHITECTURE

Complex, mixed safety-critical systems try to minimize use of shared resources such as memory, processing, display graphics, network communication, and storage. Their use requires careful balancing of safety and performance qualities. A shared-computing resource architecture in such an environment can be broadly viewed as multiple partitioned applications (Apps) grouped by safety criticality, function, control, and resource interface.

The functional Apps in the original shared-computing resource architecture that we analyzed (Figure 1) send their data to a separate centralized data mover (DM). The DM transforms functional application data into a specialized resource-specific format. The DM then forwards the formatted data to software elements specific to each resource type (e.g., marshaling data for storage or transmission, rendering data for generating an image). This virtual resource (VR) interfaces with the physical resource (e.g., storage, network, display).

A significant drawback of this architecture is that, with the DM containing all formats, including those that are safety critical, the entire DM is required to be tested to the highest safety-critical level. The subsections that follow provide details on an effort undertaken to re-architect the system to isolate safety-critical software. We discuss the static dependencies that can be managed and show that the technique provides insight into tradeoffs that were made when creating the evolved architecture.

3.1 Business Goal

The business goal to lower the costs of safety testing and technology upgrade provides the rationale for understanding the re-architecting effort. Critical data formats of a single App drive the entire DM to a higher safety testing level. The DM is rather large (on the order of 100K source lines). Integration of new software continues to add to this code base, making overall safety testing cost prohibitive. Additionally, such a large module leads to memory issues with the development tool when performing functional and safety-based testing.

3.2 Architecture Requirements and Decisions

Table 1 shows the quality attribute requirements that result from the business goal. Design decisions provide rationale for how the architecture satisfies the quality attribute requirements.

Table 1. Mapping goals, requirements, and decisions

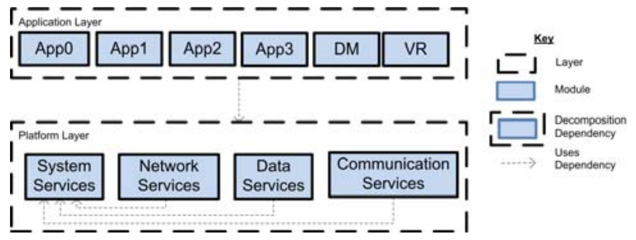
Business Goal	Quality Attribute	Architecture Design Decision
Lower safety testing and technology upgrade cost	Testability	Reduce module size. DM module is refactored into multiple data-formatting clients and a server.
	Safety	Enable safety mixed-criticality partitioning. Safety-critical modules are isolated and can be hosted on separate CPUs, insulating them from the impact of changes made to other modules.
	Modifiability	Separate function and resource concerns. Apps are separated from their associated data-formatting clients and communicate via publish-subscribe messages.

A distributed, shared-computing resource architecture is considered to be an important paradigm for achieving such objectives [7][9][12][14][21].

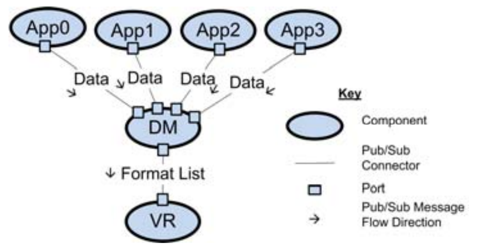
3.3 Original Architecture

The original architecture contained all user control and resource functionality, such as managing user input, formatting data, and coordinating the use of resources that depend on the data, in a single DM software module. Figure 1 shows two views of this architecture. We generalized and abstracted these views from the source documents to protect the details of the actual system while focusing the dependency analysis on those elements involved in the evolution of the generalizable architecture. We see similar examples and patterns of evolution in safety-critical embedded systems that include mobile ad-hoc network partitioning for improved use of network resources; avionics software partitioning for geo-positioning, cockpit controls, and display technology upgrades; and health care systems that include enabling secure electronic health record exchange between multiple devices and health IT records.

The module view conveys the system's principal units of implementation. The C&C view documents the system's units of execution. A representative subset of software elements in the application layer, shown in Figure 1, motivates our analysis even though there are more software elements in an actual system that has the flexibility to accommodate expansion for future applications. Similarly, we highlight a subset of business goals, software elements, and decisions to scope this use case.



(a) Module view



(b) C&C view

Figure 1. Original architecture.

These figures portray the graphical portion of the architecture description. The textual information accompanying the figures in the source documents also provided important information to fully document the views, including details of the elements, relations, and their properties and rationale. The architecture description reveals several important characteristics about the architecture:

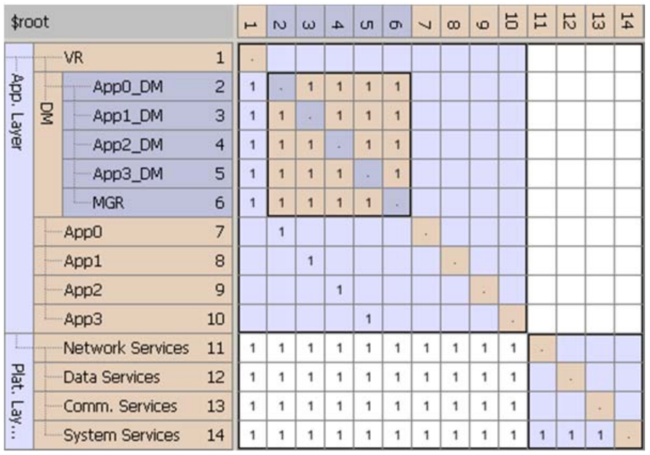
- Apps are integrated into a single application layer.
- DM receives and manages data from the Apps that require an interface to a resource. It is also responsible for input management.

- VR converts data into a suitable format, taking into account the unique configuration of the resource device.
- Partitioning of Apps is driven by separation or isolation of functionality and safety and by hardware constraints, such as CPU proximity to resource devices.
- Services are integrated into a single platform layer.

In the original architecture, certain runtime decisions in the C&C view are predetermined by decisions made in the module view:

- All data-formatting clients are hosted on a single processor by virtue of being in the same module (DM).
- Responsibilities for managing user input, transforming data, and coordinating use of the resources are consolidated into a single module (DM), which is mapped to a runtime component and deployable unit.

Figure 2 shows the DSM that captures the dependencies derived through static analysis among the different modules of the original architecture. A DSM is a matrix that maps dependencies between items in a given domain [1][29]. All elements appear in both the rows and the columns, and dependencies are signaled at the intersection points of the items in the matrix. We determined these dependencies from the views of the architecture shown in Figure 1 and the supplementary information in the source documents. The uses dependency from the module view provides information about the dependencies between layers. The data and format list connectors from the C&C view provide information about the data dependencies among the Apps. Given that the responsibilities within DM (App0_DM, App1_DM, App2_DM, App3_DM, and MGR) are not visible in any of the architecture views, we assume the worst case—that they are fully connected.



$root				1	2	3	4	5	6	7	8	9	10	11	12	13	14
App. Layer		VR	1	.													
	DM	App0_DM	2	1	.	1	1	1	1								
		App1_DM	3	1	1	.	1	1	1								
		App2_DM	4	1	1	1	.	1	1								
		App3_DM	5	1	1	1	1	.	1								
		MGR	6	1	1	1	1	1	.								
		App0	7		1					.							
		App1	8			1					.						
		App2	9				1					.					
		App3	10					1					.				
Plat. Lay...		Network Services	11	1	1	1	1	1	1	1	1	1	1	.			
		Data Services	12	1	1	1	1	1	1	1	1	1	1		.		
		Comm. Services	13	1	1	1	1	1	1	1	1	1	1			.	
		System Services	14	1	1	1	1	1	1	1	1	1	1	1	1	1	.

Figure 2. Original architecture DSM.

As shown in the DSM, the DM module provides a single point for coordination, configuration, and managing the formatting of data, but it does so at the expense of its expanding size. Due to its large share of the application layer, any changes to the DM will have a significant impact. For instance, in this architecture integrating third-party display software is quite difficult. The overall stability of the system, which measures the percentage of modules (on average) that would not be affected by a change to any module of the system, is 53.06%. System stability is computed as

$$100\% - (\text{Cumulative Component Dependency} / \text{Module Count}^2) * 100$$

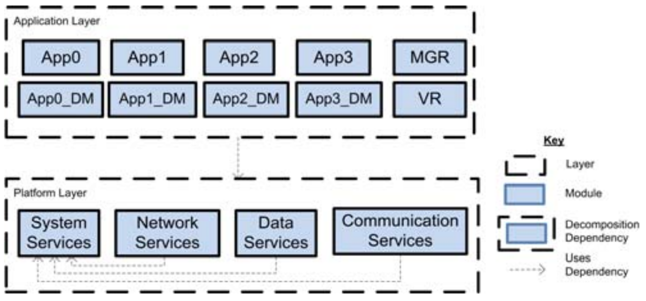
where the cumulative component dependency is the sum of all direct or indirect dependencies that modules have on each other [15]. Stability is measured as the inverse of propagation cost, the sensitivity of its architecture to changes calculated based on the transitive closure of the dependencies between modules [17]. A higher stability value indicates a system that is less sensitive to change and is preferred. Metrics for well-known systems such as Ant 1.7.1, Apache Server 2.2.8, Spring Source 3.0, Eclipse 3.0, and .NET Framework 2.0 show stability over 90% [16]. These are measures applied to the code of these systems. There are limitations in applying such structural metrics to measure properties of architectural patterns [20]. Rather than ascribe too much meaning to the absolute value, we are interested in the relative value of stability before and after the evolution to measure improvements.

Additionally, as a monolith with highly interdependent functionality, the DM cannot isolate safety-critical applications, which drives up the overall safety testing cost. Such testing costs can be associated to safety-critical testing levels enforced by particular industry standards and guidance, for example, those for avionics as described in *DO-178B, Software Considerations in Airborne Systems and Equipment Certification* [6].

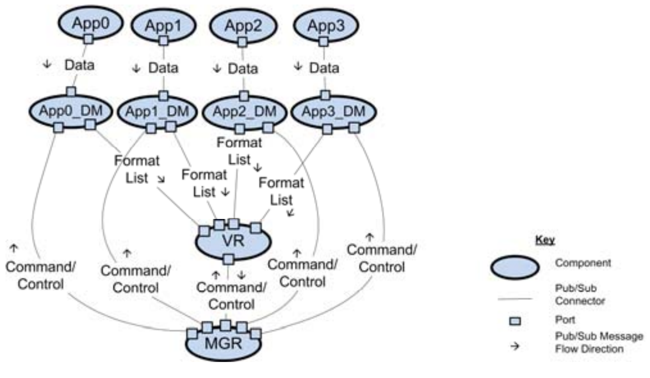
Furthermore, instrumenting DM code for safety testing increases its size, creating memory problems for testing and development tools.

3.4 Distributed Architecture

These factors suggest a refactoring of the DM module to the new distributed architecture shown in Figure 3.



(a) Module view



(b) C&C view

Figure 3. Distributed architecture.

The architecture description now shows the following characteristics:

- DM is decomposed into separate clients, App0_DM, App1_DM, App2_DM, App3_DM, and a control-and-transform manager (MGR) that manages user input and coordinates the use of resources.

- App0_DM contains the safety-critical functionality and receives its data from App0.

Decomposing the DM in this manner provides added flexibility in how modules are combined into runtime components and deployable units.

The C&C view becomes more important since certain runtime decisions that used to be predetermined are now exposed. Alternative choices can be made with several implications for the quality attribute goals of the system:

- Responsibilities for managing user input (MGR), transforming data (distributed data-formatting clients: App0_DM, App1_DM, App2_DM, App3_DM), and coordinating use of the resources (MGR/VR) are now distributed (maintainability, modifiability).

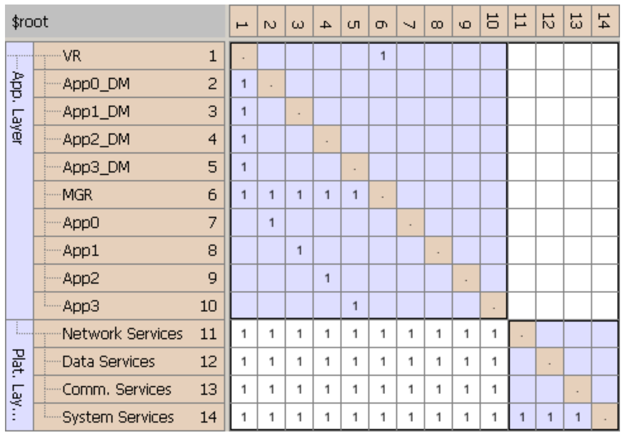
- Distributed clients can be hosted on any processor; to minimize latency, a functional App and its corresponding data-formatting client, App_DM, are hosted on the same processor (performance).

- Safety-critical clients such as App0_DM are isolated into their own user space partition apart from other software elements, so failure of one does not affect the other (safety, availability).

- Distributing responsibilities increases the number of publish-subscribe topics that are required. To enhance performance, clients process and publish their data to one or more VR instances only when they are mapped and coordinated to do so by MGR and VR. This strategy enhances both CPU utilization and network performance; network traffic is minimized to the data being actively processed (performance).

- Using MGR to coordinate the use of resources for transforming data introduces an additional configuration management burden (testability, integrability).

Figure 4 shows the DSM that captures the dependencies among the modules of the distributed architecture. We determined these dependencies from the views of the architecture shown in Figure 3 and the supplementary information in the source documents.



$root			1	2	3	4	5	6	7	8	9	10	11	12	13	14
App. Layer	VR	1	.					1								
	App0_DM	2	1	.												
	App1_DM	3	1		.											
	App2_DM	4	1			.										
	App3_DM	5	1				.									
	MGR	6	1	1	1	1	1	.								
	App0	7		1					.							
	App1	8			1					.						
	App2	9				1					.					
	App3	10					1					.				
Plat. Lay...	Network Services	11	1	1	1	1	1	1	1	1	1	1	.			
	Data Services	12	1	1	1	1	1	1	1	1	1	1		.		
	Comm. Services	13	1	1	1	1	1	1	1	1	1	1			.	
	System Services	14	1	1	1	1	1	1	1	1	1	1	1	1	1	.

Figure 4. Distributed architecture DSM.

As can be seen from the DSM, by virtue of refactoring the DM module, responsibilities are more evenly distributed. Safety-critical applications can now be isolated, thereby lowering the overall safety testing cost. Developing and integrating new third-party applications is also more straightforward and more easily accommodated. The overall stability of the system is 50.51%.

4. ANALYSIS AND DISCUSSION

The DSM views of Figures 2 and 4 capture static dependencies among the modules. A comparative analysis of the stability of the system after refactoring shows a slight deterioration and may give the impression that the effort was not cost effective. It is only when the refactored effort is analyzed with respect to the safety-critical testing levels of the components that the tradeoffs can be properly understood.

While the need for such architectural dependency analysis is clear, the analysis approach we examined does not provide adequate insights. We discuss these limitations and suggest ways to overcome them in the following sections. The discussion is structured around the two questions we set out to address.

4.1 Static Dependency Structure

Recall our first question: Does the dependency structure extracted from the system static code structure provide insight into rework costs associated with testing and technology upgrade?

Table 2 summarizes the dependencies that we captured in the DSMs in the previous section and used to compute the stability metric of each architecture.

Table 2. Module dependencies

Source Module (A)	Target Module (B)	Dependency Description
App	App_DM	Functional partitions (App) send data to their associated clients (App_DM)
MGR	App_DM	MGR orchestrates coordination with clients (App_DM) and VR
App_DM	VR	Clients (App_DM) send format-specific data list to VR
MGR	VR	MGR orchestrates coordination with clients (App_DM) and VR
VR	MGR	VR sends client management-specific data to MGR

These types of dependencies relate to the syntax or semantics of the data or the service through which two modules are dependent. We used these dependencies to analyze the change impact on the development effort, and we broadly categorize them as either data or control:

- Data (D): For a module B to execute correctly, the syntax (type or format)/semantics of the data produced by module A must be consistent with the assumptions of module B.

- Control (C): For a module B to execute correctly, the syntax (signature)/semantics of the services provided/produced by module A and invoked/used by module B must be consistent with the assumptions of module B.

Architects can determine the syntactic dependencies mostly through static analysis of code and element dependencies in the module view of the architecture. These dependencies are typically used to compute metrics that indicate the "health" of the architecture.

Table 3 captures a representative set of such architectural metrics based on research and practical experience and computed by a tool called Lattix [16]. It shows these metrics for the centralized architecture as well as the distributed architecture.

Table 3. Dependency-based metrics for architecture health

Metric	Original	Distributed	Health[*]
Stability	53.06%	50.51%	−
Average impact	6.57	6.93	−
System cyclicity	42.857%	42.857%	=
Intercomponent cyclicity	42.857%	0%	+
Internal dependencies	73	57	+
Average dependency	5.21	4.07	+
Average cumulative dependency	7.93	7.93	=
Normalized cumulative dependency	2.467	2.467	=
Connectedness	53.30%	53.30%	=
Connectedness enrichment	1.00	1.17	−
Connectedness strength	6.07	5.10	+
Coupling	16.48%	16.48%	=
Coupling enrichment	1.00	2.14	−
Coupling strength	0.49	0.30	+

[*]Health is indicated as improved (+), deteriorated (−), or remained same (=).

Other tools incorporate similar metrics and analysis. For example, Structure 101 defines a metric called XS, which is a combination of cyclomatic complexity and measures of tangle (cyclic dependencies) among classes and packages of a system [30]. While it is useful to understand cycles, such a metric illuminates only one aspect and is often not sufficient for understanding overarching architectural issues within a system. Similarly, SonarGraph and SonarQube incorporate measures of duplicate code blocks, unused methods, and average component dependencies to manage code quality as well as coding rules [22][26][27].

Architectural metrics are different from code metrics as they need to provide insights about the overall system. While there might be local code quality issues, architecture may still be sound. Or while there are no code quality issues, there might be significant end-to-end architectural issues. This can be seen from Table 3, where the metrics show a mixture of no change, slight increase, or decrease in health of the distributed architecture. It is only when the re-architecting effort is analyzed with respect to component safety-criticality levels that the tradeoffs can be properly understood.

4.2 Augmented Dependency Structure

Recall our second question: Can additional architectural information be represented in the dependency structure to provide insight into rework costs associated with testing and technology upgrade?

Many of the architectural design decisions and tradeoffs discussed in the previous section lead to dependencies other than data and control. These include

- Location (L): For B to execute correctly, the runtime location of A must be consistent with the assumptions of B.
- Sequence of control (S): For B to execute correctly, A must have executed previously within certain timing constraints.
- Quality of service (Q): For B to execute correctly, some property involving the quality of the data or service provided by A must be consistent with B's assumptions.
- Resource behavior (R): For B to execute correctly, the resource behavior of A must be consistent with B's assumptions about resource usage or ownership.
- Testing (T): To lower the overall testing cost, safety-critical aspects must be split into child modules that are separate from their non-safety-critical parent module.

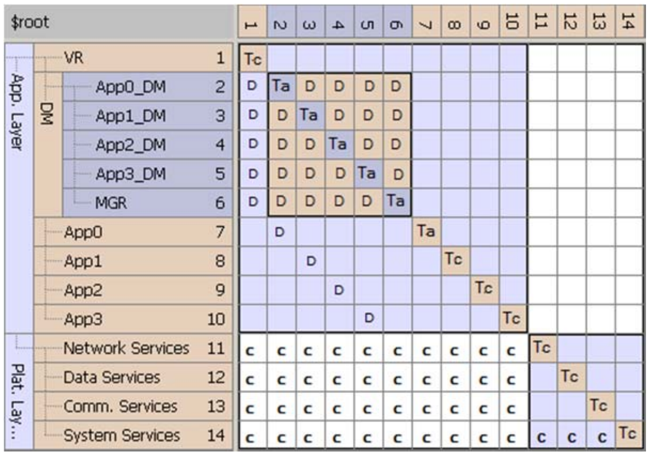
Table 4. Additional dependencies

Modules Impacted	Type	Dependency Description
App_DM, VR	Q	App_DM process and publish their data to one or more VR instances only when they are mapped and coordinated to do so (by MGR) to enhance performance.
MGR, App_DM	R	App_DM uses the resource (VR) that MGR owns.
MGR, App_DM, VR	S	MGR, App_DM, and VR collaborate closely and must be configured consistently to execute in a certain order in conformance with the command/control protocol.
App, App_DM	L+	App and App_DM are hosted on the same CPU to minimize continuous update latency.
App, App_DM	S	App_DM is scheduled to run immediately following the associated App.
MGR, App, App_DM	L−	MGR, App, and App_DM are isolated into their own user space partition so failure of one does not affect the other.
App, App_DM, MGR, VR	Ta Tc	Safety-critical responsibilities are isolated into their own modules, App0 and App0_DM, to lower the amount of level A testing (Ta). Noncritical responsibilities are isolated into their own modules to optimize testing at level C (Tc).

Recognizing these dependencies requires system analysis beyond the module view; runtime and deployment views must also be considered. In this study, we relied on our existing knowledge of types of dependencies that one module can have on another and applied it to the architecture description, including the module and C&C views of the architecture [2]. Analyzing the C&C view, and supplementing the structural information with information about behavior, design, and deployment guidelines (how the components are allocated to partitions within the execution platform), revealed additional dependencies that influence the achievement of runtime quality attributes and affect the cost of rework and testing, as summarized in Table 4. The testing dependency was something new and unexpected as it was not on our original list.

We represent these additional dependencies in the DSMs of the instantiated original and distributed architecture shown in Figure 2 and Figure 4.

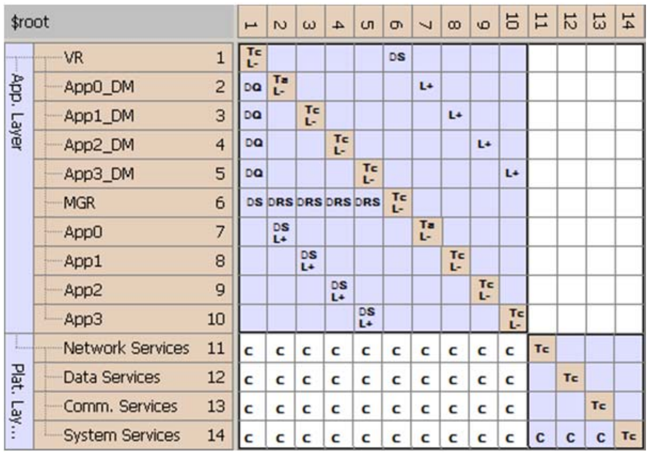
Figure 5 shows the augmented DSM for the original architecture.



$root				1	2	3	4	5	6	7	8	9	10	11	12	13	14
App. Layer		VR	1	Tc													
App. Layer	DM	App0_DM	2	D	Ta	D	D	D	D								
App. Layer	DM	App1_DM	3	D	D	Ta	D	D	D								
App. Layer	DM	App2_DM	4	D	D	D	Ta	D	D								
App. Layer	DM	App3_DM	5	D	D	D	D	Ta	D								
App. Layer	DM	MGR	6	D	D	D	D	D	Ta								
App. Layer		App0	7		D					Ta							
App. Layer		App1	8			D					Tc						
App. Layer		App2	9				D					Tc					
App. Layer		App3	10					D					Tc				
Plat. Lay...		Network Services	11	c	c	c	c	c	c	c	c	c	c	Tc			
Plat. Lay...		Data Services	12	c	c	c	c	c	c	c	c	c	c		Tc		
Plat. Lay...		Comm. Services	13	c	c	c	c	c	c	c	c	c	c			Tc	
Plat. Lay...		System Services	14	c	c	c	c	c	c	c	c	c	c	c	c	c	Tc

Figure 5. Augmented DSM for the original architecture.

Figure 6 shows the augmented DSM for the distributed architecture.



$root			1	2	3	4	5	6	7	8	9	10	11	12	13	14
App. Layer	VR	1	Tc L-					DS								
App. Layer	App0_DM	2	DQ	Ta L-					L+							
App. Layer	App1_DM	3	DQ		Tc L-					L+						
App. Layer	App2_DM	4	DQ			Tc L-					L+					
App. Layer	App3_DM	5	DQ				Tc L-					L+				
App. Layer	MGR	6	DS	DRS	DRS	DRS	DRS	Tc L-								
App. Layer	App0	7		DS L+					Ta L-							
App. Layer	App1	8			DS L+					Tc L-						
App. Layer	App2	9				DS L+					Tc L-					
App. Layer	App3	10					DS L+					Tc L-				
Plat. Lay...	Network Services	11	c	c	c	c	c	c	c	c	c	c	Tc			
Plat. Lay...	Data Services	12	c	c	c	c	c	c	c	c	c	c		Tc		
Plat. Lay...	Comm. Services	13	c	c	c	c	c	c	c	c	c	c			Tc	
Plat. Lay...	System Services	14	c	c	c	c	c	c	c	c	c	c	c	c	c	Tc

Figure 6. Augmented DSM for the distributed architecture.

The augmented DSM uses letters instead of numbers to distinguish the different types of dependencies among the modules. Several aspects of this DSM are worth noting:

1. Dependencies of type C and D among the modules can be determined through static analysis, but all others can be easily missed.
2. An individual cell can be multivalued to indicate different types of dependencies among the modules involved in that relationship. Static code analysis approaches, typically used to enforce quality and facilitate architecture discovery, fail to detect and distinguish among the different kinds of architectural dependencies such as L, S, Q, R, and T.
3. Dependencies shown in the diagonal cells are challenging to represent. The location (L−) dependency on the diagonal indicates that each component is contained in its own user space at runtime and availability is achieved by isolating modules rather than through any relationship among modules.
4. The testing (Ta, Tc) dependency indicates the testing level for each component, and lower testing cost is achieved by separating safety-critical modules from non-safety-critical modules.
5. The location, quality-of-service, resource, and sequence-of-control dependencies are not visible in the original architecture. These runtime and deployment decisions are predetermined by decisions made in the module view and constrained by the centralized nature of the design and the requirements it was meant to support.

These additional dependencies imply increased coupling and thus decreased stability and increased testing cost based on the dependency metrics.

A key issue in developing new metrics is how to take advantage of the additional information about dependencies. Let us walk through an analysis to see how the information was applied to reduce the testing cost of the system. We show the criticality levels for components in the original and target architecture in the diagonals of the DSMs in Figures 5 and 6. The criticality level A is the strongest, requiring intensive code review and testing efforts. For example, code of components classified with this level shall be fully covered using the Modified Condition/Decision Coverage (MC/DC) method, leading to a time-consuming effort. On the other hand, lower levels (such as C) do not have such requirements, and the code need only be validated against the Statement Coverage method, which is less costly and time consuming. Figure 5 shows Ta-level testing for all elements within the DM module. Figure 6 shows fewer elements requiring testing level Ta.

The comparison shows marked improvement in the distributed architecture with respect to the cost and effort of testing based on the safety-criticality levels. Yet if we follow the dependencies for possible ripple effects, we see that an element requiring testing level Ta, App0_DM, depends on MGR, which in turn depends on VR, which depends on all the other App_DMs. If any of these elements change, they might trigger Ta-level testing. The module that has to be tested is smaller, but the frequency of testing when a change occurs is the same. Looking to Figures 5 and 6 and the type information offers us a more refined interpretation of propagation. App0_DM depends on MGR through a DRS dependency, MGR depends on VR through a DS dependency, and VR depends on the App_DM through a DQ dependency.

The visual representation of the additional level of safety-critical testing information on the DSM structure, while not quantified, provides additional information to compare the architectures before and after the evolution. Additional work is needed to quantify how the different types affect the strength of the dependencies and the properties of propagation for a given system. Relying solely on module metrics proves to be insufficient and does not demonstrate that the refactoring enables reduction of testing and hence reduces overall life-cycle costs.

5. CONCLUSION

In this case study, dependency analysis was applied to a representative industrial shared-computing resource architecture in both original and evolved distributed states. The goal was to investigate how current dependency analysis techniques and metrics could provide information about the architectural tradeoffs that led to the re-architecting effort. For any long-lived, highly regulated, and safety-critical system, upgrades are the norm rather than the exception, and there is an ever-increasing pressure to reduce life-cycle costs. This requires software architecture analysis and design techniques to better incorporate quality concerns that are relevant in the later stages of the system's life cycle, including testing, deployment, and operations. The underlying challenge is investigating techniques to assist in repeatable and consistent multi-view architecture analysis.

The upgrade for the shared-computing resource architecture in particular needed to take into account safety-critical testing costs and operational behavior. The upgrade effort was motivated by the goal to reduce safety-related testing cost. Safety-critical formats drive up the entire module in which they reside to a higher safety testing level. Integration of new shared-computing resource software adds to this code base, driving up the overall safety testing cost. Additionally, the large size of the module leads to memory issues with test and development tools. The design does not provide isolation of mixed-criticality formats, resulting in requalification testing of the large module for any change in code for the software with shared-computing resources.

The take-aways of our study are based on the validity of the existing techniques and the gap between the research and the industry applications of the existing techniques for such model-based architecture dependency analysis:

- Analysis of module structure dependencies falls short of showing benefits of the re-architected system for these key business drivers.
- Existing dependency analysis tools fall short of capturing multiple quality attribute tradeoffs, in particular safety-critical testing, when it comes to architecting. Safety-critical testing is a source of major costs in many systems.
- Repeatability and practicality are barriers to adoption as many critical dependencies and tradeoffs can be captured only by examining multiple artifacts and augmenting analysis manually.

We present this study to call attention to design decisions that architects should consider when making costly design tradeoffs. Such analysis is useful when architects need to opt for feature development over upgrades of technology and can provide a rationale for allocating resources to the most useful upgrades. This analysis demonstrates the use of the diagonal in DSM visualization to capture such information as well as to incorporate additional dependencies by type. Other visualization techniques such as network diagrams and graphs can also be investigated.

6. FUTURE WORK

Making architecture views and resulting dependencies visible is a first step toward estimating the more elusive cost of long-term proactive change for strategic reasons. Incorporating safety-critical testing into analysis demonstrates the impact of testing on overall development that motivates evolution decisions.

Our motivation in investigating possible quantification and visualization techniques with existing tool support, such as DSMs, and system-level metrics, such as the stability metric, is to provide an objective comparative basis that can be incorporated into the engineering development environment. For example, when performing an architecture tradeoff analysis, engineers often depend on prototyping. Currently, there is increasing incentive to apply rigorous modeling methods to better assess tradeoffs and ultimately drive down overall system development cost. Model-based development and supporting tools are seen as key enablers, especially in safety- and mission-critical industrial systems, despite existing tooling and research challenges [11].

System and architecture analysis should also manage key architecturally significant business drivers. Isolation of safety-critical software is often done to minimize software safety testing; however, isolation should also consider tradeoffs of other system drivers. Anticipated amount of change is also a consideration. For example, safety-critical requirements and software typically experience minimal change after initial development whereas general-purpose applications are well known to evolve over the product life cycle. Additionally, other architecture views dominate in safety-critical domains when design explorations are made, such as state charts. In such domains, automatically generated code is augmented with manually written code to add other functionality. It is our experience that this breaks the know-how in existing domain-independent metrics. A potential future direction is investigating domain-specific metrics scoped by multi-view architecture analysis.

Our study reveals two evolving strategies to provide more situated and holistic insights for architecture decision making during development. First, analysis tools should be employed in parallel with any prototyping activity. Second, dependency analysis tools should be integrated into model-based development environments. Model-based development maturation could allow for visualization of module, C&C, and deployment dependencies.

7. ACKNOWLEDGMENT

We thank Neil Ernst and Neeraj Sangal for their feedback.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. DM-0000698.

8. REFERENCES

[1] Bartolomei, J. et al. 2007. *Analysis and Applications of Design Structure Matrix, Domain Mapping Matrix, and Engineering System Matrix Frameworks*. Working Paper. Massachusetts Institute of Technology Engineering System Division, Cambridge.

[2] Bass, L. et al. 2013. *Software Architecture in Practice*. Addison-Wesley, Upper Saddle River, NJ.

[3] Bohnet, J. and Döllner, J. 2011. Monitoring code quality and development activity by software maps. *Proceedings of the 2nd Workshop on Managing Technical Debt* (Waikiki, HI, May 21–28, 2011). ACM, New York, NY, 9–16.

[4] Callo Arias, T. B. et al. 2011. A practice-driven systematic review of dependency analysis solutions. *Empir. Softw. Eng.* 16, 5 (Oct. 2011), 544–586.

[5] Clements, P. 2011. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, Upper Saddle River, NJ.

[6] DO-178B. 1992. *RTCA/DO-178B, Software Considerations in Airborne Systems and Equipment Certification*. Advisory Circular. RTCA, Inc., and EUROCAE, Washington, DC.

[7] Ge, X. et al. 2010. An iterative approach for development of safety-critical software and safety arguments. *Agile Conference (AGILE), 2010* (Orlando, FL, Aug. 9–13, 2010). IEEE Computer Society Press, Washington, DC, 35–43.

[8] Hinsman, C. et al. 2009. Achieving agility through architecture visibility. *Proceedings of the 5th International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems* (East Stroudsburg, PA, June 24–26, 2009). Springer, Berlin, 116–129.

[9] Karp, J. and Paltrow, S. J. 2007. Copter contract gives Lockheed choppy ride. *Wall Street Journal* (Jul. 24, 2007): http://online.wsj.com/article/SB118523289611175502.html

[10] Kim, M. and Notkin, D. 2009. Discovering and representing systematic code changes. *Proceedings of the 31st International Conference on Software Engineering* (Vancouver, Canada, May 16–24, 2009). IEEE Computer Society Press, Washington, DC, 309–319.

[11] Kirstan, S. and Zimmermann, J. 2010. Evaluating costs and benefits of model-based development of embedded software systems in the car industry: Results of a qualitative case study. *The Fifth Workshop "From Code Centric to Model Centric: Evaluating the Effectiveness of MDD (C2M:EEMDD)," European Conference on Modeling Foundations and Applications* (Paris, France, Jun. 15–18, 2010). Paris, CEA LIST, 18–29.

[12] Koontz, R. 2012. Apache Mission Processor software: mixed-criticality partitioning & distributed display architecture. *68th Annual Forum & Technology Display* (Fort Worth, TX, May 1, 2012).

[13] Koziolek, H. 2011. Sustainability evaluation of software architectures: a systematic review. *Proceedings of the Joint ACM SIGSOFT Conference – QoSA and ACM SIGSOFT Symposium – ISARCS on Quality of Software Architectures – QoSA and Architecting Critical Systems – ISARCS* (Boulder, CO, June 20–24, 2011). ACM, New York, NY, 3–12.

[14] Kuz, I. et al. 2012. An architectural approach for cost effective trustworthy systems. *2012 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA)* (Helsinki, Finland, Aug. 20–24, 2012). IEEE Computer Society Press, Washington, DC, 325–328.

[15] Lakos, J. 1996. *Large-Scale C++ Software Design*. Addison-Wesley, Upper Saddle River, NJ.

[16] Lattix, Version 6.7.2. 2012. http://www.lattix.com.

[17] MacCormack, A. et al. 2006. Exploring the structure of complex software designs: an empirical study of open source and proprietary code. *Manage. Sci.* 52, 7 (Jul. 2006), 1015–1030.

[18] McCabe, T. J. 1976. A complexity measure. *IEEE T. Software Eng.* SE-2, 4 (1976), 308–320.

[19] Nord, R. L. et al. 2012. In search of a metric for managing architectural technical debt. *2012 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA)* (Helsinki, Finland, Aug. 20–24, 2012). IEEE Computer Society Press, Washington, DC, 91–100.

[20] Nord, R. L. et al. 2013. Variations on using propagation cost to measure architecture modifiability properties. *29th IEEE International Conference on Software Maintenance* (Eindhoven, The Netherlands, Sep. 22–28, 2013). IEEE Computer Society Press, Washington, DC.

[21] Parsons, D. 2012. Military helicopter fleets showing their age. *Natl. Defense Mag.* (Feb. 2012). http://www.nationaldefensemagazine.org/archive/2012/February/Pages/MilitaryHelicopterFleetsShowingTheirAge.aspx

[22] Penchikala, S. 2010. Architecture analysis tool SonarJ 6.0 supports structural debt index and quality model. *InfoQ* (Aug. 16, 2010). http://www.infoq.com/news/2010/08/sonarj-6.0

[23] Sangwan, R. S. and Neill, C. J. 2009. Characterizing essential and incidental complexity in software architectures. *Joint Working IEEE/IFIP Conference on Software Architecture, 2009 European Conference on Software Architecture. WICSA/ECSA 2009* (Cambridge, UK, Sep. 14–17, 2009). IEEE Computer Society Press, Washington, DC, 265–268.

[24] Schumacher, J. et al. 2010. Building empirical support for automated code smell detection. *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (Bolzano, Italy, Sep. 16–17, 2010). ACM, New York, NY, 8:1–8:10.

[25] Sethi, K. et al. 2009. From retrospect to prospect: assessing modularity and stability from software architecture. Joint Working IEEE/IFIP Conference on Software Architecture, 2009 European Conference on Software Architecture. WICSA/ECSA 2009 (Cambridge, UK, Sep. 14–17, 2009). IEEE Computer Society Press, Washington, DC, 269–272.

[26] Sonargraph. 2013. http://www.hello2morrow.com/products/sonargraph.

[27] Sonarqube. 2013. http://www.sonarqube.org/.

[28] Sosa, M. et al. 2010. *Product Architecture and Quality: A Study of Open-Source Software Development*. Technical Report #ID 1657870. Social Science Research Network, Rochester, NY.

[29] Sosa, M. et al. 2007. Studying the dynamics of the architecture of software products. *Proceedings of the ASME 2007 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, IDETC/CIE 2007* (Las Vegas, NV, Sep. 4–7, 2007). ASME, New York, NY, 329–342.

[30] Structure 101, Version 3.5, Build 1527. 2012. http://structure101.com.

[31] Telea, A. et al. 2010. Visual tools for software architecture understanding: a stakeholder perspective. *IEEE Software* 27, 6 (Nov./Dec. 2010): 46–53.

[32] Wong, S. et al. 2011. Detecting software modularity violations. *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, HI, May 21–28, 2011). IEEE Computer Society Press, Washington, DC, 411–420.