2016 13th Working IEEE/IFIP Conference on Software Architecture

Missed Architectural Dependencies:

The Elephant in the Room

Robert L. Nord,[1] Raghvinder Sangwan,[1,2] Julien Delange,[1] Peter Feiler,[1]
Luke Thomas,[3] and Ipek Ozkaya[1]

[1] Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, USA
{rn, jdelange, phf, ozkaya
@sei.cmu.edu}

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

[3] School of Engineering and Technology
Indiana University–Purdue University
Indianapolis, IN, USA
nlulthom@iupui.edu

***Abstract*—Research in code and architectural analysis has demonstrated that a clear understanding of structural dependencies among software elements helps developers comprehend the impact of change. Yet examples are abundant from industry of major issues due to missed dependencies associated with different views of the architecture. Key concerns include dependencies related to allocation of modules to implementation packages to improve safety-critical testing and allocation of implementation packages to hardware partitions to optimize performance. In this paper, we present an in-depth study of a safety-critical system that underwent major changes as a result of missed architectural dependencies. We describe the challenges that resulted in re-architecting the system, the techniques we used for intervention, our results, and the developers' perspective. While the engineering tools provided coverage of design concerns, they missed implications of end-to-end integration testing, latency, and cost of change. In our study, we observed that the tools led the engineers to focus on data and control flow and therefore to miss many data-entity relationships, resource behavior, and deployment-related dependencies. Research continues to focus on more tooling and automation to assist with dependency analysis rather than interim, easier-to-adopt solutions. Our findings demonstrate that providing developers with a lightweight, semantically well-defined description of dependencies enables them to reason about change impact and propagation implications that they might otherwise overlook.**

Keywords—software architecture; architecture views; architecture analysis; dependency analysis; change propagation; testing; re-architecting

I. Introduction

In this paper, we present a study of an industrial, safety-critical system that underwent major changes as a result of missed architectural dependencies. This multiyear aircraft engine control system was being developed and deployed as part of a product-line strategy. The development project included a local team and a subcontractor team. Independently each team developed its piece of the system correctly. Engineering tools provided coverage of inputs and outputs, internal data dependencies, and verification of the design against the requirements. However, the problem was introduced and discovered at the architectural level.

The traditional approach focused on data and control flow dependencies and missed end-to-end concerns including implications for critical integration testing, end-to-end latency, and managing cost of change that seem obvious in retrospect. Developers on the project perceived a need for an approach that considered dependencies associated with multiple perspectives or views of the architecture of a system, including implementation, run time, and deployment. In response, we proposed a solution, a lightweight guide of dependency types focusing on multiple views and their mapping, and an analysis approach that enabled early identification and discussion of architectural concerns that might otherwise be missed.

While research continues to focus on more tooling and automation to assist with dependency analysis, architects and developers still lack a consistent semantic treatment of all dependency types. This paper demonstrates an interim solution toward that goal: a multi-view dependency guide that enables team members to discuss key architectural concerns and use tools accordingly. Our study outcome demonstrates that improved tooling without a shared understanding of architectural depedencies exacerbates lack of attention to the holistic, multi-view architectural analysis.

We describe the industrial safety-critical system used in our study in Section 2. In Section 3, we describe our analysis approach and document the baseline dependencies captured from the module view of the architecture and the enhanced run-time and deployment dependencies captured from the component-connector and allocation views of the architecture. In Section 4, we use the dependency structure matrix (DSM) [1][2] to represent and analyze the baseline and enhanced set of dependencies. We also use the DSMs to analyze and understand the impact of change. In doing so, we highlight how the analysis of dependencies based on multiple views of an architecture can be more effective in providing information about the impact of a change. Section 5 discusses the developers' perspective. Section 6 presents related work, and Section 7 concludes the paper.

978-1-5090-2131-4/16 $31.00 © 2016 IEEE
DOI 10.1109/WICSA.2016.32

II. SAFETY-CRITICAL ENGINE CONTROL SYSTEM

A stepper motor system (SMS) is a part of an aircraft engine control system (ECS) that manages fuel flow by adjusting a fuel valve. The SMS is commanded to open or close the fuel valve. The SMS translates the command into the number of steps it must take to move the valve to the desired position. It must complete its operation in a time proportional to the distance covered between its current position and the desired position.

During its operation, the SMS may receive a new command from the ECS and must respond immediately. The SMS maintains a record of the fuel valve's desired, commanded, and actual positions, which must all be the same upon completion of the most recent command. A homing command is executed at initialization to move the valve to a fully closed position and synchronize the values of the desired position, commanded position, and actual position.

The SMS is an open-loop system with no feedback on the successful execution of the steps it must take for a position-change command. Since it is a safety-critical system, the developers must ensure that when they make changes to the system, the stepper motor continues to operate as expected. Untangling propagating effects of changes that result in unanticipated issues can have a time-consuming impact on troubleshooting the cause of the issues. Fig. 1 shows the operational context for the SMS derived from SCADE [3], the model-driven, embedded, safety-critical software and hardware engineering tool used by the developers.

The SMS is part of an engine control feedback loop and functions as an actuator for a fuel valve. The fuel valve controls the fuel flow to an engine, which has a built-in thrust sensor that provides the thrust reading. The ECS uses the thrust reading to control the desired position of the fuel valve.

To generate a level of thrust, the ECS communicates the desired position to the SMS, which in turn opens the fuel valve to the desired position through a mechanical control interface. The fuel valve controls the flow of fuel to the engine, thereby managing the level of thrust. A sensor communicates the thrust reading from the engine to the ECS, thus closing the feedback loop.

III. CHANGE IMPACT ANALYSIS

Although the original design of the SMS was developed and verified in SCADE, only during system tests was it discovered that, under certain circumstances, the actual fuel flow from the fuel valve to the engine did not correspond to the desired fuel flow commanded by the ECS. The problem was traced to missed execution of commanded steps due to variation in execution time. This prompted a refactoring effort to remediate the problem. Before making any changes to the system, the developers on the project perceived a need for an approach that could provide a holistic view of the dependencies within the system so they could understand the impact of changes to the system. Traditional approaches that considered only inter-module dependencies discovered through static code analysis were deemed insufficient.

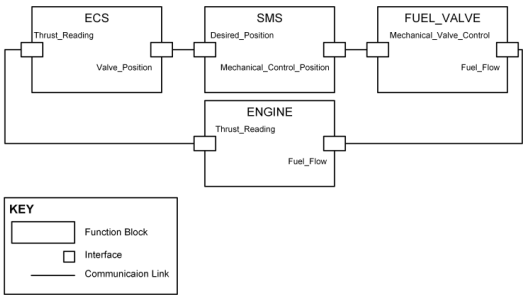

Fig. 1. Block diagram showing the context within which the SMS operates.

A. Analysis Approach

The target of our analysis was the software-reliant system's maintainability, defined as the "degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers" [4]. Maintainability is subdivided into modularity, reusability, analyzability, modifiability, and testability. We focused on modularity, the "degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on others" [4]. Impact is the ripple effect of a change in one module that may cause other modules within a system to not function properly. The change to a module may be motivated by achieving better separation of concerns to improve security, reliability, or safety-critical testing; tuning its resource consumption to improve performance or availability; or adapting its interface for achieving interoperability.

To address the impact of a change, developers needed to follow the dependencies from the modules of a system that were the focus of a change to the dependent modules that will be affected by the change. However, to effectively apply this approach, the developers needed to identify these dependent modules and their relationships. One key issue was the lack of a concrete definition of dependency among the software engineers. There were different interpretations of dependencies for managing software-to-software and software-to-hardware allocations. The developers unintentionally assumed that the tool managed those dependencies. Another issue was that implied run-time and allocation dependencies could not be easily mapped back to the implementation modules to assess the impact of change.

The literature lacks a clear description and collection of such dependencies [5]. Work with professional software architects also reveals that support for analyzing the dependencies among key architectural decisions remains a gap in industry practices [6]. Hence, for this study, we collected descriptions of common dependencies using multiple sources. Considering multiple views of the system and mapping the architectural dependencies to these multiple views serve as the ultimate ground truth; hence we started with architectural dependencies described from a multiple-view perspective, focusing on software [5][7][8]. In order to ensure that key embedded software concerns were captured, in particular software-to-hardware allocation and real-time scheduling, we also reviewed domain-specific architecture examples as represented in [9][10][11]. We narrowed the list

based on our earlier experience with safety-critical systems [12] and created a list of dependency types as an initial guide for developers for identifying and describing dependencies to analyze change scenarios. Table 1 summarizes these dependency types relevant to our study.

After creating this list of dependency types, we analyzed the system together with an architect, a tester, and a developer from the project team using the following steps:

1. *Identify change scenarios that will be used to determine the degree of effectiveness with which the system can be modified to satisfy those scenarios*: The inputs to this step are existing, unfulfilled, or new evolutionary and maintainability requirements. The output is a list of change scenarios formulated from these requirements.
2. *Create a baseline dependency model for the system from its module view*: The input for this analysis is the existing model of the system. As needed, the system's architecture documents and discussions with the development team help clarify ambiguities. The output is a baseline DSM that shows module dependencies.
3. *Create an enhanced dependency model for the system from its module, component-connector, and allocation views*: Once the baseline DSM is generated, we run scenarios through focused architectural analysis and look at not only the module but also the component-connector and deployment views (e.g., using formal modeling tools for latency and safety-criticality analysis and looking at allocation of software components to hardware components). Where needed, we use architecture documents for clarification and discussions with the development team, using the list of types as a guide for identifying dependencies. The output is an augmented DSM that captures dependencies from multiple views of the system.
4. *Identify the impact of change*: We use the baseline and enhanced dependency models (DSMs) and the change scenarios captured in the previous steps to identify the impact of change. We also evaluate the effectiveness of the enhanced model and, therefore, the effectiveness of multi-view dependency analysis.

This is an iterative process in which we refine the change scenarios as we elaborate the dependency models.

TABLE 1. DEPENDENCY TYPES

Dependency Type		Description
A	Aggregation	Data Element A and Data Element B have a semantic coherence that can be aggregated as Module AB.
C	Control	Module A depends on the presence of a correctly functioning Module B.
D	Data	For Module B to execute correctly, the syntax (type or format) and semantics of the data produced by Module A must be consistent with the assumptions of Module B.
L	Location	For B to execute correctly, the run-time location of A must be consistent with the assumptions of B.
R	Allocation of responsibilities	To separate concerns, Responsibility A (behavior and functionality) is assigned to Design-Time Element B (e.g., functional decomposition, safety criticality).
S	Sequence of flow	For B to execute correctly, it must receive the data produced by A in a fixed sequence (data flow). For B to execute correctly, A must have executed previously within certain timing constraints (control flow).
P	Physical resource behavior	For B to execute correctly, the resource behavior of A must be consistent with B's assumptions about physical resource (such as bandwidth, memory, storage capacity, and CPU) usage or ownership.
Q	Quality of service	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.
V	Virtual resource behavior	For B to execute correctly, the resource behavior of A must be consistent with B's assumptions about virtual resource usage or ownership.

B. Change Scenarios

The developers analyzed the problem and formulated six change scenarios, shown in Table 2, to correct it. These scenarios may look obvious, but it is often the case that assumptions at the team level get overlooked at the system level [13], as it was the case in this system.

Scenarios 1 and 2 represent the intended but unfulfilled requirements of the original design and relate to modifiability; the SMS should be configurable to express desired and actual fuel valve positions as percent open. In addition, the position eventually must be translated to the direction of motion and number of steps that a stepper motor must move to achieve the desired position of a fuel valve.

The remaining change scenarios evolved during the analysis process. The lightweight dependencies in Table 1 were used to understand the problem and were also helpful with eliciting and refining these scenarios.

Scenario 3 relates to safety and reliability. The SMS must guarantee consistency between the fuel valve position commanded by the ECS and the actual fuel valve position attained by the stepper motor.

TABLE 2. CHANGE SCENARIOS FOR SMS

ID	Change Scenario
Scenario 1	The desired fuel valve opening shall be commanded by the ECS in terms of *PercentOpen* and must be translated into the position that the stepper motor must attain.
Scenario 2	The actual position of the stepper motor shall be expressed in units of *PercentOpen* and must be translated into the position that the stepper motor must attain.
Scenario 3	At startup completion and at command completion, the actual position of the stepper motor must be the same as the position commanded by the ECS.
Scenario 4	A desired position command shall be completed within $T = MaxPosition * \max(StepDuration)$.
Scenario 5	The command duration shall be proportional to the distance between the current and desired position, i.e., not exceed $roundup\ (\ |\ Desired\_Position - Mechanical\_Control\_Position\ |\ * MaxStepCount/100) * FrameDuration$.
Scenario 6	There shall be a delay of no more than one frame before responding to the newly received command.

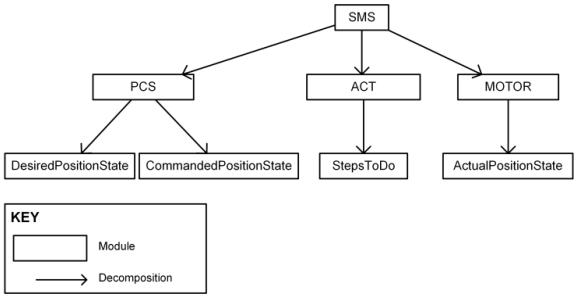

Fig. 2. Module view of the SMS architecture.

Scenarios 4, 5, and 6 focus on how the latency associated with moving the fuel valve to the position commanded by the ECS affects performance, reliability, and safety. In these scenarios, *PercentOpen* indicates the open position of the fuel valve in percentage units, *MaxPosition* indicates the maximum stepper motor position in units of steps when the fuel valve is fully open, *StepDuration* indicates the time duration for each stepper motor step, *Desired_Position* is the intended fuel valve position in percentage, *Mechanical_Control_Position* is the current fuel valve position in percentage, *MaxStepCount* has a value of maximum steps per frame, and *FrameDuration* has a value of the time duration of a single frame. These parameters and values were critical for analyzing the response measures and understanding where the architecture failed.

C. Baseline Dependencies

We analyzed the scenarios presented in Table 2 for their derived maintainability aspects, specifically the degree of effectiveness with which the system can be changed to satisfy them.

Fig. 2 shows a module view of the SMS that addresses Scenarios 1 and 2. We abstracted this view from the source information to hide the details of the system while focusing the developers' attention on the elements involved in the change scenarios necessary for dependency analysis.

TABLE 3. DEPENDENCIES BASED ON THE MODULE VIEW OF THE SMS

Dependency	Description
Type: Data (D) and Control (C) **Source**: ACT **Target**: PCS	ACT receives position change command sequence from PCS.
Type: Data (D) and Control (C) **Source**: MOTOR **Target**: ACT	MOTOR receives step execution command signals from ACT.
Type: Data (D) **Source**: PCS **Target**: DesiredPositionState	PCS updates the *DesiredPosition State*. This value is used to calculate steps for ACT.
Type: Data (D) **Source**: PCS **Target**: CommandedPositionState	PCS updates the *Commanded PositionState*. This value is used to calculate steps for ACT.
Type: Data (D) **Source**: ACT **Target**: StepsToDo	ACT updates *StepsToDo* and uses it to generate an electrical command signal and turn MOTOR through a control interface.
Type: Data (D) **Source**: MOTOR **Target**: ActualPositionState	MOTOR records (updates) its position at any given point in time in *ActualPositionState*.

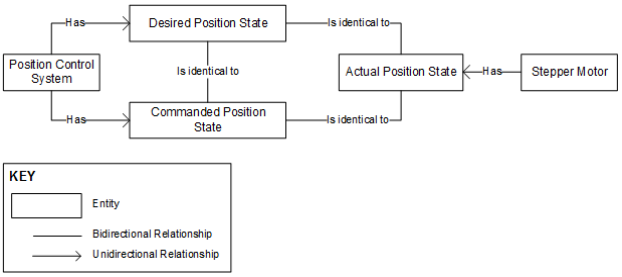

Fig. 3. Data model of the SMS architecture.

SMS is decomposed into three modules. The process control system (PCS) receives the desired fuel valve position, calculates the number of steps the stepper motor must move as the difference between the desired position and the current position, and directs the actuator (ACT) to use this value to instruct MOTOR to turn the fuel valve. The position states are managed by the submodules.

Analyzing the module view using dependency types from Table 1 as a guide, Table 3 captures dependencies among the modules of the SMS.

D. Enhanced Dependencies

Fig. 3 shows a data-model view for the SMS that addresses Scenario 3. After start-up and completion of any position change command, the *DesiredPositionState* and *CommandedPositionState* of the PCS and the *ActualPosition State* of the stepper motor must be identical.

Based on Fig. 3, Table 4 captures dependencies among the modules of the SMS.

Fig. 4 shows a component-connector view of the SMS that addresses Scenarios 4, 5, and 6. This view uses a UML sequence diagram augmented with stereotypes indicating the use of the Architecture Analysis and Design Language (AADL) process and thread concepts for the components. These concepts have well-defined execution and communication timing semantics [14]. They are relevant to address end-to-end latency and latency jitter of the data being processed by SMS.

PCS_APP is a process that periodically sends position change commands. All step counts in a command sequence for a position change, except for the last non-zero step, must

TABLE 4. DEPENDENCIES BASED ON THE DATA-MODEL VIEW OF THE SMS

Dependency	Description
Type: Aggregate (A) **Source**: DesiredPositionState **Target**: CommandedPositionState	*DesiredPositionState* must be identical to *CommandedPositionState* after startup and after every completion of a position change command.
Type: Aggregate (A) **Source**: DesiredPositionState **Target**: ActualPositionState	*DesiredPositionState* must be identical to *ActualPositionState* after startup and after every completion of a position change command.
Type: Aggregate (A) **Source**: CommandedPositionState **Target**: ActualPositionState	*CommandedPositionState* must be identical to *ActualPositionState* after startup and after every completion of a position change command.

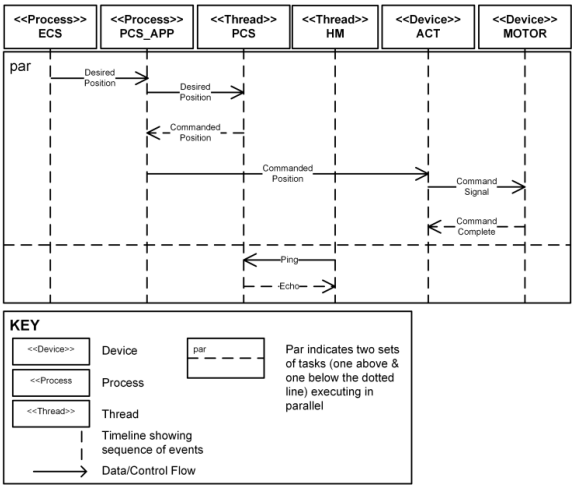

Fig. 4. Component-connector view of the SMS architecture.

equal *MaxStepCount*; all out-of-range commands are ignored. ACT immediately responds to commands received from PCS_APP and does not buffer any incoming commands. The previous command execution is aborted if it has not completed before the new command arrives. The number of steps within a command is translated into equivalent electrical command signals and completed within one frame. MOTOR reacts to the command signals received from ACT and indicates the completion of a step execution.

The core logic of the process PCS_APP runs on a thread PCS. It spawns another thread, health monitor (HM), that periodically monitors the health of the PCS thread to ensure that it is alive.

TABLE 5. DEPENDENCIES BASED ON THE COMPONENT-AND-CONNECTOR VIEW OF THE SMS

Dependency	Description
Type: Data (D) **Source**: ACT **Target**: PCS_APP	ACT receives position change command sequence from PCS_APP.
Type: Data (D) **Source**: MOTOR **Target**: ACT	MOTOR receives step execution command signals from ACT.
Type: Sequence of Control (S) **Source**: PCS_APP **Target**: ACT	ACT must complete execution of a command within a single frame.
Type: Sequence of Control (S) **Source**: ACT **Target**: PCS_APP	A new command from PCS_APP should not arrive before ACT has completed the previous command. New commands should only arrive at a rate that ACT can process.
Type: Sequence of Control (S) **Source**: ACT **Target**: MOTOR	MOTOR must maintain an execution rate of *MaxStepCount* per frame.
Type: Sequence of Control (S) **Source**: ACT **Target**: MOTOR	ACT receives a step execution completion signal from MOTOR.
Type: Location (L) & Physical Resource Behavior (P) **Source**: HM **Target**: PCS	HM preempts PCS periodically to check if it is functioning properly.

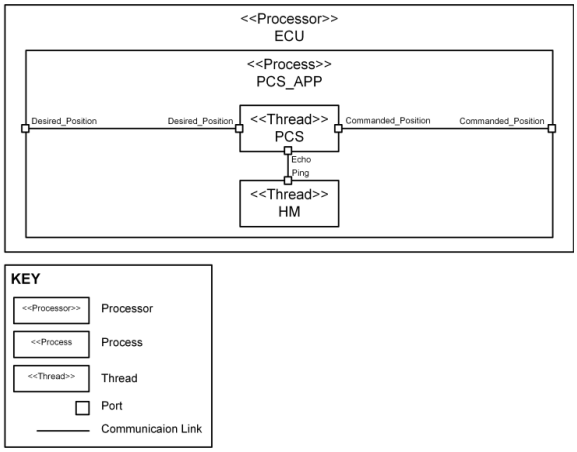

Fig. 5. Deployment view of the SMS architecture.

Based on Fig. 5, Table 5 captures dependencies among the different components of the SMS.

The timing behavior of threads and their communication are affected by how they are executed on processors and how they communicate over buses and networks. Therefore, we also consider the deployment view. Fig. 5 shows the deployment of PCS_APP on an electronic control unit (ECU).

Fig. 5 shows that the process PCS_APP consists of two threads; PCS is used to execute the core logic of the PCS_APP, and HM is used to periodically monitor the health of PCS. HM has a higher priority than PCS, so HM can preempt PCS when scheduled to run.

Table 6 captures the dependencies among the components of the SMS shown in Fig. 5.

We next use a DSM to represent and analyze the baseline and enhanced set of dependencies captured in Tables 3–6.

IV. ANALYSIS AND DISCUSSION

We chose DSMs to present our proposed analysis approach for architecture dependencies. While there are other possibilities, such as graph-based dependency analysis, DSMs offer a succinct representation of dependencies that can be easily clustered to understand their impact [1][15]. DSMs have been used to manage dependencies of software code artifacts and in the context of systems engineering [2]. They have been used to trace requirements to design in support of managing iterations [16]. And they have shown

TABLE 6. DEPENDENCIES BASED ON THE DEPLOYMENT VIEW OF THE SMS

Dependency	Description
Type: Location (L) **Source**: HM **Target**: PCS	HM and PCS run within the same process and ECU.
Type: Physical Resource Behavior (P) **Source**: HM **Target**: PCS	HM has a higher priority than PCS, so it preempts PCS when it is scheduled to run.

promising results when applied to managing dependencies at the architecture level [17].

A. Representing Multi-view Dependencies

Fig. 6 captures dependencies among the architectural elements for the SMS architecture. Fig. 6a depicts dependencies based entirely on the module view shown in Fig. 2. The table row for each module shows its dependent modules, and the columns show what other modules it depends on. The cells indicate the type of dependency based on those captured in Table 3.

Partitioning algorithms [15] can be used to produce a matrix in which the rows and columns are ordered so that items use only other items below or to their right. In the absence of cyclic dependencies, all items would produce a lower triangular matrix; any dependencies above the matrix diagonal indicate a cyclic dependency between the modules.

Within the matrix, a value in a cell can be used to indicate the number and types of dependencies from the column item to the row item. Therefore, when a module's row is populated with a large number of dependencies, this indicates a module that is heavily used by many other items (afferent coupling). In turn, when a modules's column is populated by a large number of dependencies, this indicates that a module uses many other items (efferent coupling).

Fig. 6b augments the dependencies based on additional views of the architecture described in Figs. 3, 4, and 5. Since the DSM shows the modules of SMS, we mapped elements from other views such as process PCS_APP and thread PCS to the module PCS, thread HM to module HM, device ACT to module ACT, and device MOTOR to module MOTOR. We accordingly converted the dependencies among these elements to dependencies among the mapped modules.

	PCS	DPS	CPS	ACT	StepsToDo	MOTOR	APS	HM
PCS	.			CD				
DesiredPositionState (DPS)	D	.						
CommandedPositionState (CPS)	D		.					
ACT				.		CD		
StepsToDo				D	.			
MOTOR						.		
ActualPositionState (APS)						D	.	
HM								.

(a)

	PCS	DPS	CPS	ACT	StepsToDo	MOTOR	APS	HM
PCS	.			CDS				L
DesiredPositionState (DPS)	D	.	A				A	
CommandedPositionState (CPS)	D	A	.				A	
ACT	S			.		CD		
StepsToDo				D	.			
MOTOR				S		.		
ActualPositionState (APS)		A	A			D	.	
HM	LP							.

(b)

Fig. 6. DSM based on (a) module view and (b) multiple views.

As is evident from Fig. 6b, several dependencies emerge when we consider additional views:

- Aggregate (A) dependencies emerge among modules *DesiredPositionState*, *CommandedPositionState*, and *ActualPositionState* from the data-model view in Fig. 3, indicating that at startup completion and at command completion, the actual position of the stepper motor must be the same as the position commanded by the ECS.
- Sequence of Control (S) dependencies emerge among modules MOTOR, ACT, and PCS from the view in Fig. 4, indicating that commands must be completed within certain time constraints.
- Location (L) and Physical Resource Behavior (P) dependencies emerge between modules HM and PCS, shown in Fig. 5. They are co-located and share CPU resources, implying that they could slow each other down.

The approach is fairly lightweight, yielding information useful for capturing and communicating different types of interrelationships among the elements. Identifying such dependencies based on multiple views is a significant input to change impact analysis, as we will see in the next section.

B. Dependency Impact Analysis

We use the DSMs to understand the implications of changes needed to achieve the end-to-end concerns of managing safety-critical testing, latency, and cost of change.

1) Safety-Critical Testing: Dependency models aid in determining what parts of the system should undergo retesting when a change is made. Fig. 7 shows a clustering of the DSM based on how interdependent the modules are.

As can be seen from the clustering of the DSMs, all modules appear less interdependent in the module view-based DSM in Fig. 7a than in the multiple views-based DSM in Fig. 7b. A retesting strategy based on Fig. 7a is, therefore, likely to test far fewer modules than may be necessary. For example, if a change is made to module ACT, Fig. 7a would suggest testing its dependent module MOTOR. However, Fig. 7b suggests also including HM and PCS since they are part of the same cluster of highly interdependent modules. Such analysis is beneficial during evaluation of recertification costs when changes are made to various parts of a safety-critical system.

2) Propagating Faults: Dependency models aid in determining faults likely to propagate within the system if the concerns associated with these dependencies are not completely addressed when designing the architecture (Table 7). The safety information is based on the Error Model Annex of AADL [11], which provides the capability to specify faults within the architecture and their impact on other components.

The newly discovered dependencies that led to the enhanced model are also the ones related to the satisfaction of Scenarios 3 through 6 (Table 2), which deal with the end-to-end latency of command completion by the SMS. Hence, the multi-view dependency model provides input to this type of fault propagation analysis for determining satisfaction of timing concerns, which could otherwise be missed.

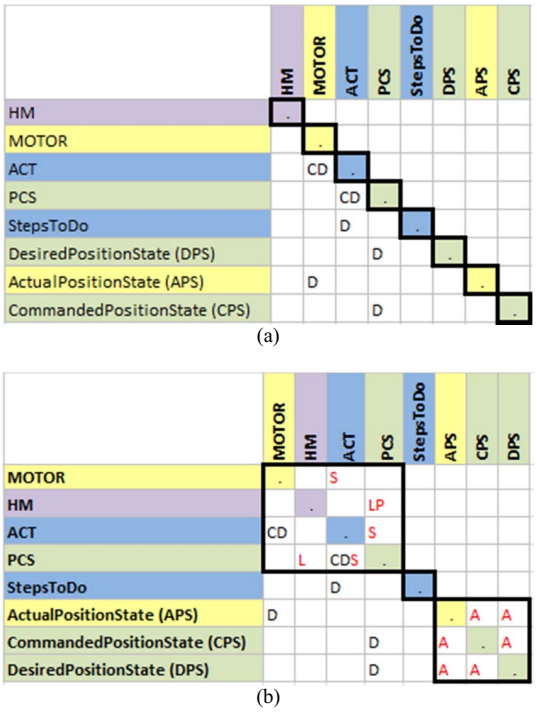

Fig. 7. (a) Module view-based DSM and (b) multiple views-based DSM with clusters of interdependent modules shown by dark borders.

3) Cost of Change: Dependency models aid in managing change. Should the existing architecture be evolved to satisfy the change scenarios, these additional dependencies can be used to determine the modules that a change may affect. We show these dependencies among the change scenarios for SMS and the corresponding modules that are impacted using a multi-domain matrix (MDM) [16] in Fig. 8. The MDM shows dependencies among the scenarios (top-left quadrant), the modules (bottom-right quadrant), and the scenarios and modules that satisfy them (top-right quadrant).

Analyzing Scenario 3, the MDM indicates that four modules are directly affected. The DSM based on the module view alone (Fig. 6a) shows there is no propagation of changes. It implies that making a change is localized to each of the four modules directly responsible for satisfying this requirement. The DSM based on multiple views (Fig. 6b) shows the semantic coherence among the three modules for position state through the aggregation dependency. We capture this distinction in Fig. 9.

The model in Fig. 9b shows that *DesiredPositionState, CommandedPositionState,* and *ActualPositionState* are more tightly coupled, as indicated by the aggregation dependency. PCS updates its *CommandedPositionState* immediately after directing ACT, assuming that the commanded number of steps will actually be executed by ACT and MOTOR. Although MOTOR signals ACT every time it completes the execution of a step, ACT can time out if it does not receive the signal in a timely manner. When a command sequence is complete, *DesiredPositionState, CommandedPositionState,* and *ActualPositionState* should have the same values, but nothing in the architecture guarantees this. Developers must

TABLE 7. DEPENDENCIES AND POTENTIAL FOR PROPAGATING FAULTS

Dependency	Potential Fault
Type: Aggregate **(A)** **Source**: DesiredPositionState **Target**: ActualPositionState **Description**: Must be identical after startup and after every completion of a position change command.	**Type**: Value Error **Description**: No guarantee they will be the same.
Type: Aggregate **(A)** **Source**: CommandedPositionState **Target**: ActualPositionState **Description**: Must be identical after startup and after every completion of a position change command.	**Type**: Value Error **Description**: No guarantee they will be the same.
Type: Sequence of Control **(S)** **Source**: PCS_APP **Target**: ACT **Description**: ACT must complete execution within a single frame.	**Type**: Timing Error **Description**: ACT does not guarantee complete execution of a command within a single frame.
Type: Sequence of Control **(S)** **Source**: ACT **Target**: PCS_APP **Description**: A new command from PCS_APP should not arrive before ACT has completed the previous one.	**Type**: Timing & Rate Error **Description**: ACT does not buffer commands and aborts processing the current command to respond to the newly arrived command.
Type: Sequence of Control **(S)** **Source**: ACT **Target**: MOTOR **Description**: MOTOR must maintain an execution rate of *MaxStepCount* per frame.	**Type**: Timing Error **Description**: MOTOR does not guarantee this execution rate.
Type: Sequence of Control **(S)** **Source**: ACT **Target**: MOTOR **Description**: ACT receives a step execution completion signal from MOTOR.	**Type**: Timing Error **Description**: ACT times out if it does not receive a completion signal from MOTOR in a timely fashion.
Type: Location **(L)** **Source**: HM **Target**: PCS **Description**: HM and PCS run within the same process and ECU.	**Type**: Process Failure **Description**: Failure of HM may impact the correct functioning of PCS since they are co-located threads within the same process.
Type: Physical Resource Behavior **(P)** **Source**: HM **Target**: PCS **Description**: HM has a higher priority than PCS, so it preempts PCS when scheduled to run.	**Type**: Timing Error **Description**: HM preempts PCS periodically and, therefore, can slow down PCS.

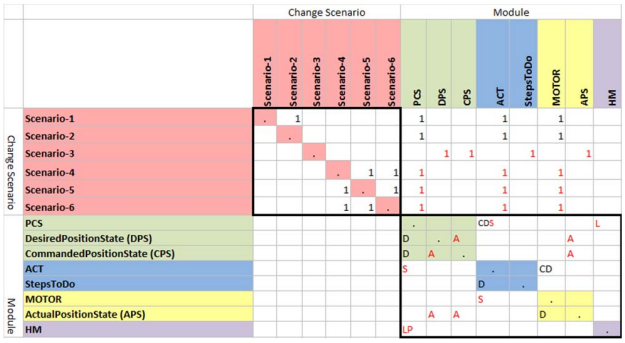

Fig. 8. MDM showing dependencies among requirements and the architectural elements that must satisfy them.

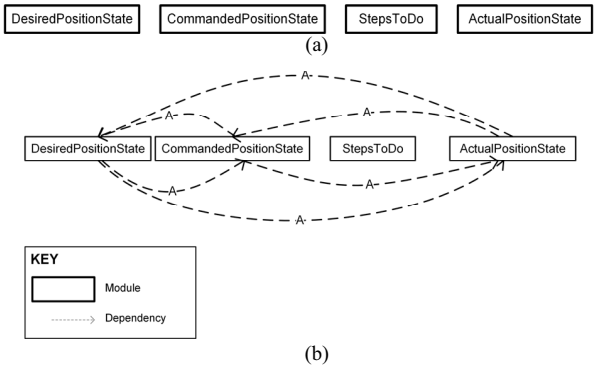

Fig. 9. Dependency model of Scenario 3 constructed using DSM based on (a) module view and (b) multiple views. Labels on the edges show the type of dependency as described in Table 1.

coordinate a change to the architecture to satisfy Scenario 3, since a change to one module is not necessarily localized and may propagate to the other two modules.

Analyzing Scenarios 4, 5, and 6, the MDM indicates that three modules are directly affected. The DSM based on the module view (Fig. 6a) shows that control and data flow propagate change among seven different modules. The DSM based on multiple views (Fig. 6b) shows that these modules are much more tightly coupled. We capture this distinction in Fig. 10. Comparing the two models shows that PCS, ACT, and MOTOR are more tightly coupled through this sequence of flow dependency. The comparison also shows that an additional module, HM, should be considered. Colocation of threads and their effect on the completion time of PCS due to preemption indicate that Scenarios 4, 5, and 6 are not completely satisfied by the architecture. Before developers make a change to the architecture to satisfy these scenarios, they should consider the ripple effects of these additional dependencies.

V. Discussion

Our study included working with the development team while they undertook the refactoring effort for the system. As they did a trade-off analysis to find the best solution to change scenarios (Table 2), their approach was influenced by the existing tools that segregate the dependencies differently than the approach we have presented in the architectural multi-view dependency analysis. Existing tools focus largely on data and control coupling, which limits the view of the system to that of Fig. 7a, showing modules that are fairly decoupled with no critical interfaces. Yet the issues that were most important to the refactoring of the safety-critical system required the type of analysis based on the view in Fig. 7b.

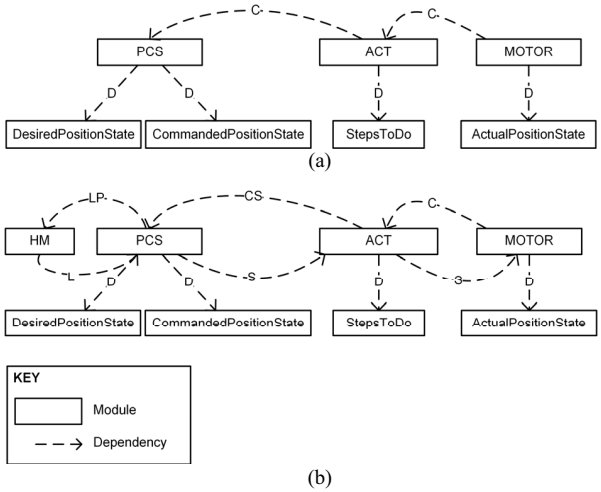

Fig. 10. Dependency model of Scenarios 4, 5, and 6 constructed using DSM based on (a) module view and (b) multiple views. Labels on the edges show the type of dependency as described in Table 1.

When exposed to multi-view dependency analysis, the development team found that the approach provided a more focused input to change impact analysis. It engaged them much more effectively to find critical problems by exposing dependencies such as those listed in Table 1 and pushed them to think beyond data and control flow. The architect summarized the team's take-away as follows:

> "This was an architectural problem where two separately developed systems [ECS and SMS] were coming together and their interface was not properly defined. Independently, each development team did the correct thing. It was at the architectural level that the failure was introduced and found. The interesting aspect is, when you focus on the internal dependencies of the systems and their inputs and outputs, we were doing the correct thing against the requirements. Multi-view dependency analysis says there are a whole lot of other relationships. The traditional approach of the existing tools segregates the dependencies differently than the architectural multi-view dependency analysis approach. This forces people to go beyond the traditional view where, if there is no data coupling, then there is no critical dependency.
>
> The 'aha' moment for me was not that this is how we can solve the problems, but rather this could help engage people to find the problems before they can become problems. The dependency table [Table 1] becomes an input; stop thinking about data and control flow only; there is other stuff too."

The approach also improved the quality of the development team's change impact analysis by using a unified model to address dependency dimensions that determine the impact of module and architectural changes on other modules and on the dependency structure. It fit within their goals of safety-critical development because it is conservative: it found more dependencies with potential critical impact than fewer. The architect alluded to the fact that they used the dependency analysis approach later:

> "There is no room for additional modeling or analysis. Each new model increases the complexity of keeping them all in sync non-linearly—probably closer to exponentially. Ideally, we would only ever have a single model of the system so that we don't have to worry and maintain disparate versions/views of the same thing. If any approach will work, it cannot be augmenting existing processes by adding new tools and models, but it should be replacing existing tools. Today, rather than more tools, more concrete, lightweight approaches are more useful. In fact, we used the dependency table [Table 1] after the fact to avoid other issues going forward. Going through major components and categorizing them as low and high risk using these dependency types as a starting point break down the barrier and increase adoption possibilities of starting to think architecturally."

VI. Related Work

Previous research has focused on dependency analysis to assist analyzing maintainability, modifiability, change impact, traceability, security, and testing. Existing practices and tools, however, focus largely on an implementation-based module view of the system, which not only limits the kinds of dependencies that can be analyzed but also neglects trade-offs [5]. Our previous experience corroborates this. For example, we conducted a comparative stability analysis on the module view of a safety-critical system that focused on data and control dependencies. The refactored system showed a slight deterioration in stability compared to its predecessor, giving the impression that the effort was not cost effective. It was only when the refactored effort was analyzed with respect to the safety-critical testing levels of the components and mapped back to the implementation elements that the trade-offs were properly understood [12].

It is widely recognized that to get a holistic understanding of a system's quality, a multi-view perspective is essential [7][18]. Progress has been limited because industry is mostly driven to use tools that map closely to the quality concerns of highest priority [19] [20]. Research has focused on improving documentation approaches driven by the limited capabilities of existing documentation tools and approaches for cross-referencing related information about a system [21].

Existing work emphasizes the need to better manage dependencies. Studies include understanding dependencies to better manage architecture modifiability and maintainability [5][22], to advance safety-critical testing and modeling [12] [23], to understand build and deployment-time issues [24], and to improve system and architecture visibility [17][25] [26]. Challenges uncovered include the need to add dependencies manually [27][28], augment the information in a dependency matrix [8], or use run-time information while evaluating a system [29]. Tools also exist for evaluating quality attributes related to run time (e.g., by evaluating network performance and latency [30] or system schedulability [10]) using a dedicated representation.

All these approaches, however, require establishing a new model of the system, which is not only time consuming but also costly as engineers must be trained to use disparate modeling languages with different semantics and must manually maintain their consistency. Moreover, they do not start with a clear understanding of semantics of architectural dependencies. Previous research has uncovered the lack of such common semantic understanding of different architectural dependencies in software development [5][27].

Using a DSM approach to visualize dependencies has become an integral part of mainstream architectural tools [18]. DSMs are single-domain square matrices, meaning that relations are defined between instances of the same type (e.g., software modules). To reach deeper conclusions about inter- and intra-domain dependencies in a dual-domain context, such as between different architectural elements and requirements, an MDM representation is needed [1]. The application of MDM to software analysis to date has been limited to mapping requirements to modularity concerns based on information from the module view [22][31].

VII. Conclusions

We presented a case study of a safety-critical system that motivated the need for an analysis approach to model architecturally significant dependencies from module, run-time, and deployment views of the system. Through our engagement with this study, we demonstrated the following:

- Identifying key multi-view dependencies and mapping them to a module view of elements and dependencies allow developers to concretely assess the impact of change and recognize system elements that must be developed further.
- Without such a model to guide the identification of well-defined dependencies, the current concerns and tools at hand drive the focus of analysis. We observed that the emphasis of model-driven engineering tools on state transitions encourages engineers to focus on data flow and events. As a result, they miss data entity relationships, resource behavior, and deployment-related dependencies.
- The modeling approaches and tools used by developers often do not identify all dependencies. Support for assisting developers to easily extract and monitor key dependencies that cause ripple effects in multiple aspects of a system (e.g., in ability to track run-time performance, propagation of faults, and cost of change) is essential. Lightweight, semantically well-defined techniques have a greater possibility of providing a focused analysis context. Our dependency guide is a first cut at this.

This study is based on a real system and its artifacts. There is a possible threat to internal validity due to variables that emerge from the developers' experience with developing control systems and using their tools, which could have caused some of the issues identified. We minimized the impact of this by extracting the aspects of the stepper motor system that are generalizable to open-loop control systems. The stepper motor is a fairly common subsystem in control systems, representative enough for the dependencies we captured; hence, it poses minimal threats to external validity.

Our study reveals an evolving strategy to provide more situated information for architecture decision making during development using dependency analysis to complement model-based approaches. The architect of the project put the real challenge for future work in his words as follows:

> "A common language to discuss key design dependencies is something keenly lacking among developers—many of whom have no architectural training or experience. Additionally, 'software architecture' is an overloaded term in practice. For instance, in *Software Considerations in Airborne Systems and Equipment Certification (DO-178B)* [32], architecture is defined basically as allocation between requirements and design components. While this isn't exactly wrong, it certainly isn't complete and makes my life difficult dealing with multiple meanings of the term as I discuss it with the teams."

Despite an ample amount of research in architectural analysis and dependency management, these techniques are still not at a maturity level that can be used at ease in practice. Researchers should take this reality into consideration in designing their research questions.

Introducing yet another tool to industry environments is a hard sell unless it is proven to replace and improve other tools. However, techniques that supplement existing development approaches, such as a common definition and understanding of architectural dependencies, can offer improvements. As a community, we should strive to improve the semantic accuracy of architectural dependencies implied by a multi-view perspective.

ACKNOWLEDGMENT

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. References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute. [Distribution Statement A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution. DM-0003205.

We thank Tamara Marshall-Keim for her feedback and expert input.

REFERENCES

[1] J. Bartolomei, M. Cokus, J. Dahlgren, R. de Neufville, D. Maldonado, and J. Wilds, Analysis and Application of Design Structure Matrix, Domain Mapping Matrix, and Engineering System Matrix Frameworks. Cambridge: Massachusetts Institute of Technology, 2007.

[2] T. R. Browning, "Applying the design structure matrix to system decomposition and integration problems: a review and new directions," IEEE T. Eng. Manage., vol. 48, pp. 292–306, Aug. 2001.

[3] Esterel Technologies. SCADE Suite. http://www.esterel-technologies.com.

[4] Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) – System and Software Quality Models, ISO/IEC 25010:2011. Geneva, Switzerland: ISO/IEC, 2001.

[5] Z. Li, P. Liang, P. Avgeriou, N. Guelfi, and A. Amaptzoglou, "An empirical investigation of modularity metrics for indicating architectural technical debt," in Proc. 10th Int. ACM SIGSOFT Conf. Quality of Software Architectures. New York: ACM, 2014, pp. 119–128.

[6] D. Tofan, M. Galster, and P. Avgeriou, "Difficulty of architectural decisions: a survey with professional architects," Lect. Notes Comput. Sc., vol. 7957, pp. 192–199, 2013.

[7] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, et al., Documenting Software Architectures: Views and Beyond, 2nd ed. Upper Saddle River, NJ: Addison-Wesley, 2011.

[8] H. Koziolek, "Sustainability evaluation of software architectures: a systematic review," in Proc. 7th Int. ACM/SIGSOFT Conf. Quality of Software Architecture. New York: ACM, 2011, pp. 3–12.

[9] P. Feiler, L. Wrage, and J. Hansson, "System architecture virtual integration: a case study," presented at the Embedded Real-Time Software and Systems Conference. Toulouse, France, May 2010.

[10] S. Li, F. Singhoff, S. Rubini, and M. Bourdellès, "Applicability of real-time schedulability analysis on a software radio protocol," ACM SIGAda Lett., vol. 32, pp. 61–68, Dec. 2012.

[11] SAE Architecture Analysis and Design Language (AADL): As-2c Architecture Analysis and Design Language (AS5506/1). http:// standards.sae.org/as5506/1/

[12] R. L. Nord, I. Ozkaya, R. S. Sangwan, and R. J. Koontz, "Architectural dependency analysis to understand rework costs for safety-critical systems," in ICSE Companion. New York: ACM, 2014, pp. 185–194.

[13] R. S. Sangwan and C. J. Neill, "How business goals drive architectural design," Computer, vol. 40, pp. 85–87, Aug. 2007.

[14] P. Feiler and D. Gluch, Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Upper Saddle River, NJ: Addison-Wesley, 2012.

[15] U. Lindemann, M. Maurer, and T. Braun, Structural Complexity Management: An Approach for the Field of Product Design. Berlin: Springer, 2010.

[16] S. Kortler, B. Helms, M. Berkovich, U. Lindemann, K. Shea, J. M. Leimeister, et al., "Using MDM-methods in order to improve managing of iterations in design processes," 12th Int. Dependency and Structure Modelling Conf. Cambridge, UK, July 2010.

[17] C. Hinsman, N. Sangal, and J. Stafford, "Achieving agility through architecture visibility," in Proc. 5th Int. Conf. Quality of Software Architectures: Architectures for Adaptive Software Systems. Berlin: Springer, 2009, pp. 116–129.

[18] A. Telea, L. Voinea, and H. Sassenburg, "Visual tools for software architecture understanding: a stakeholder perspective," IEEE Software, vol. 27, pp. 46–53, Nov./Dec. 2010.

[19] B. Hailpern and P. Tarr, "Model-driven development: the good, the bad, and the ugly," IBM Syst. J., vol. 45, pp. 451–461, July 2006.

[20] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang, "What industry needs from architectural languages: a survey," IEEE T. Software Eng., vol. 39, pp. 869–891, May 2013.

[21] A. Tang, P. Liang, and H. van Vliet, "Software architecture documentation: the road ahead," in Proc. 9th Working IEEE/IFIP Conf. Software Architecture. Washington, DC: IEEE, 2011, pp. 252–255.

[22] K. Sethi, Y. Cai, S. Wong, A. Garcia, and C. Sant'Anna, "From retrospect to prospect: assessing modularity and stability from software architecture," in Proc. Joint 8th Work. IEEE/IFIP Conf. Software Architecture and 3rd European Conf. Software Architecture. Washington, DC: IEEE, 2009, pp. 269–272.

[23] F. Cadoret, E. Borde, S. Gardoll, and L. Pautet, "Design patterns for rule-based refinement of safety critical embedded systems models," presented at the Int. Conf. Eng. Complex Comput. Syst. Paris, France, July 2012.

[24] J. D. Morgenthaler, M. Gridnev, R. Sauciuc, and S. Bhansali, "Searching for build debt: experiences managing technical debt at Google," in Third Int. Workshop Manag. Tech. Debt. Washington, DC: IEEE, 2012, pp. 1–6.

[25] A. MacCormack, J. Rusnak, and C. Y. Baldwin, "Exploring the structure of complex software designs: an empirical study of open source and proprietary code," Manage. Sci., vol. 52, pp. 1015–1030, July 2006.

[26] R. S. Sangwan and C. J. Neill, "Characterizing essential and incidental complexity in software architectures," in Proc. Joint 8th Work. IEEE/IFIP Conf. Software Architecture and 3rd Eur. Conf. Software Architecture. Washington, DC: IEEE, 2009, pp. 265–268.

[27] T. Callo, P. van der Spek, and P. Avgeriou, "A practice-driven systematic review of dependency analysis solutions," Empir. Softw. Eng., vol. 16, pp. 544–586, Oct. 2011.

[28] C. Riva, P. Selonen, T. Systa, and X. Jianli, "UML-based reverse engineering and model analysis approaches for software architecture maintenance," in Proc. 20th IEEE Int. Conf. Software Maintenance. Washington, DC: IEEE, 2004, pp. 50–59.

[29] H. Koziolek, B. Schlich, and C. Bilich, "A large-scale industrial case study on architecture-based software reliability analysis," in IEEE 21st Int. Symp. Software Rel. Eng. Washington, DC: IEEE, 2010, pp. 279–288.

[30] D. Sharman and A. Yassine, "Characterizing complex product architectures," Syst. Eng. J., vol. 7, pp. 35–60, Mar. 2004.

[31] S. Wong, Y. Cai, M. Kim, and M. Dalton, "Detecting software modularity violations," in Proc. 33rd Int. Conf. Software Engineering. New York: ACM, 2011, pp. 411–420.

[32] Software Considerations in Airborne Systems and Equipment Certification (DO-178B). Washington, DC: RTCA, Inc., and Malakoff, France: EUROCAE, 1992.