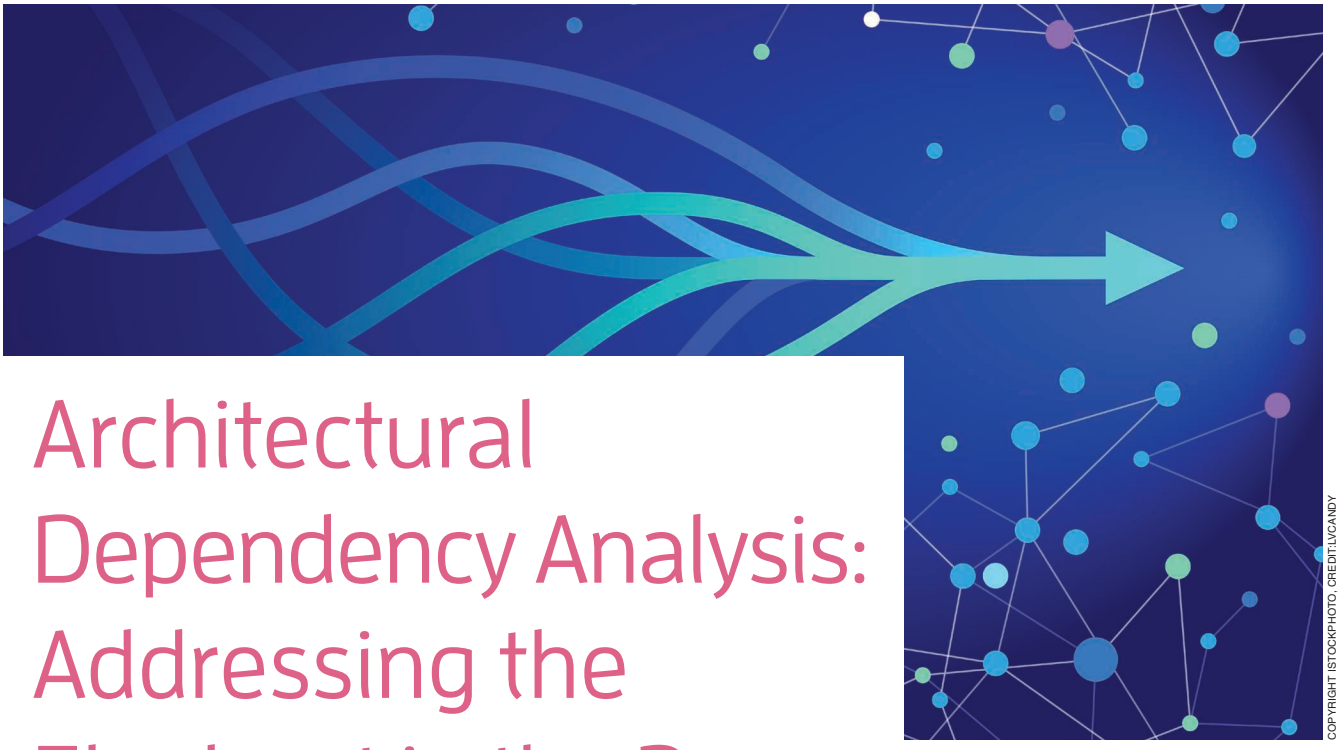
EDITOR **PHIL LAPLANTE**
Penn State, plaplante@psu.edu



Architectural Dependency Analysis: Addressing the Elephant in the Room

Raghvinder S. Sangwan, Pennsylvania State University
Robert L. Nord and Ipek Ozkaya, Carnegie Mellon University

Dependency analysis is typically limited to the static analysis of code structures. We applied this practice to safety-critical systems that were re-engineered to reduce safety testing and technology upgrade cost. We discuss the need for a well-defined description of architectural dependencies to address the observed gaps.

Several tools have been developed to help manage software evolution and maintenance and assist with architecture conformance and quality analysis. Most advances in automation and tooling have been variations on providing static analysis metrics at the level of code, object-oriented design, and module view of an architecture, although some have focused on version control history to understand the impact of change or discover architecture smells.[1,2] Our experience shows that a multiview architectural dependency analysis approach is more effective in conducting change impact analysis than approaches limited to the module view of an architecture.[3,4] The focus on syntactic dependencies related to data and control flow in a module view precludes dependencies from other perspectives, such as those that express semantic transformation associated with performance, availability, safety and testing (for instance, understanding the impact of splitting a module or using a module as a standard interface

Digital Object Identifier 10.1109/MC.2021.3050538
Date of current version: 12 March 2021

Most advances in automation and tooling have been variations on providing static analysis metrics at the level of code, object-oriented design, and module view of an architecture.

intermediary that provides a generalized publish-subscribe capability between modules to improve performance, availability, and safety while reducing cost related to safety-critical testing).

A CASE OF TWO SYSTEMS

Consider a general-purpose shared computing resource architecture.[3] The use of shared resources, such as memory, processing, display graphics, network communication, and storage in safety-critical systems, requires the careful balancing of performance and safety qualities. Applications that access a shared computing resource must be partitioned based on their function, their interface to the resource, and how safety critical they are. Figure 1(a) depicts an instance of such an architecture.

The module view in Figure 1(a) conveys the principal units of implementation, and the component-and-connector (C&C)

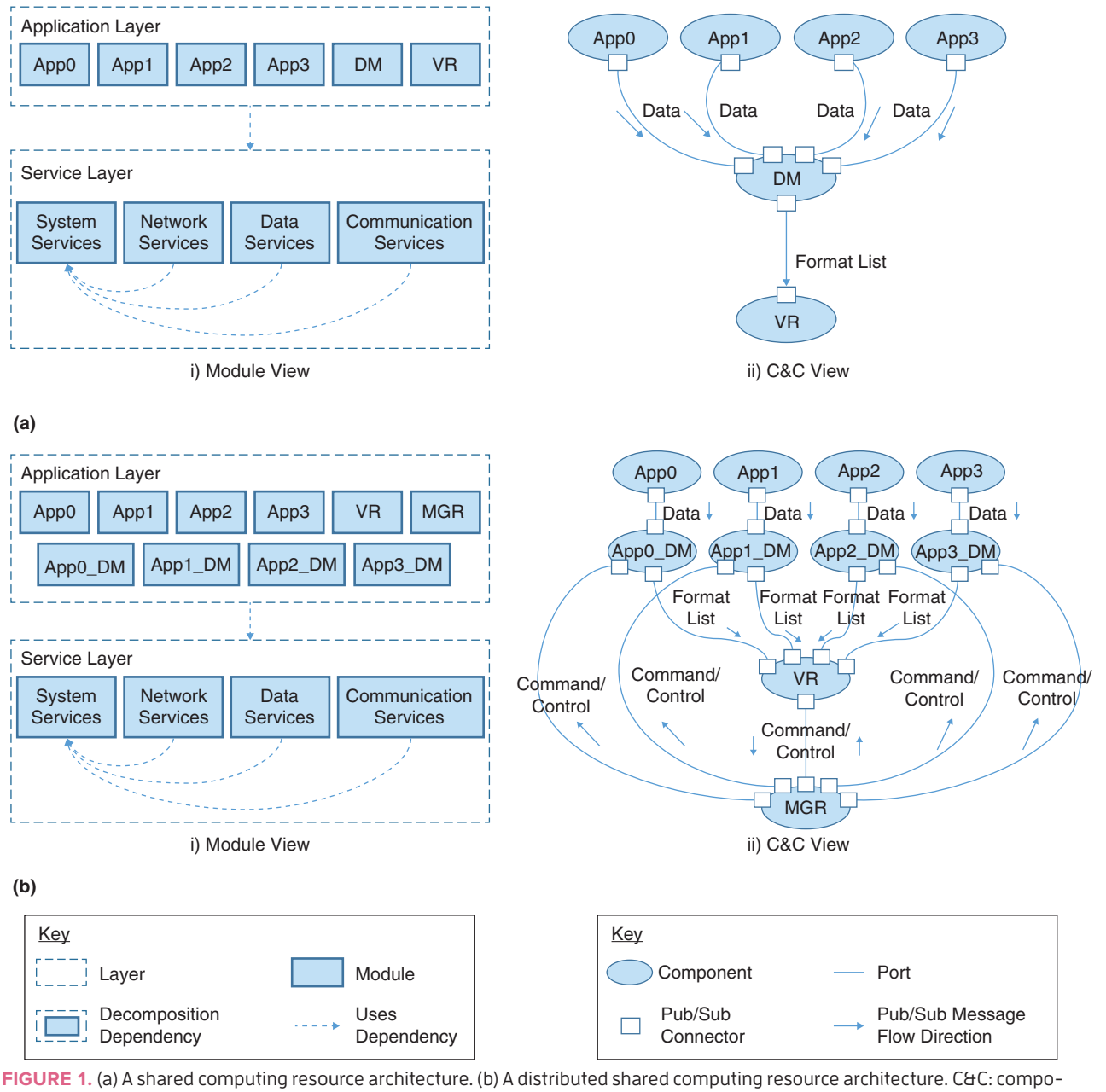


FIGURE 1. (a) A shared computing resource architecture. (b) A distributed shared computing resource architecture. C&C: component-and-connector; App: application; DM: data mover; VR: virtual resource; MGR: manager.

view illustrates how these units interact at runtime. Together, these views show the application layer consists of multiple functional applications (App0, App1, App2, and App3) that send their data to a separate centralized data mover (DM). The DM transforms application data into a specialized resource-specific format. The DM then forwards the data to a virtual resource (VR), which interfaces with the physical resources (for example, storage, network, display). The modules in the application layer use one or more services (system, network, data, or communication) from the service layer during the course of their interactions.

While the architect had designed this architecture appropriate for the time, as the system evolved, adding new applications required more effort. With the DM transforming all application data to formats for disparate resources, including those that were safety critical, when a new resource-specific format for a new application was added or an existing one changed (whether that resource was safety critical or not), the entire DM was required to be tested to the highest safety-critical level. Additionally, as a monolith with interdependent functionality, the DM could not isolate safety critical applications, which drove up the overall safety testing cost. Reaching this tipping point, the architect suggested a major change to the existing architecture. Figure 1(b) depicts a distributed shared computing resource architecture proposed by the architect.

The DM from the original architecture was decomposed into separate resource formatting clients, App0_DM, App1_DM, App2_DM, App3_DM, and a control-and-transform manager (MGR) that managed user input and coordinated the use of resources. App0_DM contained the safety-critical functionality and received its data from App0. Decomposing the DM in this manner allows safety-critical clients such as App0_DM to be isolated into their own user space partition apart from other software elements that could improve *testability* and drive down *cost* for safety-critical testing. Distributing responsibilities, however, would increase the number of publish-subscribe messages, which would require additional measures to manage network traffic and CPU utilization to offset any negative impact on performance. More importantly, however, coordinating the use of resources to process these messages introduces additional configuration management burden, which can affect testability and integrability.

So, is the proposed distributed architecture more effective at isolating safety-critical functionality and accommodating changes without increasing the effort and cost for safety-critical testing? Table 1 captures representative metrics that can be used to compare the ease of making changes to the original shared computing resource architecture and its newly proposed distributed version. These measures can be obtained through the static dependency analysis of the module view of a system using a static analysis tool.

Stability measures how much of the system is affected when a change is made, connectedness refers to reachability, coupling measures how strongly connected a system is, and system and intercomponent cyclicity measure undesirable coupling. Higher values of these measures indicate a system that is not very effective at handling change.

Stability measures how much of the system is affected when a change is made, connectedness refers to reachability, coupling measures how strongly connected a system is, and system and intercomponent cyclicity measure undesirable coupling.

Looking at the results in Table 1, it is difficult to say if the situation has improved. Since the results are mixed, there is no convincing evidence that the architect can present to the management or the development team to make an investment in the proposed distributed shared computing resource architecture.

A similar analysis on a safety-critical engine control system that manages fuel flow to an aircraft engine also highlighted these shortcomings.[4]

ANALYSIS USING ARCHITECTURAL DEPENDENCIES

The shortcoming of an analysis, such as the one in Table 1, is that it is based on the module view of a system that mostly captures data and control dependencies among its elements and is often insufficient for understanding overarching

TABLE 1. Dependency-based metrics for the original and distributed shared computing resource architecture.

Metric	Original	Distributed	Health*
Stability	53.06%	50.51%	–
System cyclicity	42.857%	42.857%	=
Intercomponent cyclicity	42.857%	0%	+
Connectedness	53.3%	53.3%	=
Coupling	16.48%	16.48%	=

*The ease of making changes is indicated as improved (+), deteriorated (–), or remained the same (=).

architectural issues within a system.[5] Many of the architectural design decisions and tradeoffs involve dependencies other than these two.[6] Elements of a system, for instance, may not be colocated on the same machine because resources drained by one may impact the performance of the other or failure of one may have an undesirable effect on proper functioning of the other. It may also be the case that certain elements being safety critical need to be separated from others to contain testing and certification costs. The literature lacks a clear description and collection of such architectural dependencies,[7] and support for analyzing these dependencies among key architectural decisions remains a gap in the industry.[8]

> More importantly, however, coordinating the use of resources to process these messages introduces additional configuration management burden, which can affect testability and integrability.

As we analyzed the two safety-critical systems described in the previous section, we compiled a list of frequently encountered architectural dependency types provided in Table 2. Using these dependency types, Figure 2 captures the dependency models for the original and the distributed shared resources architecture in a dependency structure matrix (DSM).

A DSM is a matrix where all elements of a system appear in both the rows and columns, and dependencies are signaled at their intersection points in the matrix. Given that the responsibilities within DM (App0_DM, App1_DM, App2_DM, App3_DM, and MGR) are not visible in any of the views of the original architecture in Figure 1(a), we assume the worst case—that they are fully connected—and we present them as such in Figure 2(a).

Several aspects of the DSMs provided are worth noting.

1. The DSMs use letters from the dependency table to distinguish the types of dependencies among the modules.
2. Control (C) and data (D) dependencies can be determined through static analysis, but the others can be easily missed.
3. An individual cell can be multivalued to indicate different types of dependencies among the modules involved in that relationship.
4. 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. The testing (T_a, T_c) dependency on the diagonal indicates the testing level for each component, and lower testing cost is achieved by separating safety-critical modules (tested at level A represented as T_a) from non-safety-critical modules (tested at level C represented as T_c).
5. The location (L), quality-of-service (Q), resource (R), and

TABLE 2. Architectural dependencies to reason about change impact.

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 runtime location of A must be consistent with the assumptions of B.
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 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.
T	Testing	To lower the overall testing cost, safety-critical aspects must be split into child modules that are separate from their non-safety-critical aspects of the parent module.
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.

sequence-of-control (S) 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.

Architectural dependencies provide additional information that can be used for a more refined analysis of the following:

1. *Safety-critical testing*: We can look at the criticality levels of the components along the diagonals of the DSMs in Figure 2 to understand the testing cost of the system. Level Ta is the strongest, requiring intensive code review and testing efforts. The code of components classified with this level shall be fully covered using the modified condition/decision coverage (MC/DC) method. Components classified at lower levels (such as Tc) need only be validated against the statement coverage method, which is less costly and time-consuming. Figure 2(a) illustrates Ta-level testing for all elements within the DM module. Figure 2(b) shows fewer elements requiring testing level Ta, indicating a marked improvement in the distributed architecture with respect to the cost and effort of testing.
2. *Propagating faults*: 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 Figure 2 offers a more accurate 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.
3. *Cost of change*: Should the existing architecture be refactored, the additional dependencies can

			VR	App0_DM	App1_DM	App2_DM	App3_DM	MGR	App0	App1	App2	App3	Network Services	Data Services	Comm. Services	System Services
Application Layer	VR		Tc													
	DM	App0_DM	D	Ta	D	D	D	D								
		App1_DM	D	D	Ta	D	D	D								
		App2_DM	D	D	D	Ta	D	D								
		App3_DM	D	D	D	D	Ta	D								
		MGR	D	D	D	D	D	Ta								
	App0			D					Ta							
	App1				D					Tc						
	App2					D					Tc					
	App3						D					Tc				
Platform Layer	Network Services		C	C	C	C	C	C	C	C	C	C	Tc			
	Data Services		C	C	C	C	C	C	C	C	C	C		Tc		
	Comm. Services		C	C	C	C	C	C	C	C	C	C			Tc	
	System Services		C	C	C	C	C	C	C	C	C	C	C	C	C	Tc

(a)

			VR	App0_DM	App1_DM	App2_DM	App3_DM	MGR	App0	App1	App2	App3	Network Services	Data Services	Comm. Services	System Services
Application Layer	VR		TcL-					DS								
	App0_DM		DQ	TaL-					L+							
	App1_DM		DQ		TcL-					L+						
	App2_DM		DQ			TcL-					L+					
	App3_DM		DQ				TcL-					L+				
	MGR		DS	DRS	DRS	DRS	DRS	TcL-								
	App0			DSL+					TaL-							
	App1				DSL+					TcL-						
	App2					DSL+					TcL-					
	App3						DSL+					TcL-				
Platform Layer	Network Services		C	C	C	C	C	C	C	C	C	C	Tc			
	Data Services		C	C	C	C	C	C	C	C	C	C		Tc		
	Comm. Services		C	C	C	C	C	C	C	C	C	C			Tc	
	System Services		C	C	C	C	C	C	C	C	C	C	C	C	C	Tc

(b)

FIGURE 2. DSMs showing dependencies among elements of the (a) shared computing resource architecture and (b) distributed shared computing resources architecture.

be used to determine the modules that a change may affect.

The visual representation of the safety-critical testing information on the DSM structure, while not quantified, provides additional information to compare the architectures before and after the evolution. Relying solely on metrics such as those in Table 1 proves to be insufficient and does not demonstrate that the refactoring enables reduction of testing and hence reduces overall life-cycle costs.

BUILDING ARCHITECTURAL DEPENDENCY ANALYSIS INTO SOFTWARE PROJECTS

When left unmanaged, architectural dependencies create cost overruns and degraded qualities in systems. Architecture dependency analysis in practice, however, is typically performed in retrospect, missing important dependencies that surface earlier in the development life cycle. Moreover, the tools used focus primarily on module structures and/or the runtime image that provide only a limited view of a system.

The motivational example of a shared resource architecture illustrates how existing dependency analysis tools fall short of demonstrating the benefits of the rearchitected systems and fail to capture multiple quality attribute tradeoffs when focusing primarily on module structure dependencies.

Our experience illustrates how identifying key multiview dependencies allows developers to concretely assess the impact of change and recognize system elements that must be developed further. Support for assisting developers to easily extract and monitor key dependencies that cause ripple effects in multiple aspects of a system (for example, testing, propagation of faults, and cost of change) is essential. Lightweight semantically well-defined techniques based on dependencies described in Table 2 have a greater possibility of providing a focused analysis context.

ACKNOWLEDGMENT

This material is based upon work funded and supported by the Department of Defense under contract FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. DM20-1188.

REFERENCES

1. T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Trans. Softw. Eng.*, vol. 31, no. 6, pp. 429–445, 2005. doi: 10.1109/TSE.2005.72.
2. R. Mo, Y. Cai, R. Kazman and L. Xiao, "Hotspot patterns: The formal definition and automatic detection of architecture smells," in *Proc. 2015 12th Working IEEE/IFIP Conf. Softw. Archit.*, Montreal, Canada, pp. 51–60. doi: 10.1109/WICSA.2015.12.
3. R. L. Nord, I. Ozkaya, R. S. Sangwan, and R J. Koontz, "Architectural dependency analysis to understand rework costs for safety-critical systems," in *Proc. Companion 36th Int. Conf. Softw. Eng. (ICSE Companion)*, 2014, pp. 185–194. doi: 10.1145/2591062.2591185.
4. R. L. Nord, R. S. Sangwan, J. Delange, P. Feiler, L. Thomas, and I. Ozkaya. 2016. "Missed architectural dependencies: The elephant in the room," in *Proc. 2016 13th Work. IEEE/IFIP Conf. Softw. Archit. (WICSA)*, pp. 41–50. doi: 10.1109/WICSA.2016.32.
5. H. Koziolek, "Sustainability evaluation of software architectures: A systematic review," in *Proc. Joint ACM SIGSOFT Conf.–QoSA ACM SIGSOFT Symp.–ISARCS Qual. Softw. Archit.–QoSA Archit. Crit. Syst.–ISARCS (QoSA-ISARCS '11)*, 2011, pp. 3–12. doi: 10.1145/2000259.2000263.
6. T. B. Callo Arias, P. Spek, and P. Avgeriou, "A practice-driven systematic review of dependency analysis solutions," *Empir. Softw. Eng.*, vol. 16, no. 5, pp. 544–586, 2011. doi: 10.1007/s10664-011-9158-8.
7. Z. Li, P. Liang, P. Avgeriou, N. Guelfi, and A. Ampatzoglou, "An empirical investigation of modularity metrics for indicating architectural technical debt," in *Proc. 10th Int. ACM Sigsoft Conf. Qual. Software Archit. (QoSA '14)*, 2014. pp. 119–128. doi: 10.1145/2602576.2602581.
8. D. Tofan, M. Galster, and P. Avgeriou, "Difficulty of architectural decisions: A survey with professional architects," in *Proc. 7th Euro. Conf. Softw. Archit. (ECSA'13)*, 2013, pp. 192–199. doi: 10.1007/978-3-642-39031-9_17.

RAGHVINDER S. SANGWAN is an associate professor of software engineering at the School of Graduate Professional Studies, Pennsylvania State University, Malvern, Pennsylvania, 19355, USA. Contact him at rsangwan@psu.edu.

ROBERT L. NORD is a principal member of the technical staff at the Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 15213, USA. Contact him at rn@sei.cmu.edu.

IPEK OZKAYA is a technical director of engineering intelligent software systems at the Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 15213, USA. Contact her at ozkaya@sei.cmu.edu.