The AADL Error Library: An Operationalized Taxonomy of System Errors

Sam Procter
Software Engineering Institute, Carnegie Mellon University
Pittsburgh, Pennsylvania
sprocter@sei.cmu.edu

Peter Feiler
Software Engineering Institute, Carnegie Mellon University
Pittsburgh, Pennsylvania
phf@sei.cmu.edu

ABSTRACT
The problem of how to best classify system errors has been a topic of research for years. In this paper, we present an established taxonomy that draws on a broad range of previous work in this area: the Architecture Analysis and Design Language’s (AADL) EMV2 Error Library. The error library is now part of an international standard and has been used in a range of systems and domains. In this work, we describe its features, including that: a) it is deeply integrated in a rich, semi-formal system modeling language (AADL); b) the errors it includes have formalized semantics; and c) it is designed to be easily extensible by system developers to become domain- or system-specific. We describe the original inspirations and prior work that informed the library’s design, document the error families that comprise the taxonomy, and discuss the library’s usage in an architecturally-integrated system assurance process.

ACM Reference Format:

1 INTRODUCTION
The challenge of classifying the way that things can go wrong in a component-based system is a difficult one: components—and the systems that rely on them—can fail in myriad, unpredictable ways. But it is nonetheless a challenge that should be addressed: these component-based, software-driven systems which are difficult to analyze are also used increasingly for safety-critical applications. Unfortunately, many well-established classifications and taxonomies of system errors are not what we term operationalized: directly usable in modern, model-based system engineering efforts. Rather, they were designed for human use on informal (often mental) system models: they are specified and described in natural language, rather than in any formal or semiformal specification language.

The Architecture Analysis and Design Language (AADL) is a well-established semiformal architecture specification language that is used in academia, industry [10], and is available as an international standard [23]. Various annexes have been developed that extend the core language to provide additional modeling features. One that has been thoroughly documented elsewhere [5, 16] is the error modeling annex, which is in its second version (abbreviated EMV2). What has not been addressed, however, is the rationale and design choices behind the EMV2 Error Library, which is the EMV2’s included error taxonomy. Though usable independently of AADL as a traditional, non-operationalized taxonomy, the error library is most commonly used to embed the error behavior of a system in an architectural model. It allows system designers to extend the notion of a component’s interface and architecture beyond the interface types (e.g., integer, floating point, etc.) and non-functional properties (e.g., timings, power consumption, etc.) to include behavior in the presence of errors and activated faults.

This paper presents the EMV2 error library, as well as its underlying concepts, inspirations, and impetuses. Specifically, the contributions that the library makes are that it provides:

(1) An ontology of system errors that is embeddable into system architecture models that have been specified in AADL. This ontology relates error types to one another through type extension, which enables modeling different layers of abstraction. Additionally, the types are instantiable into tokens, which flow through petri-net-like specifications of component error behavior.

(2) Formal specifications of the semantics of the error types in the library.

This paper, in describing the library, makes additional contributions by providing:

(1) a rationale for the organization of the library,
(2) an explanation of the high-level concepts which underpin the ontology’s design, and
(3) guidance on using ontologies with modern hazard analyses.

2 CONCEPTS IN THE AADL EMV2 ERROR LIBRARY
This section introduces a number of concepts that are critical to the Error Library: definitions of important terms, the importance of effects based reasoning, a number of classifications of system error, and a brief overview of relevant aspects of building system models in AADL.

2.1 Definitions of Important Terms
There are a number of definitions of common error-modeling terms present in the literature; in this paper we try to use terms within their well-understood meanings. Full definitions of all relevant terms are given in the EMV2 Standard Document [24] which notes that they are derived from an existing IEEE standard [14]. Three
particularly relevant definitions (excerpted / adapted from [24, p. 64]) are:

- **Error** “[The] difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition”
- **Fault** “[A] root (phenomenological) cause of an error that can potentially result in a failure”
- **Failure** “[The] termination of the ability of a product to perform a required function or its inability to perform within previously specified limits”

2.2 Effects-Based Reasoning

One of the key aspects of the AADL EMV2 taxonomy is its enforcement of local, effects-based error classification. The importance of focusing on the effects of errors (rather than their causes) in system analysis has been discussed previously. Walter and Suri made it an important part of their Customizable Fault / Error Model [28], Procter et al. discuss its importance and benefits in some detail [21], and it underlies design of hazard analyses with a top-down focus (e.g., STPA [17], FTA [7], SAFE [20]). Key to enabling an effects-focus (and, more generally, to the design of the Error Library) is Powell’s work on failure modes and assumptions—not only do we adopt much of his service model (see Section 2.3) but we also use his notions of observers [19]. At a high-level, the primary justifications for focusing on effects rather than causes are:

1. **Reducing Ambiguity**: Walter and Suri argue that cause-based classification methods, such as the Laprie taxonomy [15], can lead to the same fault being classified differently in different systems, but note that this is impossible with an effects-focus [28].
2. **Reducing Analysis Space**: Procter et al. note that the set of causes is essentially unbounded, while observable fault effects can, at a given level of abstraction, be known statically—making analysis of larger systems more tractable [21].
3. **Enabling Local Reasoning**: As the analysis space is reduced to a more manageable level, it is possible to gain a notion of completeness and minimalism with some effect-classification taxonomies [6]. This means that, to a limited extent, a component can be analyzed independently of other parts of the system it is a part of.
4. **Connecting to Compositional Reasoning**: There is an important connection between effects-based reasoning, assume/guarantee logics, and design by contract styles. Errors can also be conceptualized as violations of a component’s assumptions [24]. Similarly, the effects of those errors are typically violations of the component’s guarantees.
5. **Enabling Safety and Security Co-Analysis**: There is a growing recognition of the overlap between traditional safety and security concerns. An effects focus allows, in some cases, for the simultaneous co-analysis of safety and security, a topic explored by Procter et al. [21].

2.3 Error Identifiability Concepts

In this paper we use the term error identification in a broad sense. An error is said to be identifiable if system behavior could be modified as a result of its presence; this is analogous to being observable, and is closely related to the term interference in the security community [11]. An unidentifiable error, then, is an incorrect system state that has no impact on system functioning, e.g., a sensor reading that is close enough to the correct value to not cause a component (or system) failure. We use the term detection more narrowly: to mean that the system itself (instead of only an omniscient observer) can determine that something is incorrect and compensatory actions should be taken.

Whether an error can be detected (and compensated for) depends on a number of factors: system architecture, application demands, the presence of error correcting technology, etc. We propose a number of concepts relevant to the classification of errors in Fig. 1. Some error types, including those having to do with values, timings, and errors of commission and omission have their intuitive meanings. We explain the other concepts at a high level in this section and document the error types that make up the Error Library in Section 5.

**Quantity** Our views on error identification were inspired by Powell as well as Bondavalli and Simoncini [3, 19]. Central to these views are the concepts of service and service items.

1. **Service Items**: Powell defines service items as “value-time tuples,” i.e., messages that have both a value and a time [19]. In some cases errors can be identified in individual service items.
2. **Sequences**: In other cases, a component may be able to compensate for individual, erroneous service items but unable to function correctly if the errors persist past some sequence of service items (the length of which is bounded by some implementation-specific value, commonly written $k$).
3. **Services**: Lastly, every item in the entire service (i.e., as $k$ grows large) may have to be incorrect for an error to be identifiable.

**Subtlety** Some errors can be detected and compensated for using only the information a component has, while others are more subtle and would only appear to an omniscient observer. We again turn to Bondavalli and Simoncini who propose further differentiation based on this concept of error detectability [3].

1. **Detectable Errors**: Errors which are detectable by the system itself are termed coarse incorrect errors by Bondavalli and Simoncini [3], we use the term detectable.
2. **Undetectable Errors**: Other errors, which would only be identified by the hypothetical “perfect observer” who has full knowledge of the system’s specification, are subtly incorrect according to Bondavalli and Simoncini [3]; we use the term undetectable.

**Replication** Errors may only be detectable by a system if messages are correlated and compared between several receiving components. This is commonly the case in distributed systems where a sensor may provide input to multiple components: if those components cross-check their received values with each other, differences in message timings and values can be detected and potentially addressed.

**Recoverability** Many safety-critical systems will have error-correction capabilities, but most of these techniques can only compensate for errors within some range. This classification is for cases where input errors exceed the error-correction boundary.
AADL provides a set of language constructs and connections that enable system designers to build high-fidelity models of their systems and then analyze multiple aspects of them. We briefly summarize two aspects of the language that are relevant to this work here, a more complete discussion of the language is provided elsewhere [9].

**Functional and Hardware Views** AADL’s modeling constructs include both runtime and hardware elements. The runtime elements are primarily software, e.g., process, thread, subprogram, etc. and they can be bound to hardware elements such as processors, memory, and buses [9]. One of the language’s strengths is its ability to describe interactions between the two views of a system, such as how the failure of a hardware component may affect certain subprograms, or the impact of assigning a particular process to a specific processor.

**Instantiation and Propagation Path Resolution** When a system designer writes AADL, he or she is specifying what’s called the declarative model, which can then be instantiated. Instantiation—which is typically done automatically by tooling—includes a number of transformations, including fully resolving the system’s propagation paths. These paths are used by various analyses to determine where data and control events—including errors—start, are transformed, and stop. Thus, most analyses require a system model to be instantiated.

### Figure 1: Error concepts used in the Error Library.

#### Permanence
Intuitively, not all problems with system components are lasting: sensors may be bumped, control algorithms may be preempted, etc. These problems may not be fatal, and long-lasting faults should in some cases be dealt with differently. Avižienis et al. classify faults depending on whether they are permanent or transient [2] and we make a similar distinction with errors.

**Consistency** A related issue to permanence is consistency: error behavior is often inconsistent. A value error (such as “Too High”) may either occur regularly according to some pattern, or be randomly interwoven with correct values or other value errors (e.g., “Too Low”). Since jittery errors (i.e., those that do not follow a distribution) can cause different behaviors than those that appear consistently, we include the concept in our classifications.

### 2.4 Modeling with AADL

AADL provides a set of language constructs and connections that enable system designers to build high-fidelity models of their systems and then analyze multiple aspects of them. We briefly summarize two aspects of the language that are relevant to this work here, a more complete discussion of the language is provided elsewhere [9].

**Functional and Hardware Views** AADL’s modeling constructs include both runtime and hardware elements. The runtime elements are primarily software, e.g., process, thread, subprogram, etc. and they can be bound to hardware elements such as processors, memory, and buses [9]. One of the language’s strengths is its ability to describe interactions between the two views of a system, such as how the failure of a hardware component may affect certain subprograms, or the impact of assigning a particular process to a specific processor.

**Instantiation and Propagation Path Resolution** When a system designer writes AADL, he or she is specifying what’s called the declarative model, which can then be instantiated. Instantiation—which is typically done automatically by tooling—involves a number of transformations, including fully resolving the system’s propagation paths. These paths are used by various analyses to determine where data and control events—including errors—start, are transformed, and stop. Thus, most analyses require a system model to be instantiated.

### 3 RELATED WORK: CLASSIFICATION AND GUIDEWORD SCHEMES

This section discusses a number of popular safety concept classification schemes which, we argue, can be considered non-operationalized taxonomies of system errors. Often, these take the form of a set of guidewords, which are terms used in hazard analyses to guide designers/analysts to consider ways that a system can deviate from its intended functionality [7]. Guidewords are typically adapted to the needs of the system by incorporating domain knowledge, prior analyst experience, etc. Redmill et al. provide a thorough treatment of the role of guidewords and their use in software including when and how to adapt them to a particular domain [22]. If the EMV2 Error Library is used independently of AADL, it can be thought of as a taxonomy of guidewords that is to some extent interchangeable with the others in this section.
3.1 Hazard and Operability (HAZOP)
The HAZOP technique was created in the 1970s by the United Kingdom’s Institute of Chemical Industry, and it is essentially a multidisciplinary brainstorming technique [7]. It is performed by having experts in different system aspects and lifecycle phases—e.g., designers, testers, users, etc.—apply guidewords like less or late to individual components’ parameters. It has been applied to a range of system types, including software [7, 22]. Wallace, whose Fault Propagation and Transformation Calculus [27] partially inspired AADL’s Error Model, also classified failures based on a set of guidewords in a style modeled off of McDermid et al.’s use of HAZOP on software [18].

3.2 System Theoretic Process Analysis (STPA)
Leveson describes a hazard analysis technique called System Theoretic Process Analysis (STPA) in the book Engineering a Safer World [17]. Leveson explains that STPA also uses guidewords, though they are applied to the control structure of a system rather than its physical design [17, pg. 212]. The two steps of STPA—identifying hazardous control actions and determining causation—each have their own set of guidewords. STPA’s improvement over more traditional techniques is in part a result of its use of control theory; the terms it uses such as providing causes hazard or wrong timing causes hazard reflect this control-theoretic approach. Tooling (e.g., [1, 20]) and partial formalizations (e.g., [26]) for STPA and its variants and extensions have also been created.

3.3 The Avižienis Taxonomy
Avižienis et al. describe a full taxonomy of a number of concepts in the critical systems space [2]. This includes a number of classifications of faults, errors, failures etc. Like the EMV2 Error Library (and unlike HAZOP and STPA), the classification is presented as a taxonomy, rather than being described in the context of a specific analysis process.

3.4 Dolev and Yao’s Adversary Model
Dolev and Yao describe a formal, well-established adversary model where the attacker controls the network, e.g., message values and delivery timing can be modified arbitrarily [6, 12]. Their model consists of a formal description of adversary capabilities, though, so it is not similar to the discussion-oriented terminology used by HAZOP or the control-theoretic guidewords models used by STPA. This model has been used as the basis for guidewords that simultaneously address safety and security [21].

4 TAXONOMICAL MECHANISMS
The AADL Error Library is composed primarily of error types, which are organized into a hierarchy, and then (if used within an AADL model) are instantiated along with the rest of the declarative system specification. In this section we present an overview of these mechanisms and the motivation behind their use. For previous discussions and illustrations of the AADL Error annex see, e.g., [5, 16], or for complete documentation, refer to Section E.5 of the AADL language standard [24].

4.1 Foundation: Error Types
Error types are used in three related ways. They can represent: a) a propagated error, i.e., malformed input to a component that is causing erroneous behavior, modeled with the AADL error flow and error propagation constructs; b) an activated fault, i.e., an internal problem with a component that is identifiable, modeled with error source and error event; or c) the behavior state of a system component (using AADL’s error behavior construct) [24]. At first blush, this reuse of one language construct for three concepts may seem to be an abuse of notation. However, given the component-centric, effects-focused view presented by AADL and its error model, the concepts are equivalent. That is: the behavior of some component A, which receives its input from another component B, depends only on the input received. The source of deviations in that input—an activated fault in B, B’s having propagated an error from a third component C, or B’s current state—are irrelevant to A.

4.2 Relation: Building a Hierarchy
In the library, multiple error types are typically related to one another via extension. Type extension creates a subtyping relationship. For example, one of the base value error types, ItemValueError is extended into DetectableValueError and UndetectableValueError. Both of these types have the semantics of their more abstract supertype (ItemValueError) but carry additional information about the detectability of the value error. Taken together, the latter two types completely compose the former; i.e., there is no way for a service item to have an error in its value that is neither detectable nor undetectable.

4.3 Instantiation: Using Declared Types
As discussed in Section 2.4, AADL models are instantiated before analyses are conducted on them. When a system that contains error annotations is instantiated, the portions of the model specified using the error annex are also instantiated for analysis. Error types produce tokens which move through a system-wide petri net-like simulator that is derived from the collection of component error flows and propagations. This simulator can then be mined for error behaviors [5].

5 THE TAXONOMY
The AADL EMV2 Error Library (Section E.6 of the AADL Standard [24]) plays three interrelated roles. First, it is a list of guidewords, suitable for brainstorming and manual analysis in the style of e.g., HAZOP (see Section 3.1) or STPA (see Section 3.2). Second, it contains a wide range of error types that can be directly used in AADL system and component specifications. Third, it serves as the basis from which system- and domain-specific error types are derived. As we discuss in Section 7.1, the contents of the library continue to evolve in an attempt to strike the correct balance between these goals and manageable brevity.

The meaning of a particular error type in the library is a deviation from the correct value, as identifiable by an omniscient observer [19]. Note that, due to space constraints, our goal in this section is not to fully detail the exact semantics of each error type. Full details are available in the AADL EMV2 standard itself. Rather, here we
give an overview of, and the intuition behind, the families of error types used in the library. Table 1 lists some example formalizations. Error types with formalizations in Table 1 are shaded in Figs. 2-5.

Note that error types that extend the same supertype are mutually exclusive. Two error type families (Service and Replication) have a single common parent, and so the subtypes of these families cannot co-occur with other subtypes of the same family. The other two families (Value and Timing) have no such restriction, however: service items can, for example, have both a wrong value and be part of an out-of-order sequence.

### 5.1 Service Errors
Recall from Section 2.3 that we use the terms *sequence* and *service* to refer to bounded and unbounded (respectively) ordered collections of *service items*—i.e., messages, inputs, etc., where accuracy and timeliness are required for correctness. The first family of error types, shown in Fig. 2, contains errors in "the number of service items delivered by a service." [24]

**Commission and Omission** Of the six error types that extend the top level *ServiceError* type, four deal directly with either unexpected items and services (commission) or missing ones (omission). If the errors are single events, the *ItemCommission* and *ItemOmission* types should be used; if it is an entire service that is in error the Service counterparts should be used instead.

**Sequences** Errors that are more subtle than single service items or complete services are captured using the children of the *SequenceOmission* and *SequenceCommission* types. Some, like early and late service start and termination, have an intuitive meaning. Two of the remaining four—*TransientServiceOmission* and *TransientServiceCommission*—are used for intermittent versions of their service-based counterparts. The final error types, *BoundedOmissionInterval* and *BoundedCommissionInterval* are used when service item errors occur more frequently than some specific bound. Note that if a sequence error persists longer than the system-specific bound, it becomes a service error.

### 5.2 Value Errors
The second error family, shown in Fig. 3, collects errors which represent incorrect values. The collection is split into three hierarchies: one dealing with items, one with sequences, and one with services.

**ItemValueError** These errors deal with individual service items with incorrect values. The family is divided first by the detectability of the errors; i.e., if only an omniscient observer could detect them, or the system itself can as well. If they are detectable, errors can be classified as either out of range, which means they are outside of some domain-specific range (e.g., a percentage that is more than 100 or less than 0) or out of bounds, which means the value is unrepresentable in the expected type (e.g., a string is received instead of an integer).

**SequenceValueError** Some programs may be able to behave correctly when a small number of input values are incorrect, but longer sequences of erroneous values cannot be compensated for. This is further extended into out of order sequences and values that are Stuck, i.e., repeating the same value. A final type, *BoundedValueChange*, signifies successive values that are both in range but are implausibly far apart (according to some user-specified boundary).

**ServiceValueErrors** Value errors with entire services signify that all service items have value errors.
5.3 Timing Errors
Timing errors can be challenging to model because there are two different notions of timing that can be used. The first, inter-arrival time, specifies the length of time allowable between service items. The second, clock time, specifies delivery deadlines according to some more absolute notion of time, e.g., time-of-day, Unix time, time since system initialization, etc. Both types of timing issues are modelable using our library’s timing error hierarchy, shown in Fig. 4. Note that ItemTimingError can occur using either type of timing specification: they signify only that a single service item is either early or late according to some system-specific deadline.

SequenceTimingError This error family uses inter-arrival timing specifications. The more abstract SequenceTimingError can be refined into HighRate (violations of the minimum inter-arrival time), LowRate (violations of the maximum inter-arrival time), and RateJitter, which is a combination of the two.

ServiceTimingError This error family uses clock timing specifications. The generic notion of ServiceTimingError is extended by both EarlyService, where service items arrive consistently early, and DelayedService, where they are late.

5.4 Replication Errors
The family of replication error types (shown in Fig. 5) is used to model errors in replicated service items, which may come about as a result of various architectural mechanisms, e.g., redundancy patterns, parallel execution, etc. Work in this area was inspired in part by Walter and Suri’s ideas on communication symmetry [28], which posited that otherwise-undetectable errors could, in some systems, be detected if service items were replicated and used in multiple places.

In the error library, this gives rise to a new family of error types that combines the previous error families—service, value, and timing—with either symmetric or asymmetric presentation. If all replicates are in error, the error is said to be Symmetric, otherwise it is Asymmetric. In addition to timing errors, which have their intuitive meaning, replicates can differ in value and presence.

Value Errors If one or more of the values of the replicates differs from other replicates, two errors are possible, depending on the test for equality used: if any variation in replicates produces different behaviors (e.g., if the inputs are used in a hash function) then an AsymmetricExactValue error is present. Otherwise and the most appropriate error type is AsymmetricApproximateValue.
The AADL Error Library: An Operationalized Taxonomy of System Errors

HILT'18, November 2018, Boston, Massachusetts, USA

![Hierarchy of timing errors, adapted from [24]. A formalization for the shaded type is given in Table 1.](image)

![Hierarchy of replication errors, adapted from [24]. A formalization for the shaded type is given in Table 1.](image)

**Omission Errors** Errors of omission can occur in replicated services either symmetrically or asymmetrically. We further distinguish between the omission of individual service items and entire services.

### 6 EXAMPLE PROCESS

The AADL Error Library is not typically used on its own. Rather, it is most often used as a piece of the larger safety analysis process in safety-critical engineering projects. In addition to efforts specifically devoted to demonstrating or analyzing the Error Annex (e.g., [5, 16]), a number of larger projects have used the ontology in the avionics domain, though in a range of settings. These include post-hoc analysis of an industrial safety accident [13], extensive analysis of an individual component [25], and a large-scale military “shadow project” feasibility demonstration [4, 8].

Feiler gives perhaps the most detailed description of a typical use of the error ontology in his report on the military shadow project [8]. Steps 0, 1, and 3 (in the process below) compose the first phase of the effort: identifying which error types to use. Steps 2 and 4 compose the second phase: specifying how the error types are consumed and produced. This second step—essentially specifying the input and output error specifications of each component—is what takes advantage of both the propagation paths (created during instantiation) and the type system which underlies the library to enable forward and backward reasoning. Feiler’s experience was that:

1. At a high level, the safety process began with a functional and hardware model of a system specified in AADL. The ontology was used, along with the functional model of the system, to drive conversations between a safety analyst and a domain expert.

2. The output of this conversation was a collection of error types that had been customized to the domain, e.g., the ServiceOmission error type was extended into a specific subsystem failure. One of the most interesting error discoveries was that information on the system’s location traveled along two paths, one of which had a significantly longer computation time. This led to the discovery of an AsymmetricTimingError.

3. All of the findings from the dialogue were operationalized by extending error types from the ontology into a system-specific error library. These types were then used to drive a hazard analysis of the system.

4. Steps 1 and 2 were then repeated for the hardware model.

5. Finally, interactions between the hardware and functional model were analyzed using the combined set of system-specific error types to determine the final error behavior of the system.

Though other users of the error library don’t follow this exact process, most follow a similar pattern. The core usage can be summarized as: a) starting with the built-in error library; b) adapting its terms to a specific system/domain; c) re-integrating those adaptations back into their system model via type extension; and d) re-analyzing other system views with the new, more specific error types.
7 CONCLUSION

The original goals for the EMV2 error library were to develop a common set of error types based on a range of existing guideword schemes and—based on previous work by Walter and Suri—to classify errors in the set based on their effects [24, 28]. Based on the experiences we and others have had using the library, we believe these goals have been achieved. In this paper, we have laid out much of the rationale behind the library, as well as an overview of the error families and selected formalizations.

7.1 Future Work

Both AADL itself and its Error Modeling annex continue to evolve, and there is more work to do in the future.

Safety and Security There is growing interest in the overlap of security analysis with traditional safety assessment tasks. The extent to which the Error Library, and the EMV2 itself, can support these tasks is an open area of research. While we are encouraged that most security issues eventually manifest in one of the same error types as traditional safety concerns, others, such as the inadvertent leaking of privileged data, have no safety equivalent and will likely require modifications to the library.

Limitations of an Effects Focus A strict focus on the effects of errors is one of the strengths of the Error Library, though there may be limits to this approach. To this end, we are experimenting with less effect-focused error families that deal with, e.g., concurrency issues. The costs, benefits, and tradeoffs involved in expanding beyond a strict effects focus is not yet clear, however, so more study needs to be done.

Patterns for Co-Occurring Errors System behavior in the presence of errors can become arbitrarily complex when those errors can co-occur, so a concise way of specifying co-occurrence is desirable. The error library’s set of replication errors (i.e., the creation of wholly new types, see Section 5.4) is one attempt at this, as is the more general error_type_product construct, which creates a new error type from two extant types, e.g., a value error and a time error. We are still researching when one specification mechanism is preferable to another.

ACKNOWLEDGEMENTS

The authors wish to thank the anonymous reviewers, Todd Loizies, and Lutz Wrage for their assistance and numerous helpful suggestions in improving this work. All Rights Reserved. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. DM18-1164.

REFERENCES