2021 IEEE 29th International Requirements Engineering Conference Workshops (REW) | 978-1-6654-1898-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/REW53955.2021.00078

The Role of Environmental Deviations in Engineering Robust Systems

Eunsuk Kang
Institute for Software Research
Carnegie Mellon University
Pittsburgh, PA USA
eunsukk@andrew.cmu.edu

***Abstract*—Software systems are developed with various assumptions about the environment. However, over time, the environment may evolve and deviate from its expected behavior, possibly undermining desired requirements of the system. In this paper, we argue that identifying and treating *environmental deviations* as a first-class concept throughout the development cycle is a key to the successful engineering of robust software systems. We describe some examples of environmental deviations in different domains, discuss their implications on development activities, and also put forward research challenges that we believe the requirements engineering community is particularly well-positioned to tackle.**

I. Introduction

A fundamental activity in requirements engineering is establishing the following relationship between *requirements*, *domain assumptions*, and *specifications* [5]:

$$S, E \vdash R$$

That is, if the machine (e.g., software being developed) satisfies its specification (S) and certain domain assumptions (E) hold, then the desired requirement (R) must follow.

Having established this relationship, one might ask: *What if one of the domain assumptions is violated?* Such a violation may occur, for example, due to a faulty or malfunctioning entity in the environment (e.g., a faulty vehicle sensor that provides inaccurate data about the world). In security-relevant domains, a malicious agent in the environment may deliberately attempt to undermine an assumption that the system relies on (e.g., that the user does not inadvertently divulge a sensitive passcode). More generally, in many modern systems, the environment constantly evolves over time, *deviating* from expectations about its properties and behaviors that were identified during the initial development.

The challenge of developing systems that are reliable even under deviations in the environment is not new, and has been studied extensively in traditional engineering disciplines. For example, in civil engineering, structures are regularly designed to withstand possible changes in weather conditions or building loads [12]. Similarly, in aeronautics, the concept of *operational envelope* is used to describe the boundary outside the normal operating conditions under which the system is capable of maintaining safe operation [13]. In software, similar types of deviations have been studied within specific domains (e.g., hardware faults in embedded systems or network failures in distributed systems), but there appears to be a lack of systematic, general methodologies for identifying and managing such deviations during a development process.

In this paper, we argue that *environmental deviations* must be considered as an explicit, first-class concept in software engineering. Conceptually, an environmental deviation (denoted Δ) is an observable unit of change to the behavior or a property of the environment. For example, in the context of a distributed application (S), a property of interest may be the average latency of the underlying network (E); during unexpected failures in the network, its latency might increase by some k milliseconds above the expected value (i.e., $\Delta_{latency} = k$ ms), adversely affecting the system throughput (R). As a different example, consider the design of an infusion pump device (S) that relies on a human operator (E) entering correct prescription data to ensure safe delivery of medicine (R). Due to a distraction or stress, the operator may inadvertently commit an error by omitting or performing certain actions out-of-order (i.e., $\Delta_{operator}$ = an erroneous operator behavior), possibly undermining the safety of the patient. In these cases, it would be ideal for the system to continue to provide some level of performance or safety.

By making Δ explicit, we can precisely pose the problem of developing systems that are *robust* against possible deviations in the environment. To be more specific, let E' be an environment that results from the deviation of the original environment E by Δ; i.e.,

$$\Delta = E' - E$$

Under this deviated environment, the system may no longer satisfy a desired requirement R:

$$S, E \vdash R \qquad S, E' \nvdash R$$

The goal is then to develop a revised machine S' that is capable to satisfying the requirement even under these deviations:

$$S', E' \vdash R$$

Achieving such S' involves various activities throughout the system lifecycle, including requirements (for identifying relevant Δ in a given problem domain), architecture and design (for devising mechanisms to handle Δ), testing and validation (for checking that the implemented system is indeed capable

978-1-6654-1898-0/21/$31.00 ©2021 IEEE
DOI 10.1109/REW53955.2021.00078

of handling Δ), and operations (for detecting new types of deviations outside of Δ).

In the rest of the paper, we further elaborate on these activities and describe a set of open research problems that the requirements engineering community is particularly well-positioned to tackle. We first begin by describing, in the following section, some examples of environmental deviations that arise in software-intensive systems.

II. Environmental Deviations: Examples

a) Human-machine interfaces: Safety-critical systems such as medical devices, automobiles, aviation systems, and industrial control systems provide some form of human operator interface. Such a system is typically designed with a set of assumptions about the expected behavior of the operator. For instance, in a radiation therapy system, the operator (e.g., a therapist) would be expected to perform a sequence of actions in a particular order (e.g., choose radiation settings, enter patient information, and then press a confirmation button to begin the treatment). These assumptions are often implicit but sometimes encoded in system artifacts—for example, as instructions in a user manual or training materials.

In practice, humans are not perfect, and will occasionally deviate from the expected behavior [10]. The therapist under stress or distraction may inadvertently commit an error, such as skipping a critical action or performing actions out of order. Ideally, a system that is robust would ensure that a critical requirement is satisfied even under these erroneous behaviors (e.g., prevent radiation overdose even if the therapist makes a mistake). Unfortunately, there are numerous cases of catastrophic safety accidents that are attributed to human errors, including the infamous Therac-25 accidents [9] (where the software failed to resolve race conditions that occurred when the operator quickly changed the radiation settings, ultimately resulting in fatal overdose).

If these types of environmental deviations—in form of erroneous human behaviors—are anticipated and articulated during the requirements stage, the system could be explicitly designed to be robust against those deviations. For this, we can turn to the wealth of research in the human factors community on studying and classifying different types of human errors and techniques for mitigating them [14], [7]. For instance, *forcing functions* [10] are a design mechanism that can be used to constraint the user behavior and reduce the likelihood of a human error resulting in an undesirable outcome (e.g., ask the therapist to confirm the selection of a particular action before proceeding to the next step).

b) Security protocols: Security protocols play a vital role in modern software systems by providing critical services such as authorization and authentication. Like any piece of software, a security protocol is also designed with certain assumptions about the deployment environment (e.g., web browsers). These assumptions include both (1) the expected behavior of the protocol participants (e.g., the browser user) as well as (2) the knowledge and capability of the attacker (e.g., information about the system that the attacker has access to).

However, even well-established security protocols have been broken due to violations of both types of assumptions. For instance, many implementations of OAuth—a widely-used protocol for third-party authorization—have been shown to be vulnerable to attacks in part due to their reliance on an assumption that the user's browser shares a secret token only with the trusted protocol participants. Unfortunately, unless the protocol implementer takes an additional precaution, this assumption can be violated by standard browser-based attacks, such as cross-site request forgery (CSRF) [17]. Another common type of security failure occurs when a system makes a brittle assumption about the attacker's knowledge—for example, assuming that the attacker does not know the unique IDs that are assigned to users, even though they are often easily enumerable.

We believe that requirements engineering has a significant role to play in improving software security by helping the developer articulate environmental assumptions, identify those that are brittle and likely to be violated, and build in mitigations to ensure security even under those deviations.

c) Distributed systems: Modern distributed systems are deployed in a large, complex network (e.g., a cloud) with dynamic and unpredictable behaviors, such as node failures or communication delays. Unlike the earlier examples in this section, this is one domain where environmental deviations are treated as a norm and systems are developed with the goal of being robust against those deviations. For example, most successful distributed applications employ *fault-tolerant* coordination protocols, such as Paxos [8] and Raft [11].

d) ML-based systems: The increasing prevalence of machine-learning (ML) components in software has attracted a large amount of interest in validation and testing techniques for ML. One type of property in ML that is being actively studied is *model robustness*. While there are various notions of robustness in ML, in one common definition, a classifier is said to be robust if its output prediction remains the same under small perturbations to an input (e.g., a computer vision model that classifies a stop sign correctly even if the input image contains slight modifications in pixels) [16]. Given the importance of robustness in safety-critical applications such as autonomous vehicles, there is currently a large amount of research on this topic, such as verifying that a model is robust against some given set of perturbations and training models to be robust against adversarial perturbations.

While these are important research directions, we argue that the narrow focus on ML *models* is insufficient for developing ML-based systems that are truly robust against the dynamic, evolving environment. Many ML components are embedded deeply within the machine (i.e., software) and do not directly interface with the environment. Perturbations to a model input ultimately originate from certain types of deviations in the environment. For example, pixel changes to an input image may be caused by a physical disturbance to a camera, a severe weather condition (such as snowfall), or a malicious actor who deliberately tampers with the target object. Thus, to be able to anticipate the possible range of perturbations to model inputs

and decide which of them are relevant for the given system, one would first need to articulate likely deviations in the environment—foremost a requirements engineering activity.

III. Implications and Research Challenges

Our thesis is that identifying and treating environmental deviations as a first-class concept throughout the development cycle is crucial for engineering robust software systems. In this section, we explore how this concept may play different roles in different phases of the development and also discuss some research challenges that can be tackled by the requirements (and more broadly, software engineering) community.

A. Requirements

a) Elicitation: The first (and perhaps most critical) step in designing robust systems is the elicitation of deviations along with requirements from various stakeholders of the system. This will typically involve working with stakeholders to identify the set of relevant environmental entities and properties, and analyzing ways in which they may deviate from the stakeholders' expectations. In well-established domains (such as aviation systems), domain experts may already possess knowledge about common types of deviations (e.g., pilot errors, engine failures caused by a bird strike, etc.,). In newer domains, such knowledge might not readily come by, and some type of systematic elicitation method may be needed.

One community that we can turn to for potential elicitation methods is safety engineering. It may be possible to adopt risk analysis techniques such as *failure mode and effects analysis* (FMEA) and *hazard and operability study* (HAZOP) to guide stakeholders in systematically identifying relevant deviations. HAZOP, in particular, uses a set of *guide words* to explore how a particular process or entity may deviate from its expected behavior. For example, given a statement that "the network will deliver a message with an average latency of 50ms", the guide word MORE (for quantitative increase) could be used to generate deviations where the network experiences an increased latency, or NOT (for negation) to generate deviations where the network fails to deliver a message.

b) Prioritization and triaging: Not all deviations are equal, and one important activity that should be carried out with stakeholders is to distinguish those that are critical and should be explicitly addressed by the system design from the ones that are less important or less likely to occur in practice. For instance, an autonomous vehicle system should be designed to be robust against possible failures in sensors or extreme weather conditions, but it would be unrealistic to expect the system to be able to deal with another vehicle that deliberately attempts to crash into the ego vehicle. Each design mechanism that is intended to improve robustness (e.g., redundancy, failure handling, monitoring) also introduces additional cost and complexity into the system. No practical system will be robust against all possible deviations and thus, this type of prioritization and triaging is important for achieving a trade-off between robustness and overall development costs.

c) Codification and representation: Within a particular domain, systems may share similar types of deviations that can be codified into reusable patterns. For instance, the human factors community has over time established various categories of human errors, such as GEMS [14] and phenotypical categorization of errors [7]. Similarly, the computer security community has developed databases of common types of security vulnerabilities and attacks, some of the most notable ones being the Common Weakness Enumeration (CWE) and Common Vulnerabilities and Exposures (CVE). These patterns do not guarantee completeness and may not always be applicable to a particular system under development, but could nevertheless aid the elicitation and identification of environmental deviations.

One aspect to consider along with codification of deviations is their *representation*. As it is often the case with requirements in practice, deviations may be documented using an informal natural language. However, more structured and (semi-)formal representations of deviations may provide additional benefits, such as mechanized analysis and traceability throughout the development. For instance, in the phenotypical categorization of human errors [7], each type of error can be represented as a pair of regular expressions (s, s') where s is a string (i.e., a sequence of user actions) that describes an expected user behavior and s' a particular deviation from it. For example, the *omission* error type may be captured as (s_1as_2, s_1s_2), where s_1 and s_2 are strings and a is a particular action that the user may inadvertently skip; similarly, the *repetition* error can be captured as (s_1as_2, s_1aas_2). Building on this formal representation of human errors, researchers have constructed model-based tools for analyzing a computer interface and identifying errors that may lead to a violation of a safety requirement (e.g., [3]).

B. Architecture and Design

a) Robustness analysis: Having developed an initial design of the system, a type of analysis that one may wish to perform is to check whether the system (S) can establish a desired requirement (R) even under a possibly deviating environment (E', where $E' = E + \Delta$ for normative environment E and some deviation Δ); i.e., check that $S, E' \vdash R$. This type of activity, which we call here *robustness analysis*, is routinely carried out in traditional engineering disciplines. For instance, in civil engineering, structural analysis involves using simulation or mathematical analysis to evaluate how well a building under design is capable of withstanding additional loads beyond the expected norm. Within software domains, robustness analysis appears to be less common, and opportunities abound for developing techniques and tools for helping developers carry out this activity.

In systems where the environment and the system can be modeled formally (e.g., as finite state machines), it may be possible to automate robustness analysis. In our own prior work [18], we introduced a formal, *trace-theoretic* notion of robustness, where a deviation is represented as a trace—a sequence of environmental actions—that exists outside of the

behavior of the normative environment. Based on this notion, we developed an analysis tool that automatically computes the set of all possible deviations (Δ) under which the system can satisfy a desired requirement (R) and demonstrated its applications on safety-critical interfaces and network protocols. However, our analysis was limited to discrete, state-based models of the environment, and further research is needed to extend this type of analysis into other types of environmental models (e.g., those that are stochastic in nature, such as Markov decision processes).

b) Design methods for robustness: Robustness analysis may reveal that a given design of the system is not robust enough to tolerate certain kinds of deviations (i.e., $S, E' \nvdash R$). In this case, a natural next step is to consider a redesign of the system (S') to achieve a desired level of robustness (i.e., $S', E' \vdash R$). Systematic methods and tools for helping developers carry out this type of redesign are currently scarce and could strongly benefit from further research. An example of such a method could be an architectural refactoring technique that leverages a repository of design patterns to improve robustness (e.g., fault tolerance mechanisms to improve robustness against node failures in a distributed system [6]). In addition, if the environment and the system are formally modeled, this redesign activity could be formulated as a type of model transformation problem and (semi-)automated.

C. Testing and Validation

Techniques such as fuzz testing (e.g., [4]), model-based testing [2], and chaos testing [1] are used to test the robustness of a system against unexpected inputs or environmental failures. These techniques could be augmented with information about deviations to guide the test generation process and evaluate the system against those specific deviations. For example, instead of generating test cases in a random manner (e.g., random input action sequences to test a user interface), one could generate a specific set of tests to cover the relevant deviations (e.g., input sequences where certain critical actions are omitted or repeated). This *deviation-driven* approach to testing could potentially achieve a higher level of behavioral coverage using a smaller number of tests and provide improved traceability between requirements and tests.

D. Deployment and Monitoring

Environmental deviations continue to play an important role past the validation stage and well into deployment and maintenance. In most systems, a model of the environment considered during the requirements stage is likely to be a rough approximation, and associated deviations may also turn out to be inaccurate. One way to overcome this challenge is to deploy a runtime monitor that has knowledge of the deviations (Δ) that were elicitated during the requirements stage. The monitor would then continuously observe the environment and look for an environmental trace t that corresponds to:

1) One of the elicitated deviations ($t \in \Delta$; e.g., an anticipated network failure in a distributed system), in which case the monitor would trigger a mechanism that is designed to handle that particular type of deviation (e.g., activation of standby nodes to serve client requests), or
2) An unexpected or unhandled deviation ($t \notin \Delta$; e.g., a complete network failure), in which case the monitor may trigger a special fail-safe mechanism that performs a more drastic action (e.g., graceful system shutdown).

Techniques and tools developed by the *self-adaptive systems* [15] community could also be leveraged here. For example, a MAPE-K loop could be deployed to monitor the environment for unanticipated deviations and perform appropriate adaptation strategies to provide an acceptable level of performance or safety even under those deviations.

IV. Conclusion

In this paper, we have proposed a notion of *environmental deviations* and argued that they must be explicitly identified and taken into account as a first-class artifact throughout the development cycle. We believe that the deviation-related activities described in this paper will become more relevant as software is increasingly deployed into highly dynamic, constantly evolving environments, and that requirements is the most critical and effective place to start tackling the challenges of engineering robust software systems.

References

[1] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal. Chaos engineering. *IEEE Software*, 33(3):35–41, May 2016.

[2] F. Belli, A. Hollmann, and W. E. Wong. Towards scalable robustness testing. In *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement*, pages 208–216. IEEE, 2010.

[3] M. L. Bolton and E. J. Bass. Generating erroneous human behavior from strategic knowledge in task models and evaluating its impact on system safety with model checking. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 43(6):1314–1327, 2013.

[4] P. Godefroid, M. Y. Levin, D. A. Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[5] C. A. Gunter, E. L. Gunter, M. Jackson, and P. Zave. A reference model for requirements and specifications. *IEEE Software*, 17(3):37–43, 2000.

[6] R. S. Hanmer. *Patterns for fault tolerant software*. J. Wiley and Sons, 2007.

[7] E. Hollnagel. The phenotype of erroneous actions. *International Journal of Man-Machine Studies*, 39(1):1 – 32, 1993.

[8] L. Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[9] N. G. Leveson and C. S. Turner. Investigation of the therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[10] D. A. Norman. *The Design of Everyday Things*. Basic Books, Inc., USA, 2002.

[11] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319, 2014.

[12] H. Petroski. *To engineer is human: The role of failure in successful design*. St Martins Press, 1985.

[13] J. Rasmussen. Risk management in a dynamic society: a modelling problem. *Safety Science*, 27(2):183 – 213, 1997.

[14] J. Reason. *Human Error*. Cambridge University Press, New York, 1990.

[15] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *TAAS*, 4(2):14:1–14:42, 2009.

[16] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks, 2014.

[17] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich. Explicating sdks: Uncovering assumptions underlying secure authentication and authorization. In *USENIX Security*, pages 399–314, 2013.

[18] C. Zhang, D. Garlan, and E. Kang. A Behavioral Notion of Robustness for Software Systems. In *ESEC/FSE*, 2020.