Emulating Environment-Dependent Software Faults

Position Paper

Roberto Pietrantuono*, Stefano Russo*, and Kishor Trivedi†
* DIETI, Università di Napoli Federico II 80125, Napoli, Italy. Email: {roberto.pietrantuono, stefano.russo}@unina.it
† ECE Department Duke University 27708, Durham, NC, USA. Email: ktrivedi@duke.edu

***Abstract*—The interaction of software with its execution environment is an underestimated cause of complex faults activation and systems failure. This paper discusses a possible framework to emulate anomalous environment conditions in order to assess the impact of the execution environment on a software application under test. We discuss a characterization of the software's execution environment, introducing a classification of failure-exposing conditions of the environment state. Based on this, a set of possible environmental fault operators is introduced, aimed to assess the reaction of an application under unforeseen environment conditions. Practicability of the approach by means of existing fault injection and mutation testing technologies is discussed, along with future challenges and research directions.**

I. Introduction

Software-related system failures are a major cause of disruption in today's complex systems. There is a significant share of software bugs removed during the pre-release testing stage, especially for systems with high reliability requirements. Advanced verification and validation techniques (e.g., static analysis, formal proof, stress testing, model checking) are extensively used to clean the code from bugs; however, the interaction with the runtime environment causes unexpected software bugs activation whose impact might be disastrous [1]. Until few years ago, failures due to the interaction with the environment received a very scarce consideration as a testing or V&V problem, while they have been more carefully considered in the requirements and design specification phase (e.g., hazard analysis, FMECA, FTA) or as a runtime fault tolerance problem. However, the increased awareness of the potential impact of complex environment-dependent bugs (e.g., Mandelbugs), observed, with high percentage, even in critical systems [31], is shifting the system-environment interaction problem to the pre-release testing stage, wherein more extensive activities are required to explore potential interaction patterns, and support fault detection as well as fault tolerance design. This trend emphasizes the need for developing testing techniques that are able to account, since their formulation, for the impact of the execution environment on systems failure.

This paper discusses a possible fault-based testing framework to emulate anomalous conditions of the execution environment and assess its impact on a given software under test (SUT). We discuss a characterization of the environment in order to define what are the most important environment components with a potential impact on a software program behaviour. We define a set of elementary *conditions* of the environment that can cause the software under test to fail, which represent potential *triggers* of environment-related software failures. These triggers are defined as conditions that are *necessary*, but may be not sufficient by themselves, allowing complex faults activation to be represented as consequence of one or several triggers occurring together. Based on this characterization, we discuss a set of possible mutation operators (namely, environmental fault operators) whose aim is to alter the environment to induce the SUT interacting with it to failure. The outcome is a framework of operators over the environment, which can be used either to improve the test suite by considering the SUT robustness to the environment, or to improve the design of runtime fault tolerance mechanisms for guaranteeing the adoption of correct countermeasure in response to specific environmental states.

The paper ends up with a discussion on the applicability of fault operators by means of current tools and technologies, highlighting the major challenges and research directions. In the following, section II surveys relevant work in the field of mutation testing; section III describes the characterization of the environment in terms of *triggers*; section IV defines fault operators; section V and VI discuss the existing technologies that could be used for implementing the defined operators, the challenges and the main research directions.

II. Background and Related work

The idea of using faults artificially inserted into a system under test is exploited in various fields in the software engineering and fault tolerance community. The main techniques exploiting this idea are known as *mutation testing* and *fault injection*. Mutation testing is a well-known technique to assess a test suite with respect to its ability to detect software faults [2]. A set of faults are deliberately injected by means of mutation operators assumed to be representative of realistic mistakes that programmers might have introduced into the code. Applying mutation operators, many faulty versions of the software are created; test cases are run against such versions (called *mutants*) and the good performance (namely, the *adequacy score*) of the test suite is measured based on the number of killed non-equivalent mutants (i.e., mutants where the fault is detected). The adequacy score is the ratio between the number of dead mutants and the number of non-equivalent mutants. Mutation operators play a key role for this technique to be effective; many operators have been proposed in the literature with the goal of being representative of classes of

real faults [3], [4]. Besides the type of operator applied, a lot of research has been conducted on mutation analysis for the cost-effective computation of mutants (e.g., selecting the best set of mutation operators [5], [6], [7]) – a survey is presented in [8]. The cost of such a computation is, in fact, a serious obstacle to the applicability of mutation testing to large programs.

Most of the literature on mutation testing focuses on how to modify the source code of the program, without interest in assessing it against changed or unexpected environments. This is an underestimated problem in the testing literature in general. Some exceptions consider the environment under various perspectives. For instance, authors in [9] use mutation testing on environment properties of a LUSTRE program; while authors in [10] use hazard injection to emulate hostile physical environments. In both cases, the focus more on properties of the physical environment that must hold rather than the computing execution environment. Instead, authors in [11] propose to extend mutation testing considering a specific class of environmental bugs, the ones involving integer arithmetic. This is a good example of environment mutation, even though limited to one class of bugs. The work in [12] proposes mutation testing to detect what they call *interaction faults* between hardware and software in an embedded system. The work in [13] considers the environment *perturbation* as possible way to analyze the impact on the program, but authors' focus is on security problems.

The area of fault injection offers valuable techniques and tools for emulating faults in the execution environment. In addition to hardware fault injection, significant research focused on software-implemented fault injection (as known as SWIFI). Its objective is to emulate hardware faults using data or instruction corruption by bit-flip or stuck-at models, which were showed to accurately represent hardware faults with the advantage of being less expensive and easy to implement [14] (e.g., NFTape [15], Xception [16], DOCTOR [18], FERRARI [17]). The ability to corrupt data at such a low level make them useful tools also as an *indirect* form of software fault injection, namely as *error* injection techniques. In fact, what is being injected is not the fault itself but only a possible effect of the fault [19]. Another form of error injection is inserting faults at interface level; in this case, the error inserted at the interface between modules as parameters corruption in functions, API, or even OS system calls. This is usually referred to as robustness testing, and many tools exist to aid such mechanisms (e.g., BALLISTA [20], RIDDLE [21]). Since we are interested in assessing the effect of the environment on the software under test, both error injection approaches (data corruption and robustness testing) can be very useful, as discussed later. Approaches for injecting software faults in terms of programming mistakes have been developed. One of the most popular approaches was the G-SWFIT (Generic Software Fault Injection Technique) [22]. G-SWFIT acts at binary-level by modifying the executable of target application, namely by first identifying sequences of machine code instructions representing the high-level constructs, and then applying a code change in them according to a set of fault operators based on common programming errors. There are approaches to inject software faults at source code level, namely by applying fault operators as changes in the source code (e.g., SAFE tool [23], [19]). Although the injection approach is similar to mutation testing (i.e., inserting artificial faults), the objective in this case is not to assess a test suite, but to evaluate the behaviour of the system when a fault that *escaped* during the test phase appears at operational time. This entails differences in the way fault operators are defined and injected into the code, as there is a different concept of representativeness.

For the purpose of our work, existing techniques, both error injection and source code level mutation, can be very useful to implement environmental fault operators. Mutating the environment to assess its impact is an activity that can be useful both at testing time, e.g., to improve test suites (as mutation testing [2]) or to assess testing techniques performance [24], and or at operational time, e.g., to assess fault tolerance [19] or monitoring mechanisms [25]. The next section introduces a characterization of the environment and of possible failure-epxosing environmental conditions, based on which we define the fault operators.

III. Environment and triggers definition

Common experience suggests that repeating the steps that caused the failure, to recreate the same conditions, is the immediate way to reproduce and debug it. In more complex cases, this is not enough, as a particular state of the execution environment is needed for the bug to reappear. This often causes a sort of "non-determinism" in failure reproduction due to the influence of external factors as the execution environment hardware and software, as well as the user, whose behaviour is unpredictable [26], [32]. The focus of this work is to capture such potential failure-impacting conditions of the execution environment, and then introduce a conventional mutation technique on these conditions. We first define a system model to clarify what we mean by environment.

A. *System Model*

Similarly to [26], we hereafter consider the *application* under test interacting with the main external entities having a potential impact, namely the *user* and the *execution environment*. We assume the application as composed of processes and/or threads communicating with each other to accomplish the intended function, with communication channels implemented by either a global (e.g., shared memory) or a local model (e.g., message exchange). The state of the application includes the states of local processes (and/or threads), and of communication channels among them. A local state is the set of data (e.g., stored in memory or files) which the processes/threads can operate on (i.e., read from/write to). We assume the execution environment as made up of concurrently running software and hardware possibly deployed across multiple machines. Software includes the system software of each machine (i.e., both operating system kernel and other system software such as compilers, linkers, debuggers, editors, user interface, utility software, libraries), middleware (e.g.,

virtual machines, middleware for distributed computing), and application-level programs. Hardware includes the physical machines on which the application is deployed, I/O devices, as well as the network connecting them. The user (not necessarily a human) is the other external entity; s/he interacts with the application by submitting workload requests to it and getting results. We assume that a workload request can be represented as a generic *request of service* (e.g., a query to a DBMS), characterized by a *request type* (e.g., query type, like INSERT) among a set of types, and by a set of request's *input parameters* (e.g., values of an INSERT), in turn characterized by a type and a value. The request is processed and produces an output result (returned as value(s) or as a state change). To accomplish a well-defined task, the user can submit a sequence of (one or more) serial/concurrent requests. It may happen that a sequence of requests to accomplish the task is allowed to be submitted with different timing and/or ordering among requests, i.e., various timing/ordering alternatives are admissible for accomplishing that task.

B. *Environment Bug Triggers*

According to the system model, we have the following contributors for the bug manifestation process: the *workload* submitted to the application, along with the application initial state, the *execution environment*, and the *user* behaviour. While the workload is always necessary to activate a bug, the environment influence might be present or not. When the workload request(s) are the only condition necessary to expose a failure, then the bug manifestation is systematically reproducible. In such a case, reproduction may be an easy task (whenever the bug re-appears by re-submitting the last – or last few – request(s)) or a more complex activity (if longer request sequences are needed): in both cases, when the bug manifestation is just "workload-dependent", we have that *resubmitting (at most a subset of) the same workload requests that caused a failure always produces the same failure, for every valid state of the environment (i.e., for every state of the environment in which the traversed application states are allowed to occur) and for every admissible alternative of requests timing/ordering*.

Workload-dependent cases are, of course, a desirable situation from the tester and debugger perspective, as their reproduction depends exclusively on submitted program's inputs. The problem we focus on is when the environment comes into play and makes the bug discovery a daunting task – namely, whenever, given the same workload sequence, a bug may be activated or not depending on the state of the environment. More formally, we define a *bug manifestation as "environment-dependent" if resubmitting (at least a subset) of the same workload requests that caused a failure, there exists at least one (valid) state of the execution environment or user inputs' timing/ordering*[1] *causing the same failure to not occur*. We wish to describe this environment dependency by abstracting a set of conditions of the execution environment and of user requests that are necessary for a failure to be exposed (we call them "triggers" – examples are in Table I).

Specifically, the execution environment triggers are characterized by three dimensions, distinguishing whether: *i)* the condition involves one of these components of the execution environment, categorized according to the system model: *OS kernel's subsystems* (namely: *OS memory management*, *OS device drivers*, *OS filesystem*, *OS networking*, *OS process management* [33]), *other system software* (e.g.: *utility software*, *development software*, *user interface*, *libraries*), *middelware*, *application-level interacting software*, *hardware resources* (e.g., CPU, disk, physical memory, I/O devices, buses, physical network); *ii)* the condition affects the *activation* or the *propagation* (sometimes, certain environmental conditions may be required just for the bug activation, after which the failure will inevitably appear; in other cases, it happens that activating the bug is not enough, because it may be "contained" by the rest of the application, whereas more persistent conditions must hold after the activation to observe a failure – the case of "propagation" begin affected); *iii)* the required condition is *direct* or *indirect*. In particular, with respect to the second and third dimensions, the four alternatives are:

- **Direct Conditions (DC), affecting the Activation (A)**: there is an environmental condition, which caused the *bug activation*, that: *i) directly* affects the state of the application (i.e., it causes a state change) before the bug activation, and *ii)* does not hold in a successive attempt to repeat the failure-causing steps (namely, the bug activation is not deterministic). Examples are failures caused by particular threads scheduling provoking a race condition (i.e., system-dependent concurrency bugs)[2]. Others are changes of shared variables by another application, which on a successive retry does not reoccur.
- **Indirect Conditions (IC), affecting the Activation (A)**: The same as before, but the state of the application is not directly affected before the bug activation. For instance, the disk is temporarily full, a resource is temporarily unavailable, or another interacting application fails and these events are not managed correctly.
- **Direct Conditions (DC), affecting the Propagation (P)**: The bug is deterministically activated in an environment-independent way (i.e., the retry always causes the bug activation), but there is an environmental condition affecting the *error propagation*, which: *i)* causes the error being propagated in a different way on retry (and a different failure reaches the interface – it may even cause the failure to not appear, if the error is contained before reaching the interface), *ii) directly* affects the state of the application before the bug activation. For instance, the application is designed to change a policy whenever a particular environmental condition occurs (e.g., free memory threshold exceeded), and this change does not

[1] As before, *valid* means admissible, compatible environment state w.r.t. the input requests; in the case of user, it means that the same workload request(s) could be submitted in different timing/ordering producing the same result.

[2] Note that in such a case, due to the execution of a thread instead of another, the state is in general affected before the bug activation

TABLE I
CHARACTERISTICS OF TO THE BUG MANIFESTATION PROCESS

Category	Description of the trigger	Example
USER TIMING/ORDERING	The timing or ordering of user requests	Atomicity violation in concurrent requests
EXEC-ENVIRONMENT- $\langle *, DC, A \rangle$	An environmental condition causing the bug activation, that directly affects the state of the application before the bug activation	A race condition caused by the OS scheduling
EXEC-ENVIRONMENT- $\langle *, IC, A \rangle$	An environmental condition causing the bug activation, that does not directly affect the state of the application before the bug activation	The disk is temporarily full, and the bug is activated on a file writing request in that time lag
EXEC-ENVIRONMENT- $\langle *, DC, P \rangle$	An environmental condition affecting the bug propagation that directly affects the state of the application before the bug activation	The application enables some features when a memory threshold is exceeded, and this changes the way a deterministically activated bug propagates (it may even be contained)
EXEC-ENVIRONMENT- $\langle *, IC, P \rangle$	An environmental condition affecting the bug propagation, that indirectly affects the state of the application before the bug activation	A memory leak is activated deterministically on a requests sequence, and the way it leads to failure changes on a retry, depending on the free memory and on the other application tasks during that retry

affect the bug activation, but affects the error propagation (i.e., on a retry, the bug is always activated, but the policy change causes the error being propagated differently).

- **Indirect Conditions (IC), affecting the Propagation (P)** The same as before, but the state is not affected before the bug activation. For instance, if a bug causing memory leak is activated deterministically on a requests' sequence, the way it leads to failure changes on a retry, depending on the available free memory and on the other applications' tasks during that retry. Indirect conditions triggers, both affecting activation and propagation, may be seen as *pure* environment-dependent conditions, as the state of the application is not influenced by the environment before the bug activation.

We denote the execution environment groups as a triple, for instance: $\langle OS\ Memory, DC, A \rangle$ is a memory state that directly influences the bug activation; $\langle Hardware, DC, A \rangle$, may be a bit-flip, a disk ECC errors, or clock interrupt causing a change in the application state. Regarding the user, we define the following condition:

USER TIMING/ORDERING A sequence of requests to accomplish the intended task may be submitted with several timing or ordering among requests: if these admissible alternatives affect the bug surfacing (namely, a different timing or ordering may prevent the bug surfacing), the bug manifestation is said to depend on this trigger. For instance, if a minimal time lag between two requests is necessary to avoid the failure, the submission timing affects the bug activation. User-dependent concurrency bugs (e.g., atomicity violation, order violation) are also labelled with this trigger. The trigger may appear together with any previous condition.

Note that what typically and ambiguously intended as "bug type" (e.g., resource leak, concurrency, OS interaction) falls in any of the presented categories depending on how it is triggered and surfaces. Triggers are, in fact, properties of the bug manifestation process, not of the bug itself. For instance, bugs due to software aging[3], as a memory leak, may appear as $\langle$*OS Memory, IC,P*$\rangle$ if ideterministically activated, as $\langle$*OS Memory, IC, A*$\rangle$ if not (e.g., activated by concurrency [27]).

[3]Software aging is a phenomenon causing a continued and growing degradation of software internal state during its operational life. It is experienced in many long- running applications, such as web servers [28], middleware (e.g., [29], [30]) and even mission- and safety-critical system [31]

IV. ENVIRONMENTAL FAULT OPERATORS

Fault operators are required to emulate instances of the environment conditions on each identified environment component. Differently from conventional mutation testing, operators will not represent programming mistakes, but anomalous/unexpected conditions in the environment, similar to fault/error injection approaches. Triggers can manifest themselves in several different ways. We first distinguish different *modes* in which an unexpected failure-exposing condition may appear, regardless the involved environment component:

- *Data corruption*: the data on which the SUT acts (by read/write) are corrupted. This may be due to several environment components (e.g., to faulty applications sharing these data, to OS or shared libraries corruption, or to hardware faults, such as disk or memory errors).
- *SUT program corruption*: the SUT itself is corrupted, e.g., by a bit flip (so called soft error) or a hardware memory error affecting the program instructions.
- *Altered SUT program operations timing/ordering*: the operations of the SUT are scheduled in an unforeseen way (e.g., activating a synchronization bug such as a deadlock or race condition).
- *Interaction wrong value*: in a direct interaction (i.e., request/reply) with another environment software components (namely, with a library, with a middleware or a virtual machine service request, or OS system calls), the reply's value or data type (e.g., a parameter) is not as expected. This may happen because of faults in the interacting software (e.g., a library is corrupted) or because of unforeseen interactions.
- *Interaction wrong timing or omission*: in a direct interaction with other environment software components, an expected reply is provided earlier or later than needed (given two thresholds). We include, in this category, also the omitted reply case, e.g., due to a problem in the interacting application (such as a library not present, an interacting application crashed).
- *Resource request delayed or denied*: this is a special case of the previous category, whenever the interaction regards a resource request (hence the interacting unit regards a subset of the OS system calls), which is denied or delayed (e.g., disk unavailable, memory access denied, network access is delayed).

TABLE II
ENVIRONMENT FAULT OPERATORS

Mode / *Env. component*	Data corruption	SUT Prog. corruption	SUT Program ops. tim/ord	Interaction wrong value	Interaction tim/omission	Resource denied/delayed
OS memory mng.	MEM-DC	MEM-PC	–	MEM-IWV	MEM-TIM, MEM-OM	MEM-R_DEN, RES-R_DEL
OS device dirvers	DRV-DC	DRV-PC	–	DRV-IWV	DRV-TIM, DRV-OM	DRV-R_DEN, DRV-R_DEL
OS filesystem	FS-DC	FS-PC	–	FS-IWV	FS-TIM, FS-OM	FS-R_DEN, FS-R_DEL
OS network	NET-DC	NET-PC	-	NET-IWV	NET-TIM, NET-OM	NET-R_DEN, NET-R_DEL
OS process mng.	PROC-DC	PROC-PC	PROC-P_TIM/ORD	PROC-IWV	PROC-TIM, PROC-OM	PROC-R_DEN, PROC-R_DEL
Other system sw	SYS_SW-DC	SYS_SW-PC	–	SYS_SW-IWV	SYS_SW-TIM, SYS_SW-OM	–
Middleware	MW-DC	MW-PC	–	MW-IWV	MEM-TIM, MEM-OM	–
App-level sw	APP-DC	APP-PC	–	APP-IWV	APP-TIM, APP-OM	–
Hardware	HW-DC	HW-PC	–	HW-IWV	HW-TIM, HW-OM	–
User	–	–	USR-P_TIM/ORD	–	USR-TIM	–

Operators are obtained by crossing the *modes* with the environment components that cause the conditions to happen. Table II lists the operators we identify. The same *mode* is applied in the same way to different environment components (e.g., a corrupted parameter in a memory management or process management system call). Moreover, for each operator, we distinguish if it is a *direct* or *indirect* condition, depending on whether the altered condition impacts the application state also before the bug activation (e.g., a corruption of a shared variable), or not (e.g., disk full due to tasks by other applications)[4] – a topic addressed, differently, also in [13]. Each operator represents the last erroneous condition before the propagation to the SUT. For instance, any kind of fault, even from different environment parts, can cause a disk access to be delayed, but the last environment component before the error propagation to the SUT is the OS system call requiring access to the disk: operators do not represent the original cause of the fault, but the last erroneous condition before the propagation to the SUT (this is the reason why not all the possible crosses make sense in the Table). This does not mean that the way we inject anomalous conditions is necessarily an *error* injection approach: we can adopt *fault injection* or *mutation testing* to cause the desired erroneous condition whenever it is more convenient (for instance, if it is less expensive or more representative). Therefore, all the mentioned techniques can be exploited to understand *how* to implement a specific operator. In the next section, a discussion about existing technologies to implement the operators are discussed.

V. INJECTION TECHNOLOGIES

There are valuable technologies that can be used to effectively implement operators in the identified environment components. We survey some of the most relevant ones:

- *NFTape*: it can inject CPU, memory, and I/O faults (e.g., data corruption into memory, registers, and disk) to assess the fault tolerance of a system; it can also create stressful conditions thanks to a synthetic workload generator unit.
- *FERRARI*: it uses software traps to inject CPU, memory, and bus faults; when a trap is triggered (e.g., by specific memory location access or a timeout), it injects a data corruption in the selected memory location or register, resulting in either a permanent or transient fault.
- *DOCTOR*: DOCTOR can inject memory and register faults, but also network communication faults (using, as triggers, a combination of time-out, trap and code modification). It can emulate both instructions and data corruption, both transient and permanent faults.
- *Ballista*: Ballista is a robustness testing tool; it adopts a data-type based fault model, defining a subset of invalid values for every data type. It was successfully applied to assess the robustness of commercial OSs with respect to the POSIX system call interface; it is suitable to test the APIs of COTS software.
- *Riddle*: it provides an environment to combine random input, malicious input, and boundary values to test the system under anomalous conditions; it was successfully used to assess the robustness of native Windows NT system utilities and Win32 ports of the GNU utilities.
- *ORCHESTRA*: for testing distributed real-time systems; it is a script-based tool to inject faults in network protocols, by intercepting messages and injecting message corruption or delays.
- *Xception*: it uses the processor's exception handling capabilities to trigger fault injection, requiring no modification to the software. Faults are triggered upon access to specific memory locations, which could be either for data or fetching instructions, making tests accurately reproducible. Time-based triggers (timeouts) are also supported.
- *LFI*: it is a library fault injection tool; it identifies errors exposed by shared libraries, finds potentially buggy error recovery code, and injects faults at the boundary between shared libraries and applications.
- *CFIT*: it is a concurrent fault injection tool, which generates program mutants based on four common concurrent fault patterns as mutation operators.
- *ExhaustiF*: it is a commercial tool able to inject faults into both software and hardware, distinguishing software faults into variable corruption and procedure corruption, and hardware faults into Memory (I/O, RAM) and CPU (Integer Unit, Floating Unit).

[4]At this time, we are not able to define operators distinguishing the *activation/propagation* cases, for which complex propagation analyses would be required. This is left to future research.

- *Codenomicon Defensics*: a commercial tool that injects faults in more than150 different interfaces including network protocols, API interfaces, files, and XML structure.
- *SAFE*: differently from the previous ones, it allows inserting faults at source code level, according to fault types derived from the orthogonal defect classification (ODC) scheme; several operators are defined whose aim is to faithfully represent programming mistakes.
- *μJava*: a well-known mutation testing tool for Java programs, including both traditional mutation testing operators and class-level operators.
- *PIT*: a bytecode-based mutation testing tool for Java, working with Ant, Maven, Gradle and other project and build management systems.
- *Proteum*: a family of testing tools supporting mutation analysis; it can be configured to test programs in many procedural languages, with many mutation operators.

There are many other tools for fault injection, robustness testing, and mutation testing, both at binary- and at source-code level, at runtime or compile-time. This subset gives an idea of the several possibilities that can be used. For instance, binary-level data and instruction corruption tools (such as NFTape, FERRARI, DOCTOR, *Xception*, *ExhaustIF*) are very useful for the data corruption error injection approach, and can implement many of the defined operators (Table II) effectively, since the main execution environment components (e.g., memory, I/O) are covered. Network errors are, for instance, covered by tools like ORCHESTRA, *Codenomicon Defensics*, DOCTOR. Errors at interface level can be effectively injected by robustness testing solutions (namely, BALLISTA, RIDDLE, *Xception*, as well as *Codenomicon Defensics*); LFI is able to inject errors in shared libraries; RIDDLE has been tested on several software systems utilities; CFIT is designed to inject concurrency faults, that can be related to the timing/ordering operators; soruce-code fault injection, such as SAFE and mutation testing tools, are particularly useful to insert programming faults into interacting application, middelware, libraries, or system software, well-suited to emulate the case where programming errors in environment applications cause an error at the interface propagating to the SUT. It is likely that more tools should be used together to cover the environment fault operators, and different tools can be used depending on several criteria: the objective (e.g., improve the test suite, assess fault tolerance), the availability of source code, the need for representativeness (e.g., if artificial faults are required to be closer to hardware faults, as might be the case of well-tested software with low likelihood of having software faults, or closer to programming mistakes, as in the case of poorly tested software), the context (e.g., a safety-critical context might require completeness of operators at the expense of testing cost due to the application of many different operators). Since we are discussing a conceptual framework, the room for customization is wide open; in the following we highlight few major challenges we believe can be addressed in the near future for such a technique to be useful.

VI. Challenges and future research

The sketched framework opens several challenges and research directions that can be pursued; some are more generally referred to mutation testing and fault injection, while others are specific to our approach. We outline five main trends:

1) *Toolchain design*: we mentioned several tools that address specific aspects useful to alter the environment in accordance with the defined operators. A valuable future direction is to combine the benefits of these tools into a unique framework, where, depending on the operator and on the mentioned criteria, an injection technique is applied among SWIFI, G-SWFIT, mutation testing, robustness testing. Due to the extent of execution environment, which includes software at various abstraction levels as well as hardware and network, a holistic approach is needed to create effective "mutants".
2) *Refinement of operators*: defined operators are intentionally under-specified. There are numerous fault models that the mentioned techniques apply, depending on the SUT and on fault classes they are interested in. Our future research will focus on current operators to investigate whether suitable sub-classes, resembling the commonly experienced faults at environment level, should be defined. An immediate refinement could be to detail the semantic of a "corruption" and "wrong value". For instance, approaches of error injection (like robustness testing) distinguish several types of parameter corruption occurring at interface level, e.g., syntactic corruptions or semantic alteration. However, the refinement of operators makes sense depending on the objective: in fact, the definition of sub-classes might be different if we want to enrich a test suite or to test the SUT for fault tolerance [19]. Moreover, the explosion of mutants needs always to be balanced with the cost of applying them.
3) *Injection procedures*: a further line of research regards the way of inserting faults. The traditional problems of fault injection and mutation testing are transferred to this context, in that the analysis of *when*, *where*, and *how* to inject faults is even more more tough. In this case, several environment components are involved, possibly affecting each other; the main challenge is to identify locations and time instants more likely to impact the SUT in order to avoid useless injections (e.g., by exploring static or dynamic analysis techniques).
4) *Cost reduction*: as in conventional mutation testing, there is a constant need for devising techniques to reduce the cost for generating mutants and thus to the efficiency of fault operators. In the case of environment, this challenge is exacerbated as the involved code can be huge. Current techniques to reduce the number of mutants without significant loss of test effectiveness (e.g., mutant sampling, mutant clustering, selective mutants) should be tailored for the environment mutation analysis. Again, the objective of the mutation can make one technique better than others, as the notion of "test effectiveness"

changes. Suitable approaches need to be developed to control such a trade-off between cost and test effectiveness, which will be likely tied to how effectively the previous points are addressed.

5) *Higher-order mutation (HOM)*: the most complex faults are activated as consequence of several environment conditions (i.e., triggers) occurring together. An attractive research direction is to figure out how to combine more injections in one experiment while controlling mutual dependencies, e.g., masking effect (similarly to higher order mutation testing). Along this line, the understanding of subsumption relations among environment faults and the development of efficient algorithms (e.g., search-based optimization) techniques to look for the hard-to-detect combinations of simple faults that partially mask one another are valuable topics. Applying HOM to the environment would allow emulating very subtle circumstances – a useful means for safety-critical systems.

The outlined challenges suggest that there is long way to go. However, in any future testing technique, the mutual mingling of software systems with its surrounding execution (and physical) environment, as well as the interaction with other systems, make it unimaginable to reason in terms of "closed" systems. The spread of paradigms as the Internet of things and the development of cyber-physical systems, exposed to large and changing environments, demands an integrated notion of software systems, in which the environment is required to be in the loop of development and testing as an active and subtle contributor to software-related system failures.

ACKNOWLEDGMENT

This work has been partially supported by the PRIN Project "TENACE" (n. 20103P34XC) funded by the Italian Ministry of Education, University and Research. Trivedi's research was supported in part by a NASA NSSC Grant number NNX14AL90G.

REFERENCES

[1] M. Grottke and K.S. Trivedi. Fighting bugs: Remove, retry, replicate, and rejuvenate. IEEE Computer, 40(2):107–109, 2007

[2] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, Hints on test data selection: Help for the practicing programmer, IEEE Computer, 11(4), 34–41, 1978

[3] J. H. Andrews, L. C. Briand, and Y. Labiche, Is mutation an appropriate tool for testing experiments?, in International Conference on Software Engineering (ICSE), pp. 402–411, 2005

[4] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford, Design of mutant operators for the C programming language, Purdue University, Tech. Rep. SERC-TR-41-P, 2006

[5] A.J. Offutt, A. Lee, G. Rothermel, R.H. Untch, and C. Zapf, An Experimental Determination of Sufficient Mutant Operators, ACM Trans. Software Engineering and Methodology, 5(2), 99–118,1996

[6] W.E. Wongand, A.P.Mathur, Reducing the Cost of Mutation Testing: An Empirical Study, J. of Systems and Software, 31(3), 185–196, 1995

[7] M.Sridharanand, A.S. Namin, Prioritizing Mutation Operators Based on Importance Sampling, Proc. 21st Intl. Symp. of Software Reliability Engineering, pp. 378–387, 2010

[8] Y. Jiaand, M.Harman, An Analysis and Survey of the Development of Mutation Testing, IEEE Trans. Software Eng., 37(5), 649–678, 2011

[9] Huy Vu Do, C. Robach, M. Delaunay, Mutation Analysis for Reactive System Environment Properties, 2nd Workshop on Mutation Analysis, pp. 7–10, 2006

[10] H. Fouchal, A. Rollet, A. Tarhini, Robustness of Composed Timed Systems, SOFSEM 2005: Theory and Practice of Computer Science, LNCS Vol 3381, 2005, pp. 157–166

[11] E. H. Spafford. 1990. Extending mutation testing to find environmental bugs. Softw. Pract. Exper. 20, 2, 1990, pp. 181–189.

[12] A. Sung, and B. Choi, Interaction Testing in an Embedded System Using Hardware Fault Injection and Program Mutation, Formal Approaches to Software Testing, LNCS Vol. 2931, 2004, pp. 192–204

[13] W. Du, A.P. Mathur, Testing for Software Vulnerability Using Environment Perturbation, in Proc. of the International Conference on Dependable Systems and Networks (DSN 2000), Workshop On Dependability Versus Malicious Faults, pp. 603–612, 2000.

[14] M. Hsueh, T.K. Tsai, and R.K. Iyer, Fault Injection Techniques and Tools. Computer 30 (4), 75–82, 1997.

[15] D. Stott, B. Floering, D. Burke, Z. Kalbarczyk, and R.Iyer, NFTAPE: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors, Proc. Int'l Computer Performance and Dependability Symp., pp. 91–100, 2000

[16] J. Carreira, H. Madeira, and J.G. Silva, Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers, IEEE Trans. Software Eng., 24(2), 125–136, 1998

[17] G.A. Kanawati, N.A. Kanawati, and J.A. Abraham, FERRARI: A Tool for the Validation of System Dependability Properties, Proc. 22nd Ann. IEEE Int'l Symp Fault-Tolerant Computing, pp. 336-344, 1992

[18] S. Han, K.G. Shin, and H.A. Rosenberg, Doctor: An Integrated Software Fault-Injection Environment for Distributed Real-Time Systems, Proc. Second Annual IEEE Int'l Computer Performance and Dependability Symp., IEEE CS Press, Los Alamitos, Calif., pp. 204–213, 1995

[19] R. Natella, D. Cotroneo, J.A. Duraes, H.S. Madeira, On Fault Representativeness of Software Fault Injection, Software Engineering, IEEE Transactions on, 39(1), 80–96, 2013

[20] P. Koopman, J. DeVale, The Exception Handling Effectiveness of POSIX Operating Systems, IEEE Trans. Software Eng., 26(9), 837–848, 2000

[21] A.K. Ghosh, M. Schmid, and V. Shah, Testing the Robustness of Windows NT Software, Proc. 9th IEEE Int'l Symp. Software Reliability Engineering, pp. 231-236, 1998

[22] J.A. Duraes and H. Madeira, Emulation of Software Faults: A Field Data Study and a Practical Approach, IEEE Trans. Software Eng., 32(11), 849–867, 2006

[23] Software Fault Emulation Tool, http://www.mobilab.unina.it/SFI.htm

[24] D. Cotroneo, R. Pietrantuono, S. Russo, Testing techniques selection based on ODC fault types and software metrics, Journal of Systems and Software, 86(6), 1613–1637, 2013

[25] Pietrantuono, R., Russo, S., Trivedi, K.S.: Online Monitoring of Software System Reliability. In: Proc. of the European Dependable Computing Conference (EDCC), 209–218, 2010

[26] S. Chandra and P. M. Chen. Whither generic recovery from application faults? A fault study using open-source software. In Proc. 2000 Intl. Conf. Depend. Sys. Netwks., pp. 97–106, 2000

[27] Bovenzi, A.; Cotroneo, D.; Pietrantuono, R.; Russo, S., On the Aging Effects Due to Concurrency Bugs: A Case Study on MySQL, Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on, pp. 211–220, 2012

[28] L. Li, K. Vaidyanathan, and K. S. Trivedi, An approach for estimation of software aging in a web server, in Proc. the Int. Symp. Empirical Soft. Engineering, 2002, pp. 91–100.

[29] Cotroneo, D., Orlando, S., Pietrantuono, R. and Russo, S. (2013), A measurement-based ageing analysis of the JVM. Softw. Test. Verif. Reliab., 23(3), 199–239, 2013

[30] Bovenzi, A.; Cotroneo, D.; Pietrantuono, R.; Russo, S., Workload Characterization for Software Aging Analysis, Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on, pp. 240–249, 2011

[31] M. Grottke, A. Nikora, and K. Trivedi, An empirical investigation of fault types in space mission system software, in IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN), 2010, pp. 447–456.

[32] Cavezza, D.G.; Pietrantuono, R.; Russo, S.; Alonso, J.; Trivedi, K.S., Reproducibility of Environment-Dependent Software Failures: An Experience Report, Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on, pp. 267–276, 2014

[33] R. Love. Linux Kernel Development. Addison-Wesley, 3rd edition, 2010.