Architecture-based Criticality Assessment of Software Systems

Authors Name/s per 1st Affiliation (Author)
line 1 (of Affiliation): dept. name of organization
line 2: name of organization, acronyms acceptable
line 3: City, Country
line 4: Email: name@xyz.com

Authors Name/s per 2nd Affiliation (Author)
line 1 (of Affiliation): dept. name of organization
line 2: name of organization, acronyms acceptable
line 3: City, Country
line 4: Email: name@xyz.com

***Abstract*—The development of dependable software systems with acceptable costs and time often requires architectural analyses and criticality assessment strategies to achieve a detailed understanding of the system failing behavior. This information is valuable to evaluate architectural alternatives, to allocate resources, such as, testing efforts and fault tolerance means, efficiently, and to support the selection and the integration of software components.**

This paper proposes an architecture-based method, which allows characterizing the criticality of individual components and their impact on the overall system. The method includes a preliminary system characterization through an extensive software fault injection campaign and finally exploits a mathematical description of the system, in terms of components and interactions among them, which enables the overall criticality analysis. The method is applied to two real-world case studies (i) the Apache Web Server, and (ii) TAO Open Data Distribution Service (DDS).

***Keywords*-component; formatting; style; styling;**

I. Introduction

With complex software systems being employed in diverse and critical applications and with increasingly stringent market requirements, the achievements of good trade-offs between systems' dependability and cost to obtain it is becoming essential to engineers. Crucial choices during the development of software systems, which impact these trade-offs, are often determined by how much the available knowledge about the system's behavior is complete and accurate.

Adequate analysis is of paramount importance to figure out the behavior of single components and the architectural interrelationships, in order to be able to evaluate alternatives and take appropriate actions regarding, architectural choices, resources allocation (e.g., for testing and/or fault tolerance means), Off-The-Shelf (OTS) items integration, and so on.

Due to the relevance of component-based design solutions, such an analysis cannot disregard the software architecture, in that it determines how the different components interact to each other and strongly contribute to the way the system behaves and performs.

This paper proposes a method to characterize software architecture in terms of its most critical failure modes, most critical software components with respect to each considered failure mode, and most critical fault type per each component. This allows developers to obtain a set of information about the system being developed, and enables them to take actions tailored to their software architecture, regarding:

- **Architectural alternatives**: engineers can evaluate if different compositions of new components or the possible integration of off-the-shelf software items can improve the failure behavior of the system.
- **Architectural bottlenecks**: by carrying out a sensitivity analysis on the failure behvior of single components, it is possible to pinpoint which component mainly impacts the system's failure behavior.
- **Resources allocation**: by knowing the most failure-prone components, the most relevant failure modes per component, as well as the most critical fault types, engineers can allocate resources for testing and/or for fault tolerance mechanisms optimally, both regarding the amount of resources and about their type (i.e., *where* to allocate more, and *what* to allocate to each component); e.g., different testing techniques for modules sensitive to different fault types, or different FT means for modules prone to different failure modes.

As first step of this method, the software architecture is modeled by using a hierarchical architecture-based solution; this represents the system's components and how they interact to each other to carry out the intended system's function. Then through an extensive software fault injection campaign we identify the most critical software modules, as well as related fault types, which heavily impact the correct behavior of a system during the operational time. Indices provided by the fault injection campaign, along with architectural model, allows figuring out the impact of each component on that specific architecture from *failure proneness* and *fault tolerance* perspectives.

The method is applied to two distinct case studies: (i) the popular Apache Web Server, which availability level is crucial for many web-based applications, and (ii) TAO Open Data Distribution Service (DDS), an open source implementation of OMG's DDS specification [1], also em-

[1]—

ployed in safety-critical contexts [rif.]. Results shows all the information it is possible to obtain, highlighting the criticalities of both applications about their components, their failure modes and fault types most critical for them. The paper is organized as follows: —–

II. Background and Related Work

Current trends in software development exacerbate the role of *software faults* as main responsible of system failures [4], [3]. In fact, it may not be possible to validate a complex system *solely* by means of testing both due to the stringent time-to-market and technical limitations. As a result, software is likely to be released with *residual* software faults [5]. Additionally, the massive use of Off-The-Shelf items, which allow complex applications to be designed by integrating services and components rather then building them entirely *from scratch*, does not provide specific dependability guarantees, because of complex interactions with the system after their integration [1], [2].

Mentioned problems make dependability a significant challenge, especially in case of complex and critical systems. To this aim, several organizations, such as, [6], [7], [8] formalized standards and methodologies to support the development of dependable systems. More in details, they exhaustively define a set of tasks and evidences to be produced during all the phases of the software development cycle. However, these activities may be extremely time consuming, thus neglecting the actual needs of current software industry.

As stated, we believe that both testing and validation efforts may be effectively driven with a preliminary knowledge of the system, in terms of its critical components. Mentioned standards usually suggest *hazard analysis* and *risk assessment* techniques, such as, failure modes and effects analysis (FMEA), hazard and operability (HAZOP), event tree analysis (ETA), and fault tree analysis [15]. In [16] and [9] authors describe the hazard analysis methodology defined and used in railway dependable systems. In [10] safety assessment processes of ATC software systems have been proposed.

Several works propose approaches based on a dynamic flow graph methodology (DFM) [11], [12] to generate timed fault trees, for assessing risks associated with dynamic behaviors. Additionally, methodologies and/or technologies for safety assessment of real complex critical system infrastructures and operations have been proposed. Authors in [14] present a case study to apply a goal-oriented method for car security related hazard analysis. [13] has proposed a model based on a conceptual network representation, where objects represent concepts and links represent relations.

It should be noted that some issues may compromise the effectiveness of existing approaches in real-world scenarios. As for example, the DFM analysis does not have mechanisms to cope with computational complexity when dealing with large-scale software systems. Furthermore, the risk assessment phase is often performed by examining only faults at the I/O interface level without considering mitigation means present in the system architecture. To overcome these limitations, the proposed approach relies on a high-level system model, whose grain is completely up to the analyst. Additionally we make extensive use of software fault injection to emulate actual programming mistakes and we assess the system behavior with respect to these faults in order to achieve insights of its mitigation capabilities.

III. System model

Criticality assessment is driven by a system model, in order to achieve insights of the critical entities. More in details, we identify the main functional *components* of the systems along with the *interactions* among them.

We achieve insight of the internal architecture of the platform under-analysis through the available documentation. Accordingly, a set of **entities** is defined with respect to the design (Figure 1 - A). An entity is a component of the platform under analysis and it provides a well-defined capability [36]. Each entity is finally mapped to the source modules of the platform under analysis providing the capability associated to the entity. For reason of consistency with the notations, all the modules, which cannot mapped to an entity, namely, they provide a capability which the analyst is not intended to deal with, are mapped into a *special* entity. Entity thus represent the granularity for the subsequent proneness and criticality assessment.

It should be noted that the provided definition is general. Analyst can clearly specialize it according to their needs as well as the objectives of the analysis. Furthermore the analysis may encompass the system as a whole, thus including all the possible scenarios. The finer grain we hypothesize would be to map each entity with exactly one source file of the system. In fact this is clearly a trade-off between a large number of small units and a small number of large units.

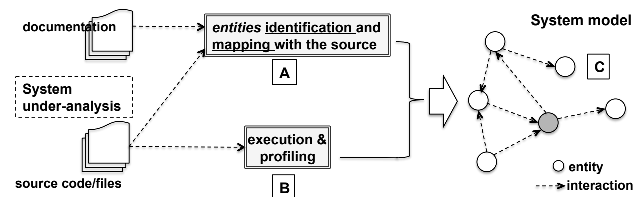

Figure 1: System modeling. Overview.

Once the high-level picture has been achieved via the entities identification, we execute and profile the system in order to define the interactions among them (Figure 1 - A). This is done by using well know profiling tools. In the context of this work we used `gprof` [37]. The profiling information is subsequently used to identify the interactions among the entities in the system. An **interaction** exists between two entities A and B, if, according to the profiling

information, a function provided by a module mapped into B is invoked within the code mapped to A. The joint information provided by the entities and interactions allows to model the system architecture as a graph (Figure 1 - C). This graph, along with the detailed information about the number of interactions existing among the entities is finally fed to the criticality assessment phase.

IV. Proneness Estimation

We quantitatively evaluate the system behavior in presence of faults. In particular we quantify the ability of the system to keep correctly working when software faults of different natures are injected in the entities identified in the modeling phase. This phase gives a preliminary overview of the criticality of the identified entities.

To obtain such a characterization, the approach proposed in this paper includes the following steps:

1) *Failure Modes Definition*. Expected failure modes are identified by the analyst in this phase. According to the its mission, the definition of potential modes by which the system could fail, such as *crash, passive/active hang, value*, and so on. Let us denote these failure modes for the system S with (F_1, F_2, ..., F_k).
2) *Fault Injection*. In this step the fault injection campaign is carried out. We create and exercise a faulty version of the system and evaluate its behavior against the faults activation. Faults of various type have to be injected in the software, since we are interested in figuring out how a given fault type impacts on the system. To this aim, we classified faults according to a well-known classification scheme, namely the *Orthogonal Defect Classification* (ODC) [34], extended according to Madeira *et al.* [35]. During these experiments, all failure data are collected, along with the faults that caused them (type and location). Section IV-A detail how we conduct the fault injection experiments in the context of this work.
3) *Failure Analysis*. The following two steps allow to obtain the system characterization to be used for figuring out the entities which are critical. In this step, designers evaluate how the system failed during experiments, and what modules are more critical:
   - *Failure Modes Distribution*: collected failure data are classified according to their modes (F_1, F_2, ..., F_k) and to the responsible entity. Hence the output is a set of indexes $\#F_{j,e_i}$ denoting the number of failures of type j (among the identified failure modes) occurred due to a fault injected in the entity e_i.
   - *Failure Proneness Computation*: per each entity e_i, a failure proneness value indicating its criticality with respect to other entities is computed as follows:
   $$FP_{e_i} = \frac{\sum_{j=1}^{K} \#F_{j,e_i}}{\#F} \quad (1)$$
   where K denotes the number of failure modes, and $\#F$ is the total number of failures. This index represents the number of failures due to e_i over the total number of failures. It represents how much an entity caused the system failures, i.e., its *criticality* with respect other entities. The output of this step is therefore the list of entities criticality.
   Note that this criticality index can be obtained also with respect to each considered failure mode, (i.e., $FP_{e_i,j} = \frac{\#F_{j,e_i}}{\#F}$, indicating how much the entity e_i is responsible for failures of mode j).
4) *Tolerance* Per each module, it is important to figure out how much an entity caused a system's failure with respect to the number of faults injected in it, i.e., an indication of its *tolerance* to the presence of faults. The following index is computed:
   $$Tol_{e_i} = 1 - \frac{\sum_{j=1}^{K} \#F_{j,e_i}}{\#SeededFault_{e_i}} \quad (2)$$
   representing the number of failures due to the entity e_i, over the number of faults injected in e_i, i.e., $\#SeededFault_{ei}$ (namely, the number of tolerated faults).
   Note that also in this case we can consider a *tolerance* of the entity e_i with respect to each failure mode, getting to the following: $Tol_{e_i,j} = 1 - \frac{\#F_{j,e_i}}{\#Fault_{ei}}$, indicating how much the entity e_i is tolerant to failures of mode j.
5) *Fault Analysis* Per each module, it is important to figure out what types of fault caused, more likely, the observed failures, in order to adopt appropriate countermeasure. In this step, the most relevant fault types are identified, i.e., the ones mainly responsible for the failures: a further index is then computed, FC (i.e., Fault Criticality, FC):
   $$FC_{t,i} = \frac{\#Fault_{t,i}}{\#SeededFault_i} \quad (3)$$
   where $FC_{t,i}$ is the average criticality of faults of type t in the entity i. It is computed by considering $\#Fault_{t,i}$, which denotes the number of faults of type t in the entity e_i that caused a failure, $\#SeededFault_i$, which denotes the number of total faults injected in e_i that caused a failure. The index represents how much the fault type t impacted in the average the observed number of failures due to e_i.

At the end of these steps, designers will have a preliminary picture of the system behavior. More in detail, they know

Table I: Fault operators [35]

Acronym	Explanation
OMFC	Missing function call
OMVIV	Missing variable initialization using a value
OMVAV	Missing variable assignment using a value
OMVAE	Missing variable assignment with an expression
OMIA	Missing IF construct around statements
OMIFS	Missing IF construct plus statements
OMIEB	Missing IF construct plus statements plus ELSE before statements
OMLC	Missing clause in expression used as branch condition
OMLPA	Missing small and localized part of the algorithm
OWVAV	Wrong value assigned to variable
OWPFV	Wrong variable used in parameter of function call
OWAEP	Wrong arithmetic expression in parameter of a function call

the average criticality index FP_{e_i} per entity, the tolerance index, which indicates how the entity behaves with respect to the seeded faults, and the fault types criticality indicating those faults more responsible for the observed failures.

A. *Fault Injection Campaign*

Software faults in the target platform by means of *changes* in the source code of the program. Changes are introduced according to specific *fault-operators* based upon actual faults uncovered in several open-source projects [35]. We recognize that this approach needs for the source code of the program in order to be applied, nevertheless its accuracy is greater than other fault injection techniques (e.g., G-SWIFT [35]). Table I summarizes fault operators.

Exactly one software fault is introduced in the target source code for each fault injection experiment. The target platform is compiled and its resulting *faulty* version is stored for the subsequent experimental campaign. We use a support tool[2] to automate the fault injection process.

Once completed the injection process, we execute the experimental campaign, which consists of 4 distinct logic phases (Figure 2), detailed in the following. In particular, for each faulty version of the target platform, a **campaign manager** program executes the following steps:

1) initializes the target platform;
2) starts a **tester** program, which exploits commonly-used platform capabilities by running a specific workload;
3) identifies the *experiment outcome*. In particular, the campaign manager figures out the specific failure (among the failure modes choosen by the analyst (step 1 of the method), if any, resulting from the injected fault. To this aim it exploits both (i) *operating system* level information (e.g., memory dumps generation) and (ii) *tester* level information (e.g., a timeout expiration, computation results provided by the platform in hand to the tester program);
4) produces a **report** containing information intended to be used during the analysis phase (e.g., injected fault type, target source file, experienced failure mode) and cleans up the system for the successive fault injection experiment.

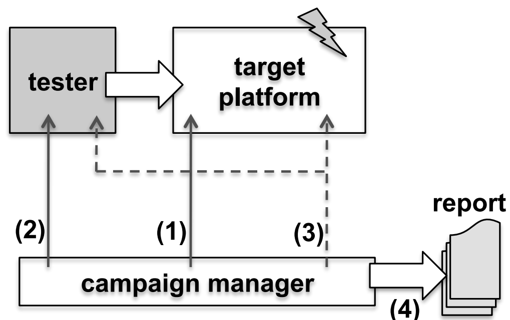

Figure 2: Experimental campaign

Machines composing the experimental testbed (Intel Pentium 4 3.2 GHz, 4GB RAM, 1,000 Mb/s Network Interface equipped) run a RedHat Linux Enterprise 4. An Ethernet LAN interconnects the machines. Platform-specific details (e.g., failure modes, and the used tester program), when needed, will be clarified in the context of the reference case-studies.

V. Criticality Assessment

Entities and interactions discussed in Section III represent the architecture as a graph. However, to obtain a mathematical description, we map the graph onto an absorbing Discrete Time Markov Chain (DTMC), as in [**?**], [**?**], [**?**], [**?**]. Assuming that an application has n entities, with the initial entity indexed by 1 and the final entity by n, DTMC states represent the entities and the transition from state i to state j represents the transfer of control from entity i to entity j, with the associated transition probability. A useful measure to develop suitable mathematical models is the *visit count*, i.e., the expected number of times an entity is visited during an execution. One way to compute it is to consider transition probabilities, through the one-step transition probability matrix of the absorbing DTMC. Partitioning such a matrix (with n states and m absorbing states) as:

$$P = \begin{pmatrix} Q & C \\ 0 & I \end{pmatrix} \quad (4)$$

[2] http://www.mobilab.unina.it/SFI.htm

we have a submatrix Q, that is an *(n-m)* by *(n-m)* stochastic matrix, an *m* by *m* identity matrix I, a *m* by *(n -m)* matrix of zeros and a matrix C, that is an *(n-m)* by *m* matrix.
Denoting with P^k the k-step transition probability matrix (where the entry *(i,j)* of Q^k represents the probability of arriving in the state j from the state i after k steps), and with $X_{i,j}$, the number of visits from the state *i* to *j* before absorption, it can be shown [**?**], [**?**] that the $m_{i,j}$ entry of the fundamental matrix M corresponds to the expected number of visits from i to j, i.e., $v_{i,j}$ = E[$X_{i,j}$]. Thus, the expected number of visits starting from the initial state to the state *j* is:

$$v_{1,j} = m_{1,j} \tag{5}$$

We denote visit count for entity j as $V_j = v_{1,j}$. Such values are particularly useful to describe the usage of each application's entity.

Failure Proneness
The DTMC representation, along with the concept of visit counts, has been used to express the system's failing behavior as a function its entities' failure behavior.
In particular, we consider the random variable $FP_{j,e_i} = \frac{\#F_{j,e_i}}{\#F}$, (where #F denotes the total number of failures), as the probability that the, given that the application failed in one execution, it failed with failure mode j and because of entity e_i; then we consider the probability that it failed due to e_i with one of the considered failure modes: $FP_{e_i} = \frac{\sum_{j=1}^{K} \#F_{j,e_i}}{\#F}$.
The impact of a given failure mode on the entire system's failure could be simple obtained as: $FP_j = \frac{\sum_{i=1}^{n} \#F_{j,e_i}}{\#F}$, i.e., how many failure of mode j occurred over the total number of failures. However, if we want to know what is the *actual* impact of a given failure mode on the entire system's failure, we have to consider not only the single entities, but also how they interact (i.e., their usage). A suitable way to combine entities' criticality with their usage is through visit counts. Hence we define the modified index $FPV_{j,e_i} = FP_{j,e_i}/V_i$, indicating the *failure proneness per visit*, i.e., the probability that e_i causes a failure of mode j during one visit to it. Then, by an approach similar to the one adopted in [] for security, we find the proneness of a system to fail with a given failure mode (given that it failed), we compute the following:

$$FP_j = 1 - \prod_{i}^{n} (1 - FPV_{j,e_i})^{X_{1,i}}. \tag{6}$$

This expression corresponds to the probability that the system fails with failure mode j, given that it failed, hence indicates what is the most critical failure mode. The product indicates that the complement of failure proneness of entities is multiplied by itself (i.e., raised to the power) as many times as the number of times they are visited. We are interested in the expected value of this expression; hence, we have:

$$E[FP_j] = 1 - \prod_{i}^{n} E[(1 - FPV_{j,e_i})^{X_{1,i}}] \tag{7}$$

Expanding the term in the product by the Taylor series, and recalling that $E[X_{1,i}] = V_i$ we have:

$$\begin{aligned} E[(1 - FPV_{j,e_i})^{X_{1,i}}] = (1 - FPV_{j,e_i})^{V_i} + \\ + \tfrac{1}{2}((1 - FPV_{j,e_i})^{V_i}) \\ \times (log(1 - FPV_{j,e_i}))^2 \sigma_{1,i}^2 \end{aligned} \tag{8}$$

The expected number of visits to the last entity e_n (which represents the absorbing state of the DTMC) is always 1, hence $E[X_{1,n}] = V_n = 1$ and $Var[V_n] = \sigma_{1,n}^2 = 0$. The expression 7 becomes:

$$\begin{aligned} E[(1 - FPV_{j,e_i})^{X_{1,i}}] = 1 - [\textstyle\prod_i^{n-1} [(1 - FPV_{j,e_i})^{V_i} + \\ + \tfrac{1}{2}((1 - FPV_{j,e_i})^{V_i}) \\ \times (log(1 - FPV_{j,e_i}))^2 \sigma_{1,i}^2]](1 - FPV_{j,e_n}) \end{aligned} \tag{9}$$

The expression also takes into account the second-order architectural effects (through the variance); neglecting such term as in[**?**] Equation 9 becomes:

$$E[FP_j] \approx 1 - \left[\textstyle\prod_i^{n-1} (1 - FPV_{j,e_i})^{V_i}\right] (1 - FPV_{j,e_n}) \tag{10}$$

Now, if we want to consider from this expression the actual probability (i.e.., taking into account the visits) that, if the system failed, it failed due the entity e_i, we can express it as in the Equation 9, i.e.:

$$\begin{aligned} 1 - E[(1 - FPV_{e_i})^{X_{1,i}}] = 1 - (1 - FPV_{e_i})^{V_i} + \\ + \tfrac{1}{2}((1 - FPV_{e_i})^{V_i}) \times (log(1 - FPV_{e_i}))^2 \sigma_{1,i}^2 \end{aligned} \tag{11}$$

Neglecting the variance of the number of visits, and expanding the term $(1 - FPV_{e_i})^{V_i}$ again according to the Taylor series, we get to:

$$\begin{aligned} 1 - E[(1 - FPV_{e_i})^{X_{1,i}}] \approx \\ 1 - \left[1 - \textstyle\sum_k^{\infty} \binom{V_i}{k} (-1)^k FPV_{e_i}^k\right] \end{aligned} \tag{12}$$

This represents the actual failure proneness of entity e_i. However, since the higher order terms are typically very small, Equation 12 can be conveniently expressed as:

$$\begin{aligned} 1 - E[(1 - FPV_{e_i})^{X_{1,i}}] \approx \\ 1 - [1 - V_i FPV_{e_i}] = FPV_{e_i} * V_i = FP_{e_i} \end{aligned} \tag{13}$$

which corresponds to the index FP_{e_i}, obtained as number of failures due to e_i over the number of total failures.

Tolerance

Similarly to the *proneness* estimate, to analyze the system failing behavior with respect to the injected faults, we consider the modified index $TolV_{e_i} = 1 - \frac{\sum_{j=1}^{k} \#F_{j,e_i}}{\#SeededFaults_{e_i}} * \frac{1}{V_i}$, indicating the tolerance per visit.

In particular, we are interested in estimating system's failure probabilities in presence of faults. Starting from single *intolerance* value, we compute:

$$Intol = 1 - E[Tol] \approx 1 - \prod_{i=1}^{n-1} (TolV_{e_i})^{V_i} (TolV_{e_n}) \quad (14)$$

This represents the expected probability that the system fails with any failure modes due to the presence of a fault per each entity. The expression neglects the second-order architectural effects (as in the Equation 10), that can be accounted for as in the Equation 9.

By the tolerance value, it is also possible to compute the expected probability that the system fails (with any failure mode) due to the presence of one fault in the system (in one of the n entities):

$$Intol' \approx [\sum_{i=1}^{n-1} P_{Fault_{e_i}} (1 - TolV_{e_i})^{V_i}](1 - TolV_{e_n}) \quad (15)$$

where $P_{Fault_{e_i}}$ is the probability that a fault is present in the entity e_i. Assuming the same probability for a fault to be present in an entity, $P_{Fault_{e_i}}$ is $\frac{1}{n}$ [QUESTO POSSIAMO DIRLO NELLA PARTE SPERIMENTALE— CIOE' NEL NOSTRO CASO E' 1/n]. Similarly, we could distinguish an *intolerance* value with respect to different failure modes, by replacing $TolV_{e_i}$ with $TolV_{e_i,j}$ in the above two expressions, obtaining the probability that the system fails with a specific failure mode j due to the presence of a fault in each entity or of one fault in the system, respectively.

Faults Criticality

The last source of information is represented by what we called the *Fault Criticality*. We consider the index $FCV_{t,i} = FCV_{t,i}/V_i$, being the probability that fault of type t causes a failure in the entity i in one visit. Combining with usage information (i.e., with *visit counts*), we obtain:

$$E[FC_t] \approx 1 - \prod_{i=1}^{n} (1 - FCV_{t,i})^{V_i} \quad (16)$$

This means that faults of type t are critical for the system if it causes a failure in at least one entity. This random variable provides an indication of fault type criticality.

All these global indexes will be used in our case studies to obtain a complete characterization of their failing features.

VI. Apache Web Server

We present the detailed analysis for a widely used software platform, i.e., the Apache Web Server[3] (version 1.3.41).

[3]http://httpd.apache.org/

Table III: Experiments breakup by fault operator (Apache Web Server)

Operator	#	Operator	#
OMFC	819	OMIA	791
OMIEB	282	OMIFS	812
OMLC	325	OMVAE	1,149
OMLPA	2,183	OMVIV	65
OMVAV	221	OWPFV	1,148
OWAEP	361		=====
OWVAV	277	**Total**	**8,433**

We briefly describe tested-specific details. Experiments and analysis results are then presented.

A. Experiments

We deploy the Web Sever on a node and, according to Section IV-A, we exercise it by means of a well-known workload generator, namely, `httperf`[4]. This has been configured to exploit most of the features offered by the Web Server (e.g., virtual-hosts, multiple methods, cookies) by means of a specific stress workload.

We identify three failure modes for the reference platform:

- *crash*: unexpected termination of the Web Server. A memory dump is generated by the operating system;
- *hang*: the Web Server process is up but, (i) one or more HTTP requests supplied by the *httperf* tool or (ii) the Web Server *start* or *stop* phases, are not acknowledged within a *proper*, i.e., tuned before the campaign by means of several fault-free runs of the Web Server, timeout.
- *content*: all error conditions that are not the result of a hang or crash (e.g., wrong value delivered to the client).

We execute 8,433 fault injection experiments involving 17 source files under the `/src/main` folder, i.e., the core of the Web Server. Table III reports the experiments breakup by fault operator.

During the campaign, 1,395 experiments result in a failure outcome (i.e., 743, 101, and 551, crashes, hangs, and contents, respectively). Figure 3 depicts the relative failures distribution. Crash and content failures are the most likely to occur. We only experience 7% of faulty runs to result in a hang outcome, mainly due to the triggering of infinite loops in the code.

B. Indexes Estimation

Reports produced by the campaign manager are used to characterize single entities, by estimating the indices introduced in Section IV, i.e., FP_{e_i}, Tol_{e_i}, and $FC_{t,i}$ per each entity identified during the modeling phase. To this aim we use Equations 1-3 described in the steps of the proposed method.

[4]http://www.hpl.hp.com/research/linux/httperf/

Table II: Failure indices (Apache Web Server)

entity	#	source file(s)	locs	failures	*crash*	*hang*	*content*	FP_{e_i}	Tol_{e_i}	V_i
e_0	1	`http_config`	783	**207**	155	1	51	0.1484	0.7356	89.771
e_1	1	`http_core`	1004	**132**	73	2	57	0.0946	0.8685	42.078
e_2	1	`http_main`	1451	**183**	105	17	61	0.1312	0.8739	246.5
e_3	1	`http_protocol`	1217	**102**	70	21	11	0.0731	0.9162	37.017
e_4	1	`http_request`	562	**27**	16	5	6	0.0194	0.9519	14.301
e_5	1	`http_vhost`	271	**11**	11	0	0	0.0079	0.9594	0.8988
e_6*	11	`alloc, buff, ...`	3145	**733**	313	55	365	0.5254	0.7669	407.16
		total	8,433	1,395	743	101	551	1.0000	-	-

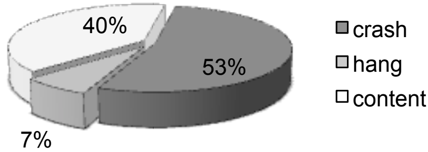

Figure 3: Failure modes distribution (Apache Web Server)

The two leftmost columns of Table II report a detailed perspective concerning the modeling phase. In the context of this work we focused the criticality analysis on most of the code concerning the *http requests* handling and the *configuration* section. Additionally a fine grain solution has been adopted. In fact each entity $e_0, ..., e_5$, is mapped to a specific source file of the platform. On the contrary, e_6 contains code mainly providing support capabilities to the platform, concerning 11 source files.

Column 3, 4, 5, 6 and 7 of Table II, report the experiment breakup by entity, i.e., the number of fault injected in the files mapped to a specific entity and the number of experiments that result in a failure outcome, as well as the number failures distinguished by failure modes. This information allows us to estimate indexes reported in column 8 and 9. The last column reports the visit counts value, useful in the next subsection for the overall criticality assessment.

Among the identified entities e_2, corresponding to the `http_main` source file exhibits the highest failure proneness and the lowest fault tolerance, according to the preliminary fault injection campaign. It should be not noted that these values are not necessarily due to the high number of failure injected in this entity, i.e., 1,451. In fact, the injection tool selects fault location in a fair and exhaustive way thus the number of fault is somewhat related to the size of the source code itself. Additionally, there exist some entities, e.g., e_0 whose proneness is greater than others, even if the number of injected fault is lower. For the same reason, the tolerance index does not depend on the number of injected faults; e.g., entity e_1 shows a greater tolerance than e_0 even with a greater number of faults injected.

Entity e_2 is among the entities mainly responsible for *crash* and *hang* failures.

ALTRI COMMENTI SIMILI AU FP E TOL— AGGIUNGERE POI DEI COMMENTI PER L'INDICE FAULT ANALYSIS, CHE NON RIPORTIAMO PER RAGIONI DI SPAZIO(CI VORREBBE UNA COLONNA PER OGNI TIPO DI FAULT—POSSIAMO DIRE DIRETTAMENTE I FAULT PIU' CRITICI COME IN EM4SOC)

C. Criticality Assessment

Starting from single entities characterization, we now take into account their interactions through the DTMC model representation. As for failure proneness, we first computed the FP_{j,e_i} values (as number of failures of mode j, columns 5, 6 and 7 of Table II over total number of failures); then the FPV_{j,e_i} (by considering the las column of Table II, i.e., *visit counts*). Through Equation 10, we computed the expected probability that the system failed of mode j, given that it failed, i.e., $E[FP_j]$, obtaining: $E[FP_{crash}] = 0.4131$, $E[FP_{hang}] = 0.0698$, $E[FP_{content}] = 0.3264$.

LA FORMULA VA CONTROLLATA – PROBABILMENTE LA SOMMA DI QUESTE TRE DEVE DARE 1

, confirming that *crash* failures are the most critical ones. By formulating the problem in this way, we can note that if we vary the architecture (either by introducing new components, or by changing the way they interact), the expected visit counts also vary, and, as a consequence, the final failure proneness indexes will vary. By doing a sensitivity analysis, we pinpoint the entity whose variation majorly determines a variation in the $E[FP_j]$ indexes*architectural bottleneck*. We found that for a variation of single failure pronenesses of 20 %, the entity e_x causes the greatest percentage variation in the index

$E[FP_{crash}]$, while entity e_y causes the greatest percentage variation in the index $E[FP_{hang}]$; entity e_z causes the greatest percentage variation in the index $E[FP_{content}]$.
As for tolerance, we consider equations 14 and 15. Results reveal that the probability that the system does not tolerate faults (i.e., it fails) given that there is a fault per each entity is: $Intol = 1 - E[Tol] = 0.6052$, (where single tolerance indexes are obtained from column 4 divided by column 3 values of Table II). While the probability that the system does not tolerate faults (i.e., it fails) given that there is one fault in the system is: $Intol' = 0.1215$. Also in this case, a sensitivity analysis reveals that —–
ULTERIORI COMMENTI

Finally, fault analysis revealed that the most critical fault type for Apache turned out to be the fault type, since it has the greatest FC_t global index.

VII. TAO Open DDS

AGGIUSTARE QUESTA SEZIONE COME NEL CASO DI STUDIO 1, SEGUENDO LA NUOVA TABELLA.

Case study 2 consists of a failure analysis of TAO Open Data Distribution Service (DDS)[5], i.e., a middleware platform an open source C++ implementation of the OMG's v1.0 DDS specification[6].

A. Experiments

We developed a DDS-based application acting as tester program, ccording to Section IV-A. It is composed by two processes, i.e., i.e., a *publisher*, which sends data bounded to a specific DDS topic, and a *subscriber* process, which receives them. Both the processes are build atop the DDS middleware which assures communication services between the described processes. It should be noted that the applicative code is simple and was accurately analyzed to make it faults-free. Again, we target only the middleware platform (linked to the processes in the form of shared library), in the context of this work.

We identify three possible failure modes:

- *crash*: unexpected termination of at least one of the DDS processes. A memory dump is generated by the operating system;
- *no messages* (no_msg, in the following): none of the messages sent by the publisher is delivered by the DDS middleware to the subscriber process within a proper timeout (i.e., tuned before the campaign by means of several fault-free runs of the DDS-based application);
- *value*: messages delivered to the subscriber process are different from the ones sent by the publisher.

We execute 5,892 fault injection experiments involving about 80 source files under the `/dds/DCPS` and

[5]http://download.ociweb.com/OpenDDS/
[6]http://www.omg.org

Table V: Experiments breakup by fault operator (TAO Open DDS)

Operator	#	Source	#
OMFC	1022	OMIA	440
OMIEB	306	OMIFS	324
OMLC	86	OMVAE	321
OMLPA	1921	OMVIV	254
OMVAV	614	OWPFV	396
OWAEP	24		=====
OWVAV	184	**Total**	**5,892**

`/dds/DCPS/transport` folders, i.e., the core of the DDS library. Table V reports the experiments breakup by fault operator.

During the campaign 1,685 experiments result in a failure outcome (i.e., 880, 712, and 93, crashes, no_msgs, and values, respectively). Figure 4 depicts the relative failures distribution. *Crash* and *no_msg* failures are the most likely to occur. *Value* failures are almost rare (only 6%). It should be noted that this behavior is somewhat different compared to the Apache Web Server (Figure 3). As a matter of fact, *hang* failures (i.e., the complete lack of service with the platform still up), resulted only in few experiments.

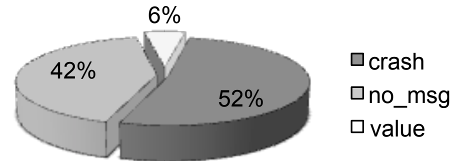

Figure 4: Failure modes distribution (TAO Open DDS)

B. Indixes Estimation

Reports produced by the campaign manager are used to estimate the *failure proneness*, i.e., FP_{e_i}, of each of the entities identified during the modeling phase. To this aim we use equation 1 described in the third step of the proposed method (Section **??**).

The two rightmost columns of Table **??** report a detailed perspective concerning the modeling result. In case of the DDS middleware we mainly focus on the architectural components providing the (i) publishing/sending, (ii) subscribing/receiving, (iii) data transmission capabilities. According to this high-level picture we identified 7 entities $e_0, ..., e_6$. In this case study we do not have a 1-to-1 mapping with a specific source file, thus a corse graining has been adopted. Again, the model is flexible and the graing maybe adjusted according to the analyst's needs. Finally e_7 contains 60 files which have not been included in a specific entity. This is treated has an entity as well.

Table IV: Failure proneness (TAO Open DDS)

entity	#	source file(s)	locs	**failures**	*crash*	*hang*	*content*	FP_{e_i}	Tol_{e_i}	V_i
e_0	3	`Service_[Domain|Participant]`	841	**250**	100	138	12	0.1484	0.7027	9.2538
e_1	2	`DataWriterImpl, PublisherImpl`	739	**220**	133	74	13	0.1306	0.7023	24.3775
e_2	2	`DataReaderImpl, SubscriberImpl`	575	**177**	91	75	11	0.1050	0.6922	21.3
e_3	2	`TransportReceive[Listener|Strategy]`	207	**45**	17	26	2	0.0267	0.7826	1.7584
e_4	6	`TransportSend[Element|Listener], ...`	417	**118**	70	38	10	0.0700	0.7170	31.5369
e_5	3	`DataLink, DataLinkSet, ...`	349	**104**	51	47	6	0.0617	0.7020	31.6287
e_6	2	`SimpleTCP[DataLink|Transport]`	308	**85**	45	34	6	0.0504	0.7240	3.1309
e_7*	60	`BuiltinTopicUtils, Qos_Helper, ...`	2,456	**686**	373	280	33	0.4071	0.7207	67.8953
		total	5,892	1,685	880	712	93	1.0000	-	-

Column 3,4 Table **??**, report the experiment breakup by entity, i.e., the number of fault injected in the files which have been mapped to a specific entity and the number of experiments, which result in a failure outcome. This information allows us to estimate the $\#F_{j,ei}$ indexes as well as the overall FP_{e_i}. This is shown in the rightmost column.

Entities e_0 and e_1 exhibit the higher failure proneness at this stage of the analysis. More in details, e_0 is the main responsible in case of *no_msg* failures, while e_1 is the main responsible for *crash* and *value* failures. Entity e_2 is critical as well. Other considered entities, namely, e_3,e_4, e_5,e_6 do not exhibit a significantly high failure proneness when compared to the discussed entities. These modules are invoked by a DDS-based application to send and receive a message, respectively. We experience that faults injected in these modules are very likely to result in an application failure. We hypothesize that this behavior is the result of the lack of an inter-lying, i.e., between the DDS application and the `DataWriter(Reader)Impl`, module, which may mitigate, if not tolerate, a fault once triggered.

C. Criticality Assessment

Starting from

per brevita forse non si puo riportare gli indici distinti per failure mode – forse per uno piu rilevante dati su fault criticality da calcolare

VIII. Conclusion

Acknowledgment

The authors would like to thank...more thanks here

References

[1] E.J. Weyuker. Testing Component-Based Software: A Cautionary Tale, *IEEE Software*, Vol. 15, n. 5, pp. 54-59, 1998.

[2] R. L. O. Moraes, J. Durães, R. Barbosa, E. Martins, H. Madeira. Experimental Risk Assessment and Comparison Using Software Fault Injection, *Proc. of the 37th International Conference on Dependable Systems and Networks (DSN)*, pp. 512-521, 2007.

[3] D.P. Siewiorek, R. Chillarege, Z.T. Kalbarczyk. Reflections on Industry Trends and Experimental Research in Dependability, *IEEE Transactions on Dependable and Secure Computing*, pp. 109-127, 2004.

[4] A. Avižienis , J.C. Laprie, B. Randell, C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing, *IEEE Trans. on Dependable and Secure Computing*, pp. 11-33, 2004.

[5] J. Gray. Why do computers stop and what can be done about it, *Proc. of Symp. on Reliability in Distributed Software and Database Systems*, pp. 3-12, 1986.

[6] CENELEC: EN 50126 Railways Applications. The specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS).

[7] DO-178B/ED12B Software consideration in airborne systems and equipment certification. *RTCA and EUROCAE*, Dec. 1992.

[8] SAF.ET1.ST03.1000-MAN-01 Air Navigation System Safety Assessment Methodology (v2-0). *EUROCONTROL EATMP Safety Management*, Apr. 2004.

[9] T., Pasquale, E., Rosaria, M., Pietro, O., Antonio: Hazard analysis of complex distributed railway systems, in proc. of *the 22nd IEEE International Symposium on Reliable Distributed Systems (SRDS'03)*, pp. 283-292, Oct. 2003.

[10] P., Mana, J.M., De Redet, D., Fowler: Assurance Levels for ATM elements: Human (HAL), Operational Procedure (PAL), Software (SWAL), in proc. of *the 2nd IEEE Int. Conference on Institution of Engineering and Technology*, pp.13-19., Oct. 2007.

[11] C. Garrett and G. Apostolakis: Automated hazard analysis of digital control systems, in *Reliability Engineering and System Safety*, Vol. 77, pp. 1-17, 2002.

[12] C. Garrett, S. Guarro and G. Apostolakis: The Dynamic Flowgraph Methodology for Assessing the Dependability of Embedded Software Systems, in *IEEE Trans. on Syst., Man, and Cybern.*, Vol. 25, No. 5, pp. 824-840, May 1995.

[13] R. Hewett: Assessment of Software Risks with Model-Based Reasoning, in proc. of *IEEE Inter. Conf. on Systems, Man and Cybernetics*, vol. 4, pp. 3238-3243, Oct. 2005.

[14] S., Supakkul, C., Lawrence: Applying a Goal-Oriented Method for Hazard Analysis: A Case Study, in proc. of *the 4th International Conference on Software Engineering Research, Management and Applications (SERA'06)*, pp. 22- 30, Aug. 2006.

[15] N. Storey: Safety-Critical Computer Systems, Pearson and Prentice Hall, 1996.

[16] A.G., Hassami, A.G., Foord,: Systems safety-a real example (European rail traffic management system, ERTMS). in proc. of *the Second IEEE International Conference on Human Interfaces in Control Rooms, Cockpits and Command Centres*, pp. 327-334, 2001.

[17] A. Avizienis, L. Chen. On the implementation of N-version programming for software fault-tolerance during program execution, *Proc. of the COMPSAC 77*, Chicago, IL, pp. 149-155, 1977.

[18] A. Avizienis. The N-Version Approach to Fault - Tolerant Systems, *IEEE Transactions on Software Engineering*, vol. SE -11, no. 12, pp.1491-1501, Dec. 1985.

[19] B. Randell. Design - Fault Tolerance, in *The Evolution of Fault-Tolerant Computing*, A. Avizienis, H. Kopertz, and J.-C. Laprie, eds., Springer-Verlag, pp. 251-270, 1987.

[20] K.H. Kim, H.O. Welch. Distributed Execution of Recovery Blocks: An Approach for uniform Treatment of Hardware and Software Faults in Real- Time Applications, *IEEE Transactions on Computers*, vol.38, no. 5, pp. 626-636, May 1989.

[21] G.K. Saha. A Single-Version Scheme of Fault Tolerant Computing, *Journal of Computer Science & Technology*, ISTEC, Vol. 6 No. 1, 2006.

[22] T. Anderson, P.A. Barrett, D.N. Halliwell, M.R. Moulding. Software Fault Tolerance: An Evaluation, *IEEE Transaction on Software Engineering*, vol. SE-11 no. 12, Dec. 1995.

[23] Matt Bishop, Deb Frincke. Teaching Robust Programming, *IEEE Security and Privacy*, vol. 2, no. 2, pp. 54-57, Mar. 2004.

[24] Tsai, Wei-Tek Zhou, Xinyu Chen, Yinong Bai, Xiaoying. On Testing and Evaluating Service-Oriented Software, *Computer* , vol.41, no.8, pp.40-46, Aug. 2008.

[25] G. Canfora, M. Di Penta. Testing services and service-centric systems: Challenges and opportunities. *IT Professional* 8 , pp. 10-17, 2006.

[26] A. Bertolino, G. De Angelis, L. Frantzen, A. Polini. The PLASTIC Framework and Tools for Testing Service-Oriented Applications, *Proc. of the International International Summer Schools on Software Engineering - ISSSE 2006-2008*, number 5413 in Lecture Notes in Computer Science, pp. 106-139. Springer, 2009.

[27] C. Bartolini, A. Bertolino, S. Elbaum, E. Marchetti, Whitening SOA Testing, *Proc. of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 161-170, 2009.

[28] L .Baresi and E. DiNitto. Test and Analysis of Web Services, *1st ed., Springer*, 2007.

[29] G. Canfora and M. Di Penta. Service Oriented Architecture Testing : A Survey, *Number 5413 in LNCS*, pp. 78-105, Springer, 2009

[30] M. Zenha Rela, H.Madeira, J.G. Silva. Experimental Evaluation of the Fail-Silent Behaviour in Programs with Consistency Checks, *Proc. of the FTCS-26*, pp. 394-403, 1996.

[31] G. K. Saha. EMP- Fault Tolerant Computing: A New Approach, *International Journal of Microelectronic Systems Integration*, vol. .5, no. 3, Plenum Publishing Corp, USA, pp. 183-193, 1997.

[32] S. Yau, F. Chen. An Approach in Concurrent Control Flow Checking, *IEEE Transactions on Software Engineering*, vol. SE-6, no. 2, pp. 126-137, 1980.

[33] D.K. Pradhan. Fault-Tolerant Computer System Design, Prentice Hall, 1996.

[34] R. Chillarege, I.S. Bhandari, J.K. Chaar, M.J. Halliday, D.S. Moebus, B.K. Ray, M.-Y. Wong. Orthogonal Defect Classification-A Concept for In-Process Measurements. *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 943-956, Nov. 1992.

[35] J. Durães, H. Madeira. Emulation of Software Faults: A Field Data Study and a Practical Approach, *IEEE Transactions on Software Engineering*, Vol. 32, n. 11, pp. 849-867, 2006.

[36] K. Goseva-Popstojanova, A.P. Mathur, K.S. Trivedi. Comparison of architecture-based software reliability models. *Proc. of the 12th International Symposium on Software Reliability Engineering (ISSRE '01)*, pp. 22-31, 2001.

[37] GNU gprof, 1998. Available at `www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html`