UDC 004.05

P. POPOV

Centre for Software Reliability, City University, Northampton Square, London, EC1V 0HB

SOFTWARE FAULT-TOLERANCE WITH OFF-THE-SHELF COMPONENTS: FROM CONCEPTUAL MODELS TO EMPIRICAL STUDIES

Building software with off-the-shelf (OTS) components is an attractive alternative to bespoke development in terms of initial development cost. Dependability assurance of such software, however, appears more difficult than that of the bespoke alternative since it is rarely possible to use evidence about the development process of OTS components. Software development with OTS components is centered upon selecting the best components in the particular development context. If dependability itself or evidence about high dependability is insufficient, often the only available option for improvement is deploying fault tolerance based on design diversity. In such cases the importance of selecting the best components is even greater and more difficult, than for non-fault tolerant solutions: the best set of components is not necessarily the components which best on their own, how effective the fault-tolerant solution is depends on diversity of the chosen components. In the paper I critically review the conceptual models, which have been used for fault-tolerant software, their limitations, and some promising ways forward to help with the selection of OTS components for fault-tolerant software design. The approaches are illustrating with the results from very recent studies with OTS SQL servers.

software, Fault-tolerance, conceptual models

Introduction

Software fault tolerance based on design diversity has been studied very extensively, often surrounded by controversy, in the last 30 years [1]. This is probably the software engineering domain subjected to the most serious scrutiny.

The intuitive rationale behind the use of design diversity is simply the age-old human belief that "two heads are better than one". For example, we are more likely to trust our answer to some complex arithmetic calculation if a colleague has arrived independently at the same answer. In this regard, Charles Babbage was probably the first person to advocate using two computers – although by computer he meant a person [2].

Design diversity has been routinely used in safety critical applications.

Examples include the Airbus A320/30/40 aircraft [3 – 4] various railway signalling and control systems [5 – 9]. Well documented controlled experiments in mid 80s, [3, 7], also demonstrated that significant dependability gains can be achieved with design diversity. The adoption of diversity has been limited, though, by doubts about its costs and about its effectiveness. The attitudes to design diversity of industry and regulators vary, between industrial sectors but also within the same sector from cautious acceptance (e.g. [10] allow a company to claim diversity as one alternative to some other, standard assurance practices, but require the company to demonstrate the specific benefits claimed from its use of diversity) to an explicit view against design diversity (e.g. Boeing decided against software diversity for its own 777 aircraft, on the grounds that it would require restrictions to communication between software and system engineers, which in turn is an important defence against requirement errors [11]).

Conceptual Probabilistic Models of Design Diversity

A recent survey of the probabilistic models of fault-tolerant software can be found in [10]. An extension of the previous models, discussed in the survey, to take account of various regimes of testing was developed in [12]. These are briefly summarized here.

The first major breakthrough in stochastic modelling of diversity came in a paper by Eckhardt and Lee [4]. The key idea (EL model) was that the different demands that a program might receive during operation would vary in 'difficulty' – specifically the probability of failure upon execution of a demand would be different for different demands.

For a particular set of requirements there is a population of all possible programs (versions), which (conceptually, at least) could be written, $\wp = \{\pi_1, \pi_2, \pi_3, ...\}$.

Many, if not most, of these programs will be incorrect, i.e. they sometimes give wrong output.

An actual product development is then modelled as a random selection of π from $\wp$, i.e., the program is a random variable Π, with $P(\Pi = \pi) = S(\pi)$, for some measure $S(\bullet)$ over $\wp$. The measure $S(\bullet)$ can be thought of as representing the *development methodology* used.

Execution of a program version involves random selection of a demand from the demand space $F = \{x_1, x_2, ...\}$.

That is, the demand is a random variable X with $P(X=x) = Q(x)$ for some measure $Q(\bullet)$ over F.

Here $Q(\bullet)$ could be thought of as the usage distribution over demands. It might vary from one user environment to another.

The failure behaviour of the program is described by the score function

$$\upsilon(\pi, x) = \begin{cases} 1, \textit{ if program } \pi \textit{ fails on } x; \\ 0, \textit{ if program } \pi \textit{ does not fail on } x. \end{cases}$$

Thus, the random variable $\upsilon(\Pi, X)$ represents the performance of a random program on a random demand: this is a model for the uncertainty both in software *development* and *usage*.

A key average performance measure is

$$\theta(x) = \sum_{\wp} \upsilon(\pi, x).S(\pi) = \mathrm{E}_S\big(\upsilon(\Pi, x)\big), \qquad (1)$$

which is the probability that a randomly chosen program fails for a particular demand x. The heart of Eckhardt and Lee's idea is the recognition that $\theta(x)$ will generally take different values for different x, representing the varying 'difficulty' in correctly processing different demands. For a randomly chosen demand X, $\theta(X)$ is a random variable.

Finally,

$$\begin{gathered} \mathrm{E}(\theta(X)) = \\ = \sum_{\wp} \sum_{F} \upsilon(\pi, x).S(\pi).Q(x) = \\ = \mathrm{E}_{S,Q}\big(\upsilon(\Pi, X)\big) \end{gathered} \qquad (2)$$

represents the probability that a randomly chosen program fails on a randomly chosen demand.

In the presence of uncertainties of both development and usage, this represents the likelihood that our software fails.

Suppose now that two program versions are created independently. That is, Π_1 and Π_2 are selected independently from $\wp$.

The independent selection models the process of development of the programs by *independent teams*, e.g. not communicating with each other. These are *truly independent* in the conventional statistical sense:

$$\begin{gathered} P(\Pi_1 = \pi_1, \Pi_2 = \pi_2) = \\ = P(\Pi_1 = \pi_1).P(\Pi_2 = \pi_2). \end{gathered} \qquad (3)$$

It then follows that the probability that both Π_1 and Π_2 fail on a given demand x is:

$$\begin{gathered} P(\Pi_1 \textit{ fails on } x, \Pi_2 \textit{ fails on } x) = \\ \sum_{\wp} \sum_{\wp} \upsilon(\pi_1, x).\upsilon(\pi_2, x).P(\Pi_1 = \pi_1).P(\Pi_2 = \pi_2) \\ = P(\Pi_1 \textit{ fails on } x).P(\Pi_2 \textit{ fails on } x) = \big(\theta(x)\big)^2. \end{gathered} \qquad (4)$$

The conditional form of the joint behaviour is:

$$\begin{gathered} P(\Pi_2 \textit{ fails on } x \,|\, \Pi_1 \textit{ failed on } x) = \\ \frac{P(\Pi_1 \textit{ failed on } x, \Pi_2 \textit{ fails on } x)}{P(\Pi_1 \textit{ failed on } x)} = \\ P(\Pi_2 \textit{ fails on } x) = \theta(x). \end{gathered} \qquad (5)$$

Thus, one can see that independently developed programs fail independently when executing a given fixed demand x.

However the situation is different when there is uncertainty concerning the demand, i.e., the programs execute a *random* demand, X:

$$P(\Pi_1 \text{ and } \Pi_2 \text{ both fail on } X)$$
$$P(\upsilon(\Pi_1, X)\upsilon(\Pi_2, X) = 1) =$$
$$\sum_F \sum_{\wp} \sum_{\wp} \upsilon(\pi_1, x)\upsilon(\pi_2, x)S(\pi_1)S(\pi_2)Q(x) = \quad (6)$$
$$\sum_F (\theta(x))^2 Q(x) = \mathrm{E}(\Theta^2) = Var(\Theta) + (\mathrm{E}(\Theta))^2 =$$
$$Var(\Theta) + (P(\Pi \text{ fails on } X))^2.$$

(here the random variable $\Theta = \theta(X)$). The conditional form of the joint behaviour is:

$$P(\Pi_2 \text{ fails on } X \mid \Pi_1 \text{ failed on } X) =$$
$$\frac{Var(\Theta)}{E(\Theta)} + P(\Pi_2 \text{ fails on } X) \geq P(\Pi_2 \text{ fails on } X). \quad (7)$$

Equality holds if and only if $\theta(x) = \theta$ identically for all x and it seems likely that this will never be the case. This is the main result of [4]: that the failure behaviour of diverse versions will necessarily be *worse* than what could be expected under the assumption of independence, *even though the versions themselves truly are 'independently developed'*.

Littlewood and Miller (LM model) extended this model, [8], to the case where several development methodologies A,B,C, etc. are available. These might represent, for example, different development environment, different types of programmers, different languages, different testing regimes, etc. Each methodology induces a measure on $\wp$, the set of all possible program versions. A random program Π_A developed using methodology A will be version π with probability:

$$P(\Pi_A = \pi) = S_A(\pi).$$

If the methodologies are very diverse, we would expect a program with a high probability of selection under one methodology to have a low, perhaps zero, probability of selection under others. Within a particular methodology, the situation is exactly like that in [4]. Thus $\theta_A(x)$ is the probability of a randomly chosen program from methodology A failing on demand x; the random variable $\Theta_A = \theta_A(X)$ is the probability of Π_A failing on the random demand X, etc.

Consider two random program versions Π_A and Π_B developed independently under methodologies A and B respectively, i.e.

$$P(\Pi_A = \pi_A, \Pi_B = \pi_B) =$$
$$P(\Pi_A = \pi_A)P(\Pi_B = \pi_B) = S_A(\pi_A)S_B(\pi_B). \quad (8)$$

It follows that, for any given demand, x, two randomly chosen programs fail independently, as in (5) while the probability of simultaneous failure on a *random* X is:

$$P(\Pi_A \text{ fails on } X, \Pi_B \text{ fails on } X) = \mathrm{E}(\Theta_A\Theta_B) =$$
$$Cov(\Theta_A, \Theta_B) + \mathrm{E}(\Theta_A)E(\Theta_B) = \quad (9)$$
$$Cov(\Theta_A, \Theta_B) + P(\Pi_A \text{ fails on } X)P(\Pi_B \text{ fails on } X).$$

The conditional form of joint behaviour is:

$$P(\Pi_B \text{ fails on } X \mid \Pi_A \text{ failed on } X)$$
$$= \frac{\mathrm{E}(\Theta_A\Theta_B)}{\mathrm{E}(\Theta_A)} = \frac{Cov(\Theta_A, \Theta_B)}{\mathrm{E}(\Theta_A)} + \mathrm{E}(\Theta_B). \quad (10)$$

This is greater than $\mathrm{E}(\Theta_B) = P(\Pi_B \text{ fails on } X)$ if and only if $\mathrm{Cov}(\Theta_A, \Theta_B) > 0$. But since it is possible that $Cov(\Theta_A, \Theta_B) < 0$, it follows that using different design methodologies it is possible in this model to do even better than the (unattainable) goal of independent performance of versions in the single methodology case. This is the main result in [8].

These two models were recently generalized [12] by taking into account the regime of testing applied to both channels. It turns out that after subjecting the channels to testing the EL and LM models will only apply if the channels are tested independently (i.e. independent development of the channels extends to testing, too). If the channels are subjected to testing together, however, e.g. on the same testing suite or even back-to-back, then even conditional failure independence (with respect to a given demand, x, that is) does not hold. Common testing introduces dependence between the channel failures, which generally would make less reliable than under the EL and LM models.

Models 'on average' vs. models 'in particular'

One will have noticed that the models described so far express the probability of simultaneous failures of two *randomly* selected channels from the population of available versions (hypothetically possible to develop using a specific design methodology to a given specification). These models are very useful to study the limitations of design diversity at a high level of abstraction. A typical question that these models allow one to answer is 'Is it reasonable to expect that if I recruit two teams to develop the channels of a fault-tolerant system and keep them isolated form each other the failures of the channels are likely to be stochastically independent?'. This is not a naïve question. It has far reaching implications on how I would assess the system once the channels are developed. If I knew that independence is likely, then I would concentrate on assessing the channels' probability of failure and once this is done, I would compute the probability of failure of the system by multiplying these. The EL and LM models explain why, unfortunately, such an assessment procedure is unsound: independence of failures is unlikely and one should plan for assessment of the system without reference to failure independence. The conclusion that is drawn, however, is about the populations of *possible versions* (and the respective probabilistic measures defined for them). The probability of coincident failure that the EL and LM models allow us to compute, (7) and (10), is thus a measure 'on average'.

Once the channels have been developed (i.e. the selection from the respective populations has taken place) one is interested in a different question: 'How good is the *particular pair* of channels that I am dealing with'. For this reason the EL and LM models and any extensions thereof are useless. Indeed, the versions in the populations from which one chooses the pair may vary (and significantly so) in their reliability, hence the reliability of the pair will vary, too. For instance, in the controlled experiment reported in the mid 80s by Knight and Leveson [7] 6 out of the 27 versions developed in the experiment did not fail in 1,000,000 tests used to asses the version reliability. Clearly, any of these 6 versions will make a 1-out-of-2 system perfect. The problem of assessing the available pair, thus, will be very different from the conceptual question whether independent development is likely to deliver "on average" independently failing software versions. Assessing how reliable the particular pair of versions is requires models 'in particular'. One such model was developed a few years back [14], [13]. The model can be derived from the model "on average" by eliminating the uncertainty associated with the selection of the versions. The only uncertainty to be accounted for is that associated with the usage, $Q(x)$[1].

While mathematically the models "on average" and "in particular" are very similar, the difference between them in practical terms is very significant. While for the models 'on average' there is *no hope* that the parameters of interest (various measures computed on the entire populations of versions) can be estimated, this problem seems tractable in the models in particular.

The particular approach taken in the above mentioned papers is using the idea of partitioning the demand space, an approach used widely to achieve effective software testing. In each partition one can estimate the probability of failure of the channels, and thus reduce significantly the uncertainty about the probability of system failure over the entire demand space.

Consider the *pfd* of a pair of versions, A and B, which are *completely known*: for each demand, one knows the score functions of the channels, A and B, $\omega_A(x) \equiv \upsilon(A,x)$ and $\omega_B(x) \equiv \upsilon(B,x)$. Probabilities of failure of versions A and B on a randomly selected demand X (probability of failure per execution) are:

$$P_A \equiv P(A \text{ fails on } X) = E(\omega_A(X)) = \sum_{x \in F} Q(x)\omega_A(x)$$

[1] One can think of this model 'in particular' as a special case of the models 'on average' in which the size of the population of versions from which one selects the channels is reduced to just a single version.

and

$$P_B \equiv P(\text{B } \textit{fails on } X) = \mathrm{E}(\omega_B(X)) = \sum_{x \in F} Q(x)\omega_B(x),$$

where F denotes the demand space, and $Q(x)$ is the probability of demand x (the demand profile of the software). For a specific demand x, the probability of common failure is then either 0 or 1:

$$P(A \text{ fails on } x \text{ and } B \text{ fails on } x) = \omega_A(x)\omega_B(x)$$

pfd_{AB}=P(A and B fail on randomly chosen demand) =

$$\sum_{x \in D} Q(x)\omega_A(x)\omega_B(x). \qquad (11)$$

It turns out that this expression can be written as:

$$pfd_{AB} = P_A P_B + \mathrm{cov}(\Omega_A, \Omega_B), \qquad (12)$$

where the random variables Ω_A and Ω_B are defined as the values taken by ω_A and ω_B on a randomly chosen demand.

Equation (12) is identical to (9). The only difference between the two is that (9) is based on the 'difficulty functions' for two "development methodologies", which can take any value between 0 and 1 (representing the probability that a randomly chosen version, developed with that methodology, would fail on a given demand). (12), instead, operates with two known versions. The functions $\omega_A(x)$ and $\omega_B(x)$ can only take the values 0 and 1, and the only uncertainty concerns the choice of the next demand, x, described by the probability distribution $Q(x)$. The description given this far would only be useful if one knew the behaviour of each version on each possible demand, i.e., for each demand whether it is a failure point or a success point, for each version. This level of detailed knowledge is normally *unattainable*. The knowledge that can be obtained is at a much coarser level: by realistic testing, one can estimate the likelihood of each version failing on a randomly chosen demand. We can also specialise this knowledge slightly, by testing separately for separate classes of demands that completely cover the demand space, without any overlapping between them (a *partition* over the demand space) [2]. We call the subdomains themselves S_1, S_2, …, S_n. We can define the probability of failure of a version when subjected only to demands from a specific subdomain, e.g. $P(A|S_i)$ will designate the probability that A fails on a demand chosen randomly from subdomain S_i, according to the probability distribution of demands in actual operation. We can then write the probability of common failure as:

$$pfd_{AB} = P(A,B) = \sum_i P(A,B \mid S_i)P(S_i). \qquad (13)$$

Within each subdomain, clearly:

$$P(A,B \mid S_i) \neq P(A \mid S_i)\, P(B \mid S_i). \qquad (14)$$

Equality would only apply in special cases, e.g. hardware-only versions that are subject only to physical failures and for which the stress to which they are subject is known to be constant across a certain class of demands. In most cases, one would expect a restricted class of demands to pose similar problems to the designers of two versions, so that the EL model would apply: in each subdomain, the left-hand term in (14) would be greater than the right-hand term. So, in practice, a regulator can use the sum in the left-hand side of the following expression:

$$\sum_i P(A \mid S_i)P(B \mid S_i)P(S_i) \leq \sum_i P(A,B \mid S_i)P(S_i) = pfd_{AB}, \qquad (15)$$

as a likely *claim limit* for the *pfd* of a two-version system.

Even if formula (14) could be written with an equal sign (independent failures of the two versions, *conditional* on demands from a given subdomain) for all subdomains, this would not imply unconditional independence of failure. In terms of reliability estimates over subdomains, pfd_{AB} can be written, in a general form, as:

[2] Subdividing demands into classes is common practice for designers (e.g., in terms of *modes* of operation of a system) and software testers, who call these classes *sub-domains* (in the demand space). For instance, testers find it useful to define sub-domains on the basis of which 'function' of the program (as defined in its requirements) the demands invoke, or on the basis of which parts of the code they cause to be executed.

$$pfd_{AB}=P(A)\,P(B)+\mathrm{cov}_1+cov_2, \qquad (16)$$

where the term cov_1 is obtained by considering the *pfd* values of the two versions as functions of the subdomains, and taking their covariance over all the subdomains,

$$\mathrm{cov}_1=\sum_i\big(P(A|S_i)-P(A)\big)\big(P(B|S_i)-P(B)\big)P(S_i) \qquad (17)$$

and the term cov_2 is obtained by computing the covariance of the Ω functions of the two versions in each subdomain, and taking its average over all subdomains:

$$\mathrm{cov}_2=\sum_i\big(\mathrm{cov}(\Omega_1,\Omega_2|S_i)\big)P(S_i)\,. \qquad (18)$$

Each term in the inner sum above represents the difference between the two sides of inequality (14). Assuming each such term to be 0, i.e., conditional independence within each subdomain, makes *cov*2 equal to zero.

Bayesian Assessment

How do these models help one with the selection of OTS software to be deployed in a fault-tolerant configuration?

The models "on average" allows one to structure one's argument – failure independence is unlikely to occur. If one decides to argue in favor of failure independence then one should build a very strong argument. The models "in particular", indicate that unless full knowledge about the channels is attained, there will be uncertainty regarding the probability of coincident failure due to the term cov_2, the covariance between the channels in the respective sub-domains. The implications, for justifying having achieved a particular reliability target with the fault-tolerant software is that this uncertainty *should be managed*, ideally bounded. The magnitude of the uncertainty will vary between 'know nothing' (i.e. the probability of simultaneous failure may be anywhere between 0 and the probability of failure of the more reliable of the two channels in the particular sub-domain, i.e. diversity buys me nothing in terms of dependability improvement) or some quantification, e.g. based on the empirical evidence that will be accumulated over the life-cycle of the fault-tolerant system.

In the latter case, when new evidence is expected to emerge after the deployment of the system, Bayesian assessment techniques [9] seem particularly well suited – they allow in a mathematically sound way to combine the *a priori* knowledge (possibly inaccurate, possibly largely based on expert judgment) with new empirical evidence as it emerges during the life-time of the system to produce a more refined *a posteriori* view on system dependability. The conceptual models described above will guide the assessor about choosing an adequate Bayesian model, i.e. such that would allow the 'data to speak for itself', i.e. allow the posterior to capture the true dependence (positive or negative correlation between the failures of the channels). Choosing an inadequate model, i.e. based on various unreasonable assumptions (e.g. of independence or some parameters being known with certainty) will prevent the assessor from learning about the system behaviour, a recipe of being unpleasantly surprised. An example of an inadequate model that I argued against is given in [11]. The mistake criticized was assuming that the components used to build a software system fail independently. There is nothing wrong with the assumption in principle: provided the components do indeed fail independently, assuming independence will not be refuted by empirical. If independence is assumed merely for convenience, i.e. to simplify the calculations, however, and evidence is collected against the assumption, then predictions obtained with the model (i.e. the posterior distribution of the probability of system failure) may lead to uncontrollable error: the prediction may be pessimistic (which may be acceptable, e.g. in safety-critical context although may be expensive – the conclusion that the system is good enough may be prolonged merely due to using inadequate model), but may also be optimistic, which might be dangerous, as illustrated in [11]. In summary, the models described in the previous sec-

tions will allow one to structure the argument avoiding unreasonable simplifications.

Off-the-shelf software

While employing software diversity was seen in the past as an expensive method for increasing dependability due to the need of *building* more than one component. With off-the-shelf components this problem is overcome: there may be many different components that will have the required functionality, therefore bespoke development may not be required[3]. Moreover many of these components are free and open-source, thus the cost of procurement may be non-existent.

There exist a plethora of available methods for COTS assessment, mainly developed for managers in the context of non-fault-tolerant solutions.

The problem of assessment though still remains. If we were interested in building a 1-out-of-2 system, simply choosing the two best components that exist in the market may not be enough. What is of interest is how well the pair works together. The optimal pair will be the one with the lowest probability of simultaneous failures of both components of the pair. The components that form the best pair may not necessarily be the ones, which are the best individually. As the conceptual models described above show the probability of simultaneous failure may be reduced by improving the reliability of the components or by choosing those with low, possibly negative correlation between their failures.

But how can one choose the best pair before the OTS components are integrated together?

Ideally the decision should be based on empirical evidence about the products, although very often 'soft' aspects (e.g. good relationship with the vendors, vendor's credentials, market position, etc.) are the main factors that dictate which components get chosen. Putting this aspect aside, methods are needed to justify the selection so that the pair is likely to deliver the highest possible dependability. In this section a summary is presented of a *recently developed method*, which allows for rigorous selection procedure, based on empirical data that is likely to be available for a wide range of OTS software components/products. The method [5] based on Bayesian inference, allows one to use expert judgment (in defining the priors) and empirical evidence about the products considered for selection, such as their bug records.

The method was applied to a set of bugs of several off-the-shelf database servers [6] and used these as a sample from the *stressful environments*, defined by all bugs in the servers. The selection of the servers will be optimal for the so defined stressful environment, a reasonable option for the particular system in mind – a fault-tolerant SQL server. After all the fault-tolerant solution with a pair of servers is intended to cope with the difficult situations (demands) where the individual channels might be deficient.

Table 1

The observations for the 6 diverse server pairs on the bug reports of the different partitions. In the partition column the subscript indicates for which server these bugs have been reported. N is the total number of bugs run. r_1, r_2 and r_3 represent the counts of bugs which caused failure of only the first server, of only the second server, and of both servers of the pair, respectively

Server Pair	Partition	N	r_1	r_2	r_3	Server Pair	Partition	N	r_1	r_2	r_3
PG & IB	S_{PG}	24	21	0	0	IB & OR	S_{PG}	18	0	0	0
	S_{IB}	28	0	23	1		S_{IB}	31	25	0	0
	S_{OR}	3	0	0	0		S_{OR}	4	0	3	0
	S_{MS}	9	0	0	0		S_{MS}	10	1	0	0
PG & OR	S_{PG}	30	27	0	0	IB & MS	S_{PG}	21	0	1	0
	S_{IB}	24	1	0	0		S_{IB}	35	27	0	2
	S_{OR}	4	0	2	1		S_{OR}	4	0	0	0
	S_{MS}	7	0	0	0		S_{MS}	12	0	6	1
PG & MS	S_{PG}	33	28	0	2	OR & MS	S_{PG}	27	0	2	0
	S_{IB}	25	1	2	0		S_{IB}	30	0	2	0
	S_{OR}	3	0	0	0		S_{OR}	4	3	0	0
	S_{MS}	18	1	7	5		S_{MS}	12	0	7	0

[3] Apart from *'glue code'* (usually referred to as *middleware)* which may be needed to ensure the components can be deployed for a given system in a coordinated manner as required by the particular system context.

The bugs used in the study have been collected for four SQL servers [6], namely PostgreSQL 7.0, Interbase 6.0, Oracle 8.0.5 and Microsoft SQL server 7 (for the sake of brevity referred to as PG, IB, OR and MS). The union of the bugs for all the compared COTS products forms a stressful demand space, in which there is a partition stressing each of the products. The logs of the known bugs are treated as a *sample* (without replacement) from the corresponding partition (representing the server, for which the bug has been reported). The partitions are named $S_{Server\ name}$. Partition S_X is called an 'own' partition for server X and a 'foreign' partition for any other server $Y \neq X$. The data collected from the bug logs is summarized in Table 1.

The servers are then compared using a Bayesian inference procedure (using the same prior for all servers/pairs, justified for each of the partitions).

The results of the comparison are summarized in Table 2.

Table 2

The percentiles of the probability of system failure for each server pair

Server Pair	50th percentile		99th percentile	
	Prior	Posterior	Prior	Posterior
PG & IB	0,3	0,02	0.61	0,12
PG & OR		0,07		0,19
PG & MS		0,09		0,20
IB & OR		0,02		0,14
IB & MS		0,04		0,14
OR & MS		0,02		0,14

Quite unexpectedly, it turned out that the best pair is the pair formed of two open source products, Interbase and PostgreSQL (highlighted in the 99th percentile column). They scored better than the commercial servers. As a validation procedure to assess whether the method is trustworthy, the same method was used to compare the servers in terms of two non-functional attributes – dependability and performance (measured in terms of response time on a standard performance benchmark for on-line transaction processing for databases, TPC-C). The study revealed that Oracle is individually the best server, which is in line with the common view that Oracle is the best SQL server. We concluded that the method, despite its limitations, may be useful for optimal selection.

Conclusion

This paper surveys a range of topics related to building fault-tolerant software with off-the-shelf components. The current state-of-art in probabilistic modeling of fault-tolerant software has provided evidence, both theoretical and empirical, against assuming that software diversity is likely to deliver failure independence "on average". Empirical studies have shown that software development process cannot deliver products with *consistent reliability*: the reliability of a particular single version and fault-tolerant software, thus, can vary greatly. The probabilistic models "in particular" show the nature of uncertainty and a way of managing it in the context of Bayesian assessment, which allows one to combine expert judgment with empirical evidence. Finally, we illustrated how Bayesian assessment techniques can be used to make an optimal selection of OTS products for building fault-tolerant software. The usefulness of the method is demonstrated with complex products such as off-the-shelf SQL servers.

References

1. Avižienis A., Laprie J.-C., Randell B., Landwehr C. Basic Concpets and Taxonomy of Dependable and Secure Computing // IEEE Transactions on Dependable and Secure Computing. – 2004. – 1. – P. 11-33.

2. Babbage C. On the Mathematical Powers of the Calculating Engine (Unpublished manuscript, December 1837) // The Origins of Digital Computers: Selected Papers. Randell, B., Springer. – 1974. – P. 17-52.

3. Eckhardt D.E., Caglayan A.K., Knight J.C., Lee L.D., McAllister D.F., Vouk M.A., Kelly J.P.J. An experimental evaluation of software redundancy as a strategy for improving reliability // IEEE Transactions

on Software Engineering. – 1991. – 17. – P. 692-702.

4. Eckhardt D.E., Lee L.D. A theoretical basis for the analysis of multiversion software subject to coincident errors // IEEE Transactions on Software Engineering. – 1985. – SE-11. – P. 1511-1517.

5. Gashi I., Popov P. Uncertainty Explicit Assessment of Off-the-Shelf Software: Selection of an Optimal Diverse Pair. 6th International IEEE Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (ICCBSS'07) Banff, Alberta, Canada // IEEE Computer Society Press. – 2007. – P. 93-102.

6. Gashi I., Popov P., Strigini L. Fault diversity among off-the-shelf SQL database servers. Dependable Systems and Networks (DSN'04), Florence, Italy // IEEE Computer Society Press. – 2004. – P. 389-398.

7. Knight, J.C., Leveson, N.G. An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming // IEEE Transactions on Software Engineering. – 1986. – SE-12. – P. 96-109.

8. Littlewood B., Miller D.R. Conceptual Modelling of Coincident Failures in Multi-Version Software // IEEE Transactions on Software Engineering, IEEE. – 1989. – SE-15. – P. 1596-1614.

9. Littlewood B., Popov P., Strigini L. Assessment of the Reliability of Fault-Tolerant Software: a Bayesian Approach // 19th International Conference on Computer Safety, Reliability and Security, SAFECOMP'2000, Rotterdam, the Netherlands, Springer. – 2000.

10. Littlewood B., Popov P., Strigini L. Modelling software design diversity - a review. // ACM Computing Surveys. – 2001. – 33. – P. 177-208.

11. Popov P. Reliability Assessment of Legacy Safety-Critical Systems Upgraded with Off-the-Shelf Components // SAFECOMP'2002, Catania, Italy, Springer-Verlag. – 2002. – P. 139-150.

12. Popov P., Littlewood B. The Effect of Testing on Reliability of Fault-Tolerant Software. Dependable Systems and Networks (DSN'04), Florence, Italy // IEEE Computer Society Press. – 2004. – P. 265-274.

13. Popov P., Strigini L., May J., Kuball S. Estimating Bounds on the Reliability of Diverse Systems. IEEE Transactions on Software Engineering // IEEE Computer Press. – 2003. – 29. – P. 345-359.

14. Popov P.T., Strigini L. Conceptual models for the reliability of diverse systems - new results // 28th International Symposium on Fault-Tolerant Computing (FTCS-28), Munich, Germany. – IEEE Computer Society Press. – 1998. – P. 80-89.

Поступила в редакцию 19.02.2007

Рецензент: д-р техн. наук, проф. В.С. Харченко, Национальный аэрокосмический университет им. Н.Е. Жуковского «ХАИ», Харьков.