View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Springer - Publisher Connector

Nascimento et al. *Journal of Software Engineering Research and Development* 2014, **2**:13
www.jserd.com/content/2/1/13

Journal of Software Engineering Research and Development
a SpringerOpen Journal

RESEARCH **Open Access**

Designing fault-tolerant SOA based on design diversity

Amanda S Nascimento[1*], Cecília MF Rubira[2], Rachel Burrows[3], Fernando Castor[4] and Patrick HS Brito[2,5]

*Correspondence:
anascimento@iceb.ufop.br
[1]Institute of Exact Sciences and Biology, Federal University of Ouro Preto, Ouro Preto, MG, Brazil
Full list of author information is available at the end of the article

Abstract

Background: Over recent years, software developers have been evaluating the benefits of both Service-Oriented Architecture (SOA) and software fault tolerance techniques based on design diversity. This is achieved by creating fault-tolerant composite services that leverage functionally-equivalent services. Three major design issues need to be considered while building software fault-tolerant architectures based on design diversity: (i) selection of variants; (ii) selection of an adjudication algorithm to choose one of the results; and (iii) execution of variants. In addition, applications based on SOA need to function effectively in a dynamic environment where it is necessary to postpone decisions until runtime. In this scenario, control is highly distributed and involves conflicting user requirements. We aim to support the software architect in the design of fault-tolerant compositions.

Methods: Leveraging a taxonomy for fault-tolerant systems, this paper proposes guidelines to aid software architects in making key design decisions. The taxonomy is used as the basis for defining a set of guidelines to support the architect in making decisions related to fault tolerance in SOA. The same taxonomy is used in a systematic literature review of solutions for fault-tolerant composite services.
The review investigates how existing approaches for fault-tolerant composite services address design diversity issues and also specific issues related to SOA.

Results: The contribution of this work is twofold: (i) a set of guidelines for supporting the design of fault-tolerant SOA, based on a taxonomy for fault tolerance techniques; and (ii) a systematic literature review of existing solutions for designing fault-tolerant compositions using design diversity.

Conclusion: Although existing solutions have made significant contributions to the development of fault-tolerant SOAs, there is a lack of approaches for fault-tolerant service composition that support strategies with diverse quality requirements and encompassing sophisticated context-aware capabilities. This paper discusses which design issues have been addressed by existing diversity-based approaches for fault-tolerant composite services. Finally, practical issues and difficulties are summarized and directions for future work are suggested.

Keywords: Software fault tolerance; Design diversity; Service-oriented architecture; Systematic literature review; Fault-tolerant service composition

Springer

© 2014 Nascimento et al.; licensee Springer. This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/2.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly credited.

1 Introduction

Nowadays, society is highly dependent on systems utilizing Service-Oriented Architectures (SOA) for its basic day-to-day functioning (Huhns and Singh 2005; Papazoglou et al. 2007). These systems range from online stores to complex applications, called mashups, that combine their own resources with content retrieved via services from external data sources to create new functionalities (Huhns and Singh 2005; Papazoglou et al. 2007; Zheng and Lyu 2010b). Nevertheless, it is unlikely that services (often controlled by third parties) will ever be completely free of software faults arising from wrong specifications or incorrect coding (Trivedi et al. 2010). Consequently, SOA-based applications should operate according to their specification in spite of faults from reused services. If faults are not tolerated then undesirable consequences could happen, which may range from mildly annoying to great financial losses (Nascimento et al. 2011; Papazoglou et al. 2007; Zheng and Lyu 2010b).

The adoption of software fault tolerance techniques based on design diversity has been advocated as a means of coping with residual software design faults in operational software (Lee and Anderson 1990). Design diversity is the provision of software components called variants, which have the same or an equivalent specification but with different designs and implementations (Gärtner 1999). An assumption of software fault tolerance techniques is that the probability of having the same fault in multiple variant components is lower, meaning that a fault present in a component should be detected and tolerated based on the behaviour of other variants (Lyu 1996). In the nineties, the use of techniques based on design diversity to tolerate software faults was widely criticised since variant software components used to be developed from scratch, which is very expensive (Anderson et al. 1985; Vouk et al. 1993). Therefore these techniques were generally used only in highly critical systems, in which the occurrence of failures would result in large financial losses or even loss of life. Nevertheless, in the context of SOA on the web, there are already many services that provide equivalent functionality, thus making such techniques more practical (Zheng and Lyu 2010b). These variant services might be simply cost-free and open access, or even offered by external organizations to cope with changes to user quality of services (QoS) requirements (Papazoglou et al. 2007). Due to the low cost of reusing existing variant services, several diversity-based approaches have been developed to support reliable SOA-based applications. These approaches operate as mediators between clients and variant services. The latter are structured in fault-tolerant composite web services (Nascimento et al. 2011; Zheng and Lyu 2010b). Hereafter we refer to fault-tolerant composite web services as FT-compositions. From the clients' viewpoint, an FT-composition works as a single, reliable service.

In order to design reliable SOA applications, important design decisions have to be made by the architect. Such decisions are difficult, especially in the context of mashups, since the architect has to consider many aspects related to both fault tolerance and SOA-specific quality requirements. Regarding fault tolerance, the architect has to consider, for example: the availability of variants, the best fault tolerance technique to be used in a certain context, how the system should fail and scenarios involving error detection and handling (failure modes), the categories of faults to be tolerated (fault latency), assumptions about the environment and components (fault assumptions), etc.

The software architect developing a fault-tolerant SOA application should consider three major design issues when using design diversity: (i) selection of variants, since the

variants need to be sufficiently diverse and able to tolerate software faults; (ii) selection of an adjudicator to determine the acceptability of the results obtained from the variants (Daniels et al. 1997; Lee and Anderson 1990); and (iii) execution strategy that directs the execution of the variants. For each of these design issues, the architect has to choose the specific fault tolerance technique to be used. Guidelines to support this task should also take into account implementations that realize different sets of quality requirements (e.g. memory consumption, financial cost, response time or reliability). The design decisions are also affected by characteristics of the functionality (e.g., allows re-try, allows undo operation). Variants can be executed either sequentially or in parallel and variant outputs can be adjudicated by adopting different voting and acceptance algorithms (Daniels et al. 1997; Laprie et al. 1990).

When designing a system, the architect may reuse and adapt existing solutions. Nevertheless, existing work regarding fault-tolerant service compositions is written from different viewpoints and relies on different technical backgrounds. As a result, it is hard to compare them and to choose an appropriate solution to be applied. Thus, it is unclear the extent to which existing solutions support the above mentioned design issues related to software fault tolerance based on design diversity. In order to avoid neglecting important design issues related to fault tolerance, architects should use guidelines to support the identification of which design issues and respective solutions should be used depending on the application's requirements.

In this sense, the contribution of this work is twofold: (i) the proposal of guidelines for supporting the design of fault-tolerant SOA based on a taxonomy for fault tolerance techniques and (ii) a systematic review of existing solutions for designing FT-compositions using design diversity. Results from reviewed solutions are presented to support reuse of existing solutions. The central purpose, which unifies the two contributions, is to support the architect in the design of fault-tolerant compositions. The proposed guidelines utilize an existing taxonomy for fault-tolerant systems (Pullum 2001) in order to guide architects in their design decisions. Then, the systematic literature review classifies existing solutions according to the same taxonomy, thus providing a basis for comparison and analysis of the solutions according to the architect's specific needs.

Guidelines support different decisions regarding the design of FT-SOA: first, the guidelines address different failure modes. This enables the architect to plan which faults are to be tolerated. Subsequently, the guidelines support the architect in modelling the faults and the behavioural pattern of how these faults should be tolerated. This decision influences the choice of the adjudicator type. For example, intermittent faults can be detected and tolerated utilizing design diversity and majority election design techniques. Also, the number of variants will be directly affected by the number of concurrent failures to be tolerated. The architect can assess the number of variants that are required to achieve the desired level of reliability. This design decision is also affected by the availability of resources as this will place limits on the number of variants that are feasible.

The systematic review compares characteristics of existing solutions in order to support architectural-level decisions regarding software fault tolerance and SOA quality requirements. We followed the literature review method proposed by Kitchenham (2007). We first investigated design issues related to the selection and the execution of variants, as well as the adjudication of their outputs. Secondly, we investigated which SOA-specific requirements were addressed by the existing solutions. The proposed guidelines address

the three major design issues and their different implementation solutions. Finally, we report our main findings and identify gaps in current approaches in order to suggest opportunities for research on reliable SOA-based applications.

The remainder of the paper is structured as follows. Section 2 presents important concepts related to SOA and software fault-tolerance. These concepts are utilized in the taxonomy for fault-tolerant SOA. Section 3 presents the taxonomy combining elements from the fault-tolerance domain with elements from the SOA domain. Section 4 presents the results and discussion. This includes the guidelines for supporting the SOA architect in designing FT-compositions. It also presents results from the systematic literature review.

Section 5 discusses the threats and the validity of the review. Section 6 presents related work, which also considers related literature reviews of software fault tolerance techniques. Finally, Section 7 presents some concluding remarks and directions for continuing work.

2 Background

2.1 Service-Oriented Architecture (SOA)

Many software systems are being implemented following the Service-Oriented Architecture (SOA) approach with the aim of achieving higher levels of interoperability (Huhns and Singh 2005; Papazoglou et al. 2007). SOA is focused on creating a design style, technology, and process framework that allow enterprises to develop, interconnect, and maintain enterprise applications and services efficiently and cost-effectively (Huhns and Singh 2005). A service in SOA is an exposed piece of functionality with three essential properties. Firstly, a service is self-contained in that it maintains its own state. Secondly, a service is platform-independent, implying that the interface contract to the service is limited to platform-independent assertions. Lastly, a service can be dynamically located, invoked and (re)combined (Huhns and Singh 2005; Papazoglou et al. 2007). Therefore, multiple services running over heterogeneous systems may then interact and be used as building blocks for new applications (Papazoglou et al. 2007).

2.2 SOA-specific scenarios

As previously mentioned, software architects can achieve the benefits of both fault-tolerant and service-oriented architectures by structuring variants as FT-compositions. Papazoglou et al. (2006, 2007, 2007) list a set of roles and functionalities that a service composition should encompass for the aggregation of multiple services into a single composite service. We use this list to classify elements of our primary studies as described below:

- *Interoperability Capabilities*: Whenever a service composition provides its functionalities by means of interfaces that are platform-independent, a client from any communication device using any computational platform, operating system, or programming language can reuse the solution. That is, the service composition aggregates services provided by other service providers into a distinct value-added service and may itself act as service provider (Papazoglou et al. 2006).
- *Autonomic composition of services:* service compositions should equip themselves with adaptive service capabilities so that they can continually morph themselves to

respond to environmental demands and changes without compromising operational and financial efficiencies (Papazoglou and Heuvel 2007). Examples of support for autonomic composition of services include automatically discovering new partners to interact with; automatically selecting partners and options that would, for example, maximize benefits and reduce costs (Papazoglou et al. 2006); and automatically detecting that some business composition requirements are no longer satisfied by the current implementation and react to requirement violations (Papazoglou et al. 2006).

- *QoS-aware service compositions:* To be successful service compositions need to be QoS-aware. For example, services should be composed in accordance with an extensible set of Quality-of-Service (QoS) properties and high-level policies (e.g. performance levels, security requirements, SLA stipulations, and so forth). QoS encompasses important non-functional service requirements, such as performance metrics (response time, for instance), security attributes, (transactional) integrity, reliability, scalability, and availability (Papazoglou and Heuvel 2007; Papazoglou et al. 2006).
- *Business-driven automated compositions:* a service composition at the business-level should pose the requirements, possibly from different stakeholders with conflicting needs, and the boundaries for the automatic composition at the system level. While the service composition at the business level should be supported by user-centered and highly interactive techniques, system level service compositions should be fully automated and hidden to the end users. System level compositions should be QoS-aware, should be generated and monitored automatically, and should also be based on autonomic computing principles (Papazoglou and Heuvel 2007; Papazoglou et al. 2006).

2.3 Software fault tolerance and design diversity

A fault is the identified or hypothesized cause of an error (Avizienis et al. 2004; Trivedi et al. 2010). An error is part of the system state that is liable to lead to a failure (Avizienis et al. 2004; Trivedi et al. 2010). A failure, in turn, occurs when the service delivered by the system deviates from the specified service (Avizienis et al. 2004). So, with software fault tolerance, we want to prevent failures by tolerating faults whose occurrences are known when errors are detected (Lee and Anderson 1990). When designing fault tolerance, a first prerequisite is to specify, by means of fault models, the faults that should be tolerated (Gärtner 1999). The next step is to enrich the system under consideration with components or concepts that provide protection against faults from the fault models (Gärtner 1999).

For instance, this work specifically addresses software faults According to Pullum (2001), '*software faults may be traced to incorrect requirements (where the software matches the requirements, but the behaviour specified in the requirements is not appropriate) or to the implementation (software design and coding) not satisfying the requirements*'. Software faults are also called design faults or bugs (Pullum 2001).

Software faults cannot be tolerated by simple replication of identical software components since the same mistake will exist in each copy of the components (Lee and Anderson 1990). A solution to this problem is to introduce diversity into the software replicas (Lee and Anderson 1990; Lyu 1996; Wilfredo 2000). Design diversity is the provision of

functionally-equivalent software components, called variants, through different designs and implementations.

Design diversity begins with an initial requirements specification. Each developer or development organization is responsible for a variant and implements the variant according to the specification (Gärtner 1999; Laprie et al. 1990; Pullum 2001). Figure 1 illustrates the basic design diversity concept. Inputs are distributed to variants. The variants execute their operations and produce their results, from which a single correct or acceptable result must be derived, if any (Pullum 2001). The mechanism responsible for this task is called an *adjudicator*.

Adjudicators generally come in two flavours, voters and Acceptance Tests (ATs). A brief description of voter characteristics and differences are presented in Section 3.1, in the context of the proposed taxonomy. We refer to Pullum (2001) for a more detailed description about the various types of adjudicators and their operations (pages 269-324). For example, specific adjudicators covered by Pullum (2001) are exact majority, consensus, formal consensus, formal majority, median, mean, weighted, and dynamic voters; acceptance tests can be based on satisfaction of requirements, accounting tests, computer run-time acceptance tests and reasonableness acceptance tests.

The philosophy behind design diversity is to decrease the probability that variants fail at the same time for the same input value because this usually makes failures of variants detectable (Lyu 1996). To illustrate this point with a hypothetical example, Figure 2 presents a diversity-based solution that leverages three variants and a majority voter to tolerate software faults. For instance, two variants present software faults, however these variants do not fail for the same input cases. Consequently, for the provided input, the

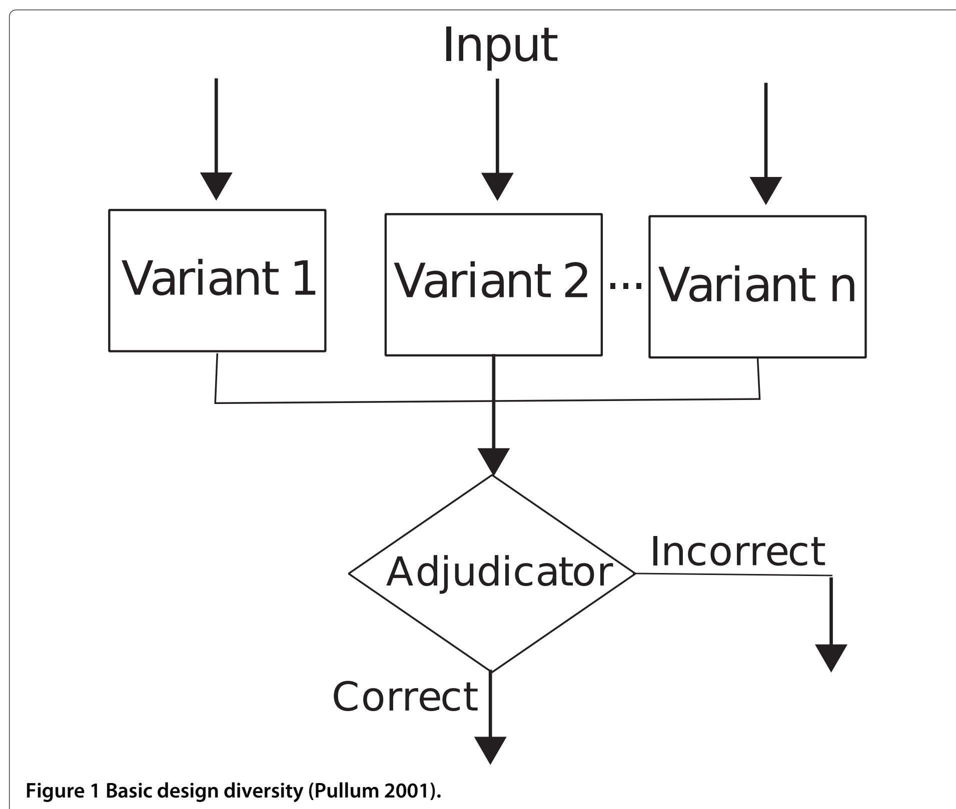

Figure 1 Basic design diversity (Pullum 2001).

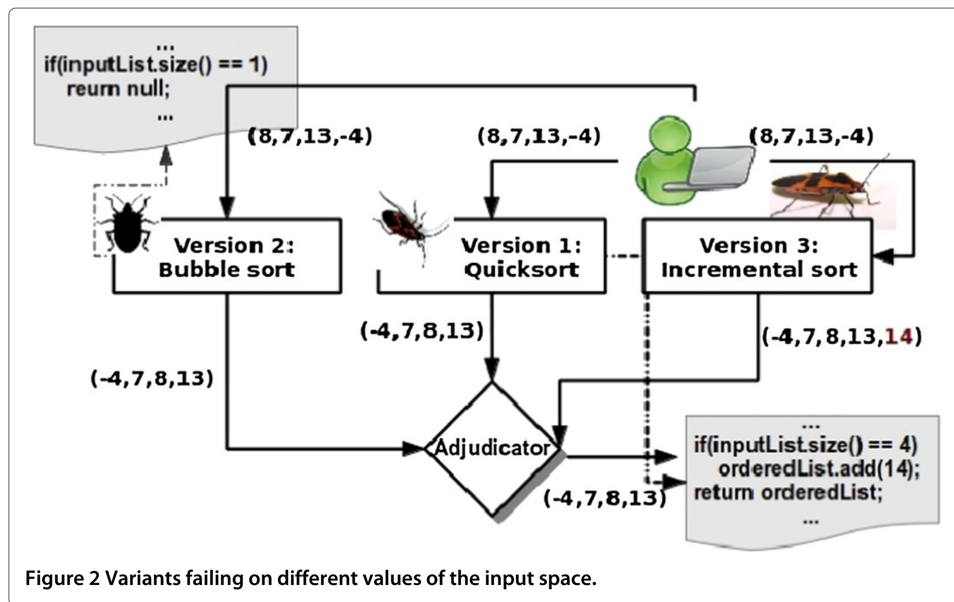

Figure 2 Variants failing on different values of the input space.

majority voter is able to tolerate the software fault whose activation has led to failure of one of the variants. Figure 3 illustrates a diversity-based solution that leverages three variant services such that two of them fail on the same input value, leading to a failure of the majority voter as whole.

2.4 Error recovery

Error recovery is the process in which the erroneous state is substituted with an error-free state (Lee and Anderson 1990). Error recovery is performed using either backward

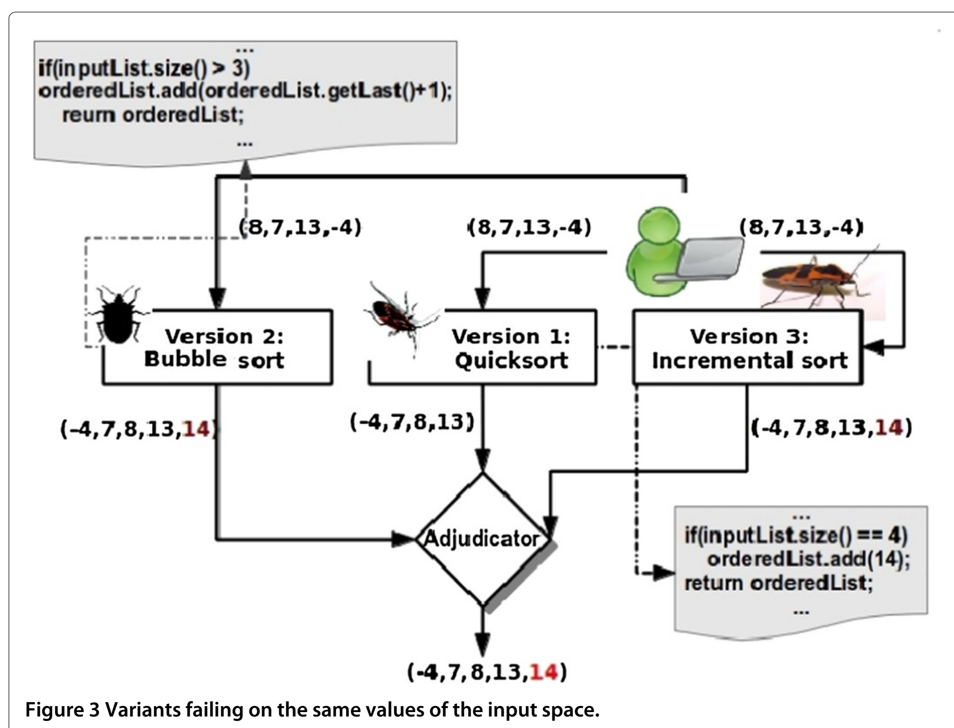

Figure 3 Variants failing on the same values of the input space.

recovery or forward recovery. On the one hand, *backward recovery* attempts to return the system to a correct or error-free state by restoring or rolling back the system to a previously saved state, which is assumed to be error-free. On the other hand, *forward recovery* attempts to return the system to a correct or error-free state by finding a new state from which the system can continue operation. Compared to backward error recovery, forward recovery is usually more efficient in terms of the overhead (*e.g.* time and memory) it imposes (Lee and Anderson 1990). On the other hand, it is usually not possible to design general forward recovery mechanisms.

3 Method

3.1 A taxonomy for software fault tolerance based on design diversity for SOA

We define a taxonomy combining elements from the fault-tolerance domain (Section 2.3) with elements from the SOA domain (Section 2.2).

Figure 4 presents the taxonomy, which considers the common design issues and their different design solutions, as well as the specific roles desirable in SOA-based applications. The proposed taxonomy was adopted to classify our primary studies. Both design issues and decisions were derived from the analysis of fault tolerance techniques based on design diversity (*e.g. Recovery Blocks*, *N-Version Programming*, *N-Self Checking Programming*, *Consensus Recovery Block* and *Acceptance Voting*) and adjudicators (Elmendorf 1972; Horning et al. 1974; Kim 1984; Laprie et al. 1990; Lee and Anderson 1990; Lyu 1996; Pullum 2001; Scott et al. 1987). We also considered the reliable hybrid pattern structure proposed by Kim and Vouk (1997). In comparison with their work, our work *(i)* identifies different types of voters and acceptance tests based on the general taxonomy of adjudicators presented by Pullum (2001); and *(ii)* explicitly distinguishes the different schemes of variant execution (i.e. sequentially or in parallel). Different design decisions employ different measures of quality requirements (Pullum 2001). These differences make each design solution suitable for a particular application. In Section 4.1, we briefly

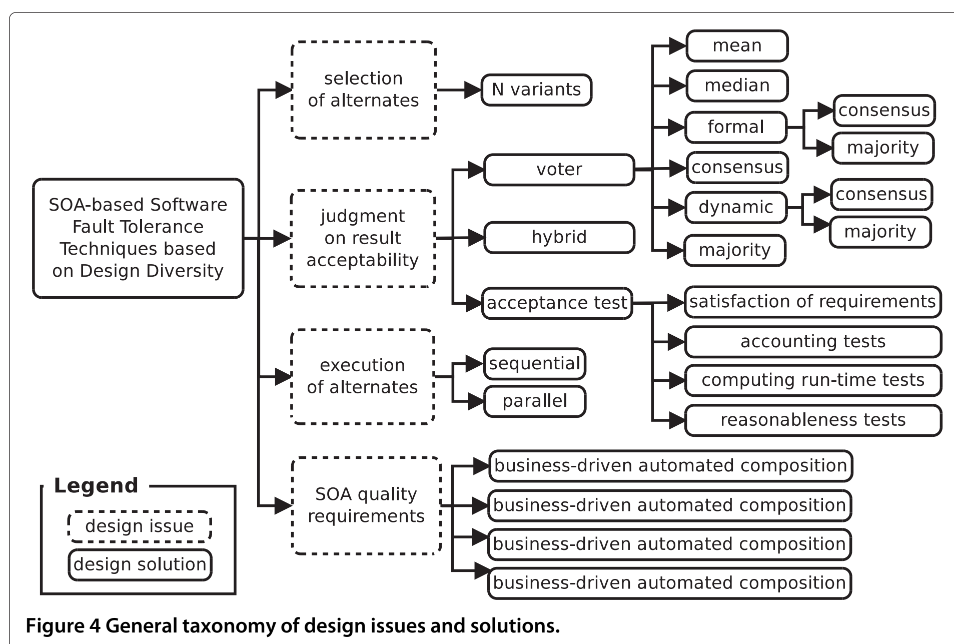

Figure 4 General taxonomy of design issues and solutions.

compare the described design solutions and present some general remarks about their effectiveness.

The elements of the taxonomy are described below.

Design issue I - selection of variants: The number of variant software components (n) must each be provided by components with different software designs and implementations. The main goal of increasing diversity is to detect faults in variant when the variants fail on disjoint subsets of the input space. Even if variants fail on overlapping subsets of the input space, design diversity is still considered a useful measure of reducing the risk from design faults (Laprie et al. 1990; Lyu 1996). If variants have minimal overlaps between their failure regions, a reliable configuration may be formed even when variants individually have modest reliability. Finally, variants might be chosen at different points during the software lifecycle.

Design issue II - judgement on result acceptability: Adjudicators, or decision mechanisms, generally come in two flavours, voters and Acceptance Tests (ATs).

Voters: Voters are based on a relative judgement on result acceptability by comparison of variant results (Pullum 2001). We present an overview of voters that are mostly described in the literature (Lee and Anderson 1990; Pullum 2001). We refer to Pullum (2001) for further details on voter procedures and pseudocode.

- **Exact majority voter:** The exact majority selects the value of the majority of the variants as its presumably the correct result (McAllister and Vouk 1996). This voter is also known as the m-out-of-n voter (Pullum 2001). The agreement number, m, is the number of versions required to match for system success (Eckhardt and Lee 1985). The total number of alternatives, *i.e.* n, is rarely more than 3. Consequently, the majority voter is generally seen as a 2-out-of-3 voter.
- **Consensus voter:** This voter allows the selection of a consensus or set of matching variant results as the adjudicated result if no majority exists (Pullum 2001). That is, this voter is a generalization of the majority voter (Vouk et al. 1993).
- **Formal consensus and Majority voter:** The *Formal Consensus* and *Majority* voters are variations of, respectively, the consensus and the exact majority voters (Pullum 2001). Basically, the formal voter uses a comparison tolerance indicating the maximum distance allowed between two correct output values for the same input. In this way, variant results that are different, but quite close together, are the adjudicated correct answers.
- **Median voter:** The median voter selects the median of the variant output values as its adjudicated result. Variant outputs must be in an ordered space (Pullum 2001).
- **Mean and weighted voter:** The mean and weighted voter select, respectively, the mean or weighted average of the variants' output values, which are in an ordered space, as the adjudicated result (Broen 1975). Additional information related to the trustworthiness of the alternatives might be used to assign weights to the variant outputs, if using the weighted average voter (Pullum 2001).
- **Dynamic majority and Consensus voters:** Unlike the previously described voters, dynamic voters are not defeated when any variant fails to provide a result (Pullum 2001). Dynamic majority and consensus voters operate in a way similar to, respectively, majority and consensus voters, with the exception that dynamic voters can handle a varying number of inputs (Pullum 2001). When the dynamic voter

adjudicates upon two results, a comparison takes place. When comparing, if the results match, the matching value will be output as the correct result. Otherwise, no selected output will be returned.

Acceptance Tests (ATs): ATs rely on an absolute judgement with respect to a specification (Lee and Anderson 1990; Pullum 2001). With ATs, only one variant is executed at a time. The AT is responsible for checking whether the produced result is correct. In case it is not, another variant is executed until a correct result is obtained, if possible.

- **Acceptance tests based on satisfaction of requirements:** ATs are constructed with conditions that must be met at the completion of variant execution (Pullum 2001). These conditions might arise from the problem statement of the software specifications.
- **Accounting tests:** Accounting ATs are suitable for transaction-oriented applications with simple mathematical operations (Pullum 2001). For example, when a large number of records are reordered or transmitted, a tally is made of both the sum over all records and the total number of records of a particular data field. These results can be compared between the source and the destination to implement an accounting check AT (Pullum 2001).
- **Computer run-time tests:** Run-time tests detect anomalous states such as overflow, undefined operation code, underflow, write-protection violations, or end of file (Pullum 2001).
- **Reasonableness tests:** These ATs are used to determine if the state of an object in the system is reasonable, *e.g.*, precomputed ranges or expected sequences of program states (Pullum 2001).

  Hybrid adjudicators: A hybrid adjudicator generally incorporates a combination of AT and voter characteristics. For example, variant results are evaluated by an AT, and only accepted results are sent to the voter (Laprie et al. 1990).

Design issue III - execution of variants: Variants can be executed either sequentially or in parallel. The execution schemes should provide all variants with exactly the same experience of the system state when their respective executions start to ensure consistency of input data (Nascimento et al. 2013), which be can be achieved by employing backward recovery or forward recovery (*Section 2.4*). Sequential execution often requires the use of checkpoints (it usually employs backward recovery), and parallel execution often requires the use of algorithms to ensure consistency of input data (it usually employs forward recovery by invoking all the variants and coordinating their execution through a synchronization regime) (Pullum 2001; Wilfredo 2000).

- **Sequential:** in implementing a sequential execution scheme the variants are executed one at a time. Generally, in the sequential execution scheme, the most efficient variant, *e.g.* in terms of response time or financial cost, is located first in the series, and is termed *primary variant*. The less efficient variants are placed serially after the primary variant and are referred to as (secondary) variants. Thus, the resulting rank of the variants reflects the graceful degradation in the performance of the variants (Pullum 2001).

- **Parallel:** in the parallel execution scheme, variants are executed concurrently. The resulting outputs can be provided to the adjudicator in an *asynchronous* fashion as each version completes, or in a *synchronous* manner (Daniels et al. 1997).

3.2 Systematic literature review method

This review has been conducted as a systematic literature review based on guidelines proposed by Kitchenham and Charters (2007). The guidelines cover three phases of a systematic review: planning, executing and reporting. In our work, the goal of the review is to provide a better understanding of diversity-based approaches for FT-compositions (Kitchenham and Charters 2007), in order to support decisions of the software architect in terms of reuse. In the following, Section 3.3 presents details about the planning of the systematic review, while Section 4.2 presents details about the execution and results.

3.3 Planning

3.3.1 Research questions

This work aims to answer the following research questions.

RQ1 What design issues and respective design solutions related to the fault tolerance taxonomy of Figure 4 are being addressed?

RQ2 What SOA specific requirements are being addressed?

To address RQ1, we classify existing solutions using the proposed taxonomy of design solutions for fault tolerance based on design diversity (Figure 4). To address RQ2, we consider a list of quality requirements that service compositions should address, presented in Section 2.2. Related to RQ1 and RQ2, we also describe some important design issues related to general software fault tolerance techniques, and analyze such solutions in terms of preserving important requirements inherent of SOA.

3.3.2 Search process

Searches for primary studies were performed upon databases of Software Engineering research that met the following criteria (Williams and Carver 2010):

- Contains peer-reviewed software engineering journals articles, conference proceedings, and book chapters.
- Contains multiple journals and conference proceedings, which include volumes that range from 2000 to 2012.
- Used in other software engineering systematic reviews (e.g. (Cardozo et al. 2010; Jorgensen and Shepperd 2007; Kitchenham et al. 2007; Williams and Carver 2010)).

The resulting list of databases was: *(i)* ACM Digital Library; *(ii)* IEEE Electronic Library; *(iii)* SpringerLink; *(iv)* Scopus; and *(v)* Scirus (Elsevier).

3.3.3 Search string

A search string was created to extract data from each database. We adopted various combinations of terms from *(i)* the main purpose of this review; *(ii)* the research question (Kitchenham and Charters 2007; Kitchenham et al. 2009) and *(iii)* meaningful synonyms and variant spellings. Whenever it was necessary, this search string was decomposed into several search terms (*e.g.* Recovery Block AND Service-Oriented Architectures) due

to restrictions imposed by some of the search engines. The resulting search string is summarised in following:

(*fault tolerance OR diversity OR fault-tolerant OR redundancy OR Recovery block OR N-Version Programming OR Distributed Recovery Blocks OR N Self-Checking Programming OR Consensus Recovery Block OR Acceptance Voting OR dependability OR dependable OR reliable OR reliability*) $<$ *AND* $>$ (*service-oriented architecture OR SOA OR service computing OR SOC OR web services*)

3.3.4 Study selection

The database searches resulted in a large number of candidate papers. We adopted study selection criteria to identify those studies that provide direct evidence about the research question.

Inclusion criteria:

- Approaches based on software diversity for FT-Compositions that specifically focused on web services. Additionally the solutions that were targeted supported one or more issues identified in the taxonomy (Figure 4), *i.e.*, selection of variant services, execution of variants and judgement on result acceptability.

Exclusion criteria:

- Solutions for reliable SOA-based applications employing solely replicas of identical services - although the adoption of identical replicas can improve system availability, they are not able to tolerate *software* faults (Gärtner 1999; Lee and Anderson 1990).
- Solutions for fault-tolerant SOAs based on data diverse software fault tolerance techniques.
- Solutions for fault-tolerant SOA relying solely on exception handling - variant services are not employed as part of the exception handling mechanism.
- Duplicate reports of the same solution - when several reports of the proposed solution exist in different papers the most complete version of the study was included in the review.
- Short papers, introductions to special issues, tutorials, and mini-tracks.
- Studies presented in languages other than English.
- Papers addressing empirical studies on fault tolerance based on design diversity applied to *SOAs* - not proposing any particular solution to employ FT-compositions.
- Grey literature, that is, informally published written material.

These criteria were applied as performed in (Kitchenham and Charters 2007; Williams and Carver 2010):

1. Reading the title in order to eliminate any irrelevant papers.
2. Reading the abstract and keywords to eliminate additional papers whose title may have fit, but abstract did not relate to the research question.
3. Reading the introduction and, whenever it is necessary, the conclusion to eliminate additional papers whose abstract was not enough to decide whether the inclusion/exclusion criteria are applicable.

4. Reading the remainder of the paper and including only those that addressed the research question.

3.3.5 Data collection and synthesis

We created a data extraction form to collect all the information needed to address the review question (Kitchenham and Charters 2007). The contents of the designed data form are composed by (Kitchenham and Charters 2007; Kitchenham et al. 2009): the source (*e.g.* journal, conference) and full reference; date of extraction; summary of the proposed solution; supported design issues/solutions (Figure 4); and space for additional notes.

4 Results and discussion

4.1 Design decision guidelines to realize fault tolerance in SOA

The proposed solution encompasses a set of guidelines to support the architect in key decisions relating to fault tolerance. The proposed guidelines address the design decisions of the taxonomy presented in Section 3.1 (Figure 4). Figure 5 presents the activities that are part of the proposed guidelines. First, we show guidelines which advise the architect to specify the failure modes of the system. Then, the guidelines lead the architect to specify the fault latency and the scenarios of fault tolerance. For example, Byzantine faults (intermittent faults) could be acceptable (Lee and Anderson 1990; Randell 1975) whereas other types of faults, such as permanent faults, may be unacceptable. This information may influence the choice of the adjudicator type (e.g., majority election, average value).

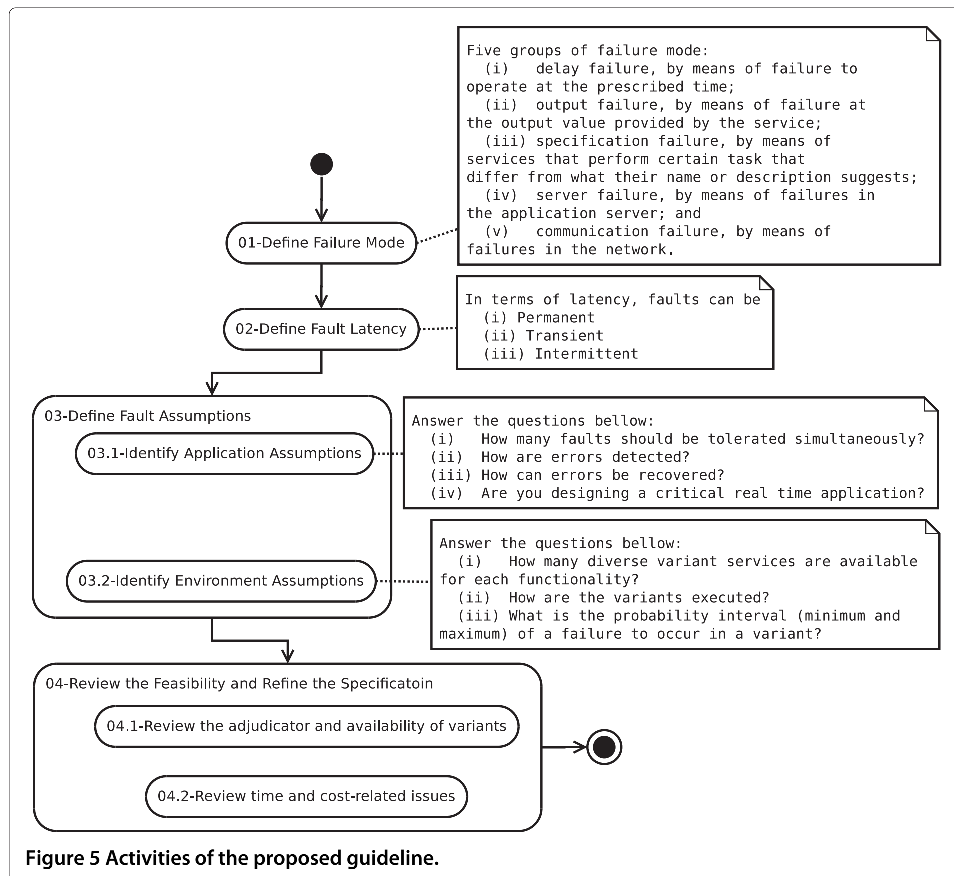

Figure 5 Activities of the proposed guideline.

Finally, the guidelines assist the architect in setting assumptions, in order to outline the scope of fault tolerance. After following the guidelines, the architect will have a classification of the features related to fault tolerance, following the classification of the taxonomy presented.

At the end of the process, the architect can assess whether the projected solution is feasible with respect to the availability of resources: e.g., number of variants. Depending on the number of variants needed to meet the demands of the system, the architect can now assess if it is necessary to reduce the scope of fault tolerance. For example, intermittent faults, also known as Byzantine faults, are usually detected and tolerated utilizing design diversity and majority election techniques. In this case, the number of variants will be directly affected by the number of concurrent faults to be tolerated. Depending on the available variants, the number of concurrent tolerated faults should be decreased.

In the following sections (4.1.1 to 4.1.6) we suggest four steps for designing fault tolerance: (i) define failure mode; (ii) define fault latency; (iii) define fault assumptions; and (iv) review the feasibility and refine the designed solution. After reviewing the solution, the next step is to try to reuse existing solutions. To support the architect, Section 4.2 presents a systematic literature review that classifies existing approaches according to the same taxonomy presented in Section 3.1 that was also used by the guidelines presented here.

4.1.1 Define failure modes

Failure modes can be seen as ways in which a software, equipment, or machine failure can occur. The specification of the failure mode helps the identification of faults in applications, which are the underlying causes of failures or which ones initiate a process which leads to failure.

In terms of SOA-based applications, we consider five groups of failure modes: (i) delay failure, which is a failure to operate at the prescribed time; (ii) output failure, when the output value provided by the service is incorrect; (iii) specification failure, when a service performs a task that differs from what their name or description suggests; (iv) server failure, a failure in the application server; and (v) communication failure, a failure in the network.

The architect must indicate which failure modes need to be addressed within the application model. That is, if its behaviour detection and handling should be part of the application scope. The answer here should be either yes or no. Details about how to consider each group are defined in the next steps: fault latency and fault assumptions.

4.1.2 Define fault latency

Fault latency is defined as the interval between the moments of fault occurrence and error generation (Shin and Lee 1984). This information is extremely important, since it directly affects how to handle the error caused by the fault. In terms of latency, faults can be permanent, transient or intermittent. Permanent faults are continuous and stable. In hardware, permanent faults reflect an irreversible physical change. Permanent faults are also known as “hard” faults. Transient faults result from temporary environmental conditions, and can be fixed by changing the respective conditions that cause the fault. Transient faults are also known as “soft” faults. Intermittent faults are only occasionally present due to unstable hardware or varying hardware or software states. Intermittent

and transient faults are the major source of system errors. Even when it is not possible to repair the fault, the use of redundant resources allows the system to tolerate faults (Gärtner 1999; Lee and Anderson 1990). The manifestations of transient and intermittent faults and of incorrect hardware or software design are much more difficult to determine than permanent faults.

Fault latency is part of the fault model. The fault model is a design model that tries to predict what could go wrong, as well as the consequences of particular faults. In order to detail the fault model, the proposed guidelines state that, for each failure mode identified (Section 4.1.1), the architect has to specify how the error caused by the fault is expected to occur: permanently, transiently, or intermittently.

For example, a delay failure (failure mode) is characterised by performance problems, when a service says to execute in a certain period of time, but for some reason it is not able to do so. If the architect is considering this kind of failure in the failure mode, it is necessary to define how these failures can occur: permanently, in a transient way, or in an intermittent way. Considering it as a permanent fault, the error could be detected by time-out and a solution with two variant services would be sufficient to tolerate the error detect the problem by changing the faulty service to another one. If it is a transient fault caused by server overload, the handling could be, for example, to change the server. But assuming a fault model of intermittent faults, in some execution a certain service is faster than another, but in the following execution the scenario can be the opposite. So, redundant services could be executed in parallel and the adjudicator could consider the first reply it receives.

4.1.3 Define fault assumptions

The definition of *fault assumptions* contains information relating to the error detection and error recovery mechanisms. The proposed solution focuses on two types of assumption: (i) application assumptions; and (ii) environmental assumptions. Section 4.1.4 presents the guidelines for identifying assumptions related to the application, while Section 4.1.5 presents the guidelines for identifying assumptions related to the environment.

4.1.4 Identify application assumptions

In order to identify application assumptions, we outline four questions to be answered:

1. *How many faults should be tolerated simultaneously?* The answer to this question should be a number. It will interfere with the number of variants required to tolerate the faults. This also depends on the adopted design solution.
2. *How are errors detected?* The answer to this question can be either *by comparison* or *by checking a single result.* It aims to identify functionalities whose validity can be verified through acceptance tests, with no need of redundancy for error detection. The answer can also affect the number of variants required for fault tolerance.
3. *How can errors be recovered?* The answer to this question can be either *forward* or *backward* error recovery, according to the error recovery strategy of the respective functionalities (see Section 2.4). The answer to this question should support the decision regarding how to execute variants. In some cases, when a forward error recovery strategy is more effective, a sequential execution is recommended. For example, when a billing system is executed in parallel, after

processing redundant bills, some of them should be cancelled, since an undo operation is not applicable in such case.

4. *Are you designing a critical real-time application?* The answer of this yes/no question also supports the decision of which execution strategy to use. In case of a critical real-time system, a parallel execution could be preferable, if redundancy is needed.

4.1.5 Identify environment assumptions

Environment assumptions are restrictions related to the services' availability, the diversity amongst the variant services available, as well as existing execution policies. For this, three questions should be answered:

1. *How many diverse variant services are available for each functionality?* The answer to this question should be a number. It also supports the final decision regarding the appropriate design solution. For example, depending on the compatibility between the number of faults to be tolerated and the number of diverse variant services available, the feasibility of a fault-tolerant solution may not be possible.
2. *How are the variants executed?* The answer to this question can be *for free*, *cheap*, or *expensive*. Such answer should support the decision of which execution strategy to use (sequential or in parallel). For example parallelism may have expensive running costs.
3. *What is the probability interval (minimum and maximum) of a failure to occur in a variant?* The answer to this question can guide the architect in choosing and discarding variants based on monitoring data. Besides, this information may also interfere with the choice of the adjudicator, as discussed in Section 4.1.6.

Finally, for each failure mode and fault latency identified by the architect, a proper handling behaviour should be specified. Table 1 illustrates handling behaviour for errors related to faults, according to the failure mode and the assumed fault latency. The example considers scenarios involving the reliability quality attribute. It is important to know that the architect may assume more than one fault latency for the same failure mode. In these cases, the more general solution, which fits all the latencies, should be used.

4.1.6 Review the feasibility and refine the specification

After defining the initial specification related to fault tolerance, the last step is to review the feasibility of the designed solution and refine it based on concepts related to classical fault tolerance techniques. In order to make this task easier, in the following we provide a summary comparing classical fault tolerance techniques, which is important for choosing a design solution, giving a special attention to the type of adjudicator. It is important to stress that the design issues presented are compliant with the taxonomy presented in Figure 4. Instead of presenting a full discussion on such techniques, we summarise some of the findings on design diversity, in particular the findings related to non-functional characteristics of the described design solutions, represented by the taxonomy. The analysis of these characteristics provide a good support to define the system fault tolerance (*Section 2.3*). A summary of results and discussion for supporting the SOA architect on choosing fault tolerance techniques is presented in the following. A detailed discussion in the context of a literature review of existing solutions is presented in Section 4.2.

Table 1 Example of fault tolerance specification using the guidelines

Failure mode	Fault latency	Fault assumption	How to tolerate
Delay	Permanent	Inefficient Algorithm	Change the faulty service to another one
		Hardware problem	Change the application server
	Transient	Server overload	Change the application server
	Intermittent	Environmental instability	Execution in parallel to get the fastest reply
Output	Permanent	Software bug	Use variant services to detect and majority election to tolerate
	Transient	Server overload	Use variant services to detect and majority election to tolerate
	Intermittent	Electromagnetic interference	Use variant services to detect and majority election to tolerate
Specification	Permanent	Malicious service	Use variant services to detect and majority election to tolerate
	Transient	DNS mistake due to environmental instability	Use variant services to detect and majority election to tolerate
	Intermittent	Unknown cause	Use variant services to detect and majority election to tolerate
Server	Permanent	Hardware failure	Change the application to a mirror server
	Transient	Memory overload	Change the application to a mirror server
	Intermittent	Environmental instability	Use redundant servers in parallel, and synchronisation algorithms for keeping the consistency among servers
Communication	Permanent	Breaking the optical fibber cable	Change the network link
	Transient	Network overload	Change the network link
	Intermittent	Electromagnetic interference	Use redundant communication channel in parallel

Review the adjudicator and availability of variants Firstly, it is essential to check the availability of variants that are sufficiently diverse in order to decrease the probability of occurrence of coincident failures (Eckhardt et al. 1991; Hilford et al. 1997; Knight and Leveson 1986; Lyu et al. 1994; Nascimento et al. 2012a). Regarding the judgement on the result acceptability, the adjudicator would run its decision-making algorithm on the results and determine which one (if any) to output as the presumably correct result. Just as we can imagine different specific criteria for determining the 'best' item depending on what that item is, so we can use different criteria for selecting the 'correct' or 'most acceptable' result to output. The probabilities of activation of related faults between variants are likely to be greater for voters than for acceptance tests (ATs) (Arlat et al. 1988; McAllister and Vouk 1996). But in general, ATs are more difficult to construct in practice because they are strongly application-dependent and because it is not always possible to determine a criterion to judge variant results (Di Giandomenico and Strigini 1990). As a consequence, voting is a more useful technique in a practical setting, because voting adjudicators are easier to develop (Wilfredo 2000).

The exact majority voter is most appropriately used to examine integer or binary results, but it can be used on any type of input (Saglietti 1992). The majority voter has a high probability of selecting the correct result value when the probability of a variant failure is less than 50% and the number of variants, n, is 'large' (Blough and Sullivan (1990) used n = 7 and n = 15 in the study). However, when the probability of a variant failure exceeds

50%, then the majority voter performs poorly (Blough and Sullivan 1990). In fact, all voters have a high probability of selecting the correct result value when the probability of variant failures is less than 50% (Blough and Sullivan 1990). A median voter can be defined for variant outputs consisting of a single value in an ordered space (*e.g.* real number). Median voter is a fast algorithm and is likely to select a correct result in the correct range (Blough and Sullivan 1990). If it can be assumed that, for each input value, no incorrect result lies between two correct results, and that a majority of the replicas' outputs are correct, then the median voter produces a correct output (Pullum 2001).

Consensus voting is more stable than majority voting and always offers reliability at least equivalent to majority voting (McAllister and Vouk 1996; Vouk et al. 1993). Nevertheless, in terms of implementation, the consensus voting algorithm is more complex than the majority one, since the consensus voting algorithm requires multiple comparisons (Lee and Anderson 1990). Blough performed a study on the effectiveness of voting algorithms (Blough and Sullivan 1990). He states that the median voter is expected to perform better than the mean voting strategy. He also shows the overall superiority of the median strategy over the majority voting scheme (Blough and Sullivan 1990). Furthermore, under circumstances in which some or all variants might not produce their results (*e.g.* some or all variants not providing their results within the maximum expected time frame; catastrophic failure of some or all of the variants), dynamic voters are the best option since they can process zero to *n* inputs. Finally, acceptance tests, exact majority, consensus and dynamic voters can process any type of variant outputs, while the remaining adjudicators must receive inputs in an ordered space (Pullum 2001).

We summarise some details of the described voters in Table 2, which is based on a summary table by Pullum (2001). We have added the type of variant results in which the voter is able to judge. This table states the corresponding recommended fault tolerance technique, given the type of variant results provided to the voter and the type of voter. To use this table, it is important to consider the primary concerns surrounding the application areas of the software system and details about the output space. For example, if safety is the primary concern, it is recommended to adopt the voter that would rather raise an exception and produce no selected output than present an incorrect output as a presumably correct one (*e.g.* the exact majority or dynamic majority voters) (Pullum 2001). If the primary goal is to avoid cases in which the voter does not reach a decision, i.e., *an* answer is better than no answer, than it is sufficient to adopt the voter that reaches a 'No output' result least often (*e.g.* the median voter) (Pullum 2001). Based on this criterion, exact majority voter, formal majority voter, and dynamic majority voter can be considered the safest voters, because they produce incorrect output 'only' in cases where most or all of the variants produce identical and wrong results.

Review time and cost-related issues With respect to the execution of variants, in the parallel scheme, there is an underlying assumption that sufficient hardware resources are available to enable the execution of variants concurrently. Even with sufficient parallelism, the execution time of this scheme will be constrained by the slowest version - there may be a substantial difference in the execution speeds of the fastest and slowest version because of the need to generate independent designs (Lee and Anderson 1990). When variants are executed in parallel, there is also a synchronisation time overhead. The time required to execute variants in a sequential way will range from the execution time of the primary

Table 2 Voter results given details about variant output space ((Pullum 2001) - page 310)

Variant results	**Voter**							
	Exact majority	**Median**	**Mean**	**Weighted average**	**Consensus**	**Formal majority**	**Dynamic majority**	**Dynamic consensus**
All outputs identical and correct	C	C	C	PC	C	C	C	C
Majority Identical and correct	C	C	PC	PC	C	C	C	C
Plurality identical and correct	NO	PC	PC	PC	C	NO	NO	C
Distinct outputs, all correct	NO	C	PC	PC	NO	NO	NO	NO
Distinct outputs, all incorrect	NO	I	PI	PI	NO	NO	NO	NO
Plurality identical and wrong	NO	PI	PI	PI	I	NO	NO	I
Majority identical and wrong	I	I	PI	PI	I	I	I	I
All outputs identical and wrong	I	I	I	I	I	I	I	I
Variant result type	Any type	Ordered space	Ordered space	Ordered space	Any type	Floating-point arithmetic	Any type	Any type

The voter outputs are:

- **C** - Correct: The voter outputs a correct result;
- **PC** - Probably correct: The voter outputs a result that is probably correct, but may be incorrect;
- **PI** - Probably incorrect: The voter outputs a result that is probably incorrect, but may be correct;
- **I** - Incorrect: The voter outputs an incorrect result;
- **NO** - No output: The voter does not output a result; an exception is raised.

variant (if acceptance tests are employed and the primary result is acceptable) to the sum of execution time of all variants, *e.g.* if all variant results are subjected to ATs (Buys et al. 2011). Nevertheless, this time will normally be constrained by the execution time of the primary variant (Lee and Anderson 1990). Furthermore, under some circumstances, a specific execution scheme is not applied, *e.g.*, when there is a processing cost charged for the use of variant services, invoking them in parallel might incur in greater actual cost,*i.e.*, in sequential schemes, not all variants are necessarily executed (Zheng and Lyu 2010a). Moreover, the error recovery strategy defined in the fault assumptions (backward or forward) might also interfere on the way variants should be executed.

In the following we present details about the systematic literature review involving solutions for composing fault-tolerant services. Such a review aims to support the architect in the task of reusing existing solution. The most relevant works are presented and classified in the target taxonomy (Figure 4).

4.2 Systematic literature review results

In this section, we present the results obtained.

4.2.1 Search results

The searches returned thousands of papers that were filtered down to 23, and then to 17 primary studies, three of which are journal articles (Mansour and Dillon 2011; Yuhui and Romanovsky 2008; Zheng and Lyu 2010a), three book chapters (Dillen et al. 2012; Gorbenko et al. 2005; Kotonya and Hall 2010), ten conference papers (Abdeldjelil et al. 2012; Buys et al. 2011; Gonçalves and Rubira 2010; Gotze et al. 2008; Looker et al. 2005; Nascimento et al. 2011, 2012b; Nourani 2011; Santos et al. 2005; Townend et al. 2005), and one workshop paper (Laranjeiro and Vieira 2007).

In the solutions by Dillen et al. (2012), Santos et al. (2005) and Mansour and Dillen (2011), despite each service being named a replica by the authors, these solutions were designed to tolerate different responses by means of adjudicator mechanisms. This suggests these solutions could be also implemented as a diversity-based solution. Therefore, these solutions were included as primary studies.

Regarding the papers excluded from the first filtered 23 publications, the articles (Zheng and Lyu 2008, 2010b) are short versions of another article (Zheng and Lyu 2010a), the article (Abdeldjelil et al. 2012) is also an extended version of the article (Faci et al. 2011). The study presented by Gorbenko et al. in (Gorbenko et al. 2009) and by Nascimento et al. (2012a) are based, respectively, on the solutions proposed in (Gorbenko et al. 2005) and in (Nascimento et al. 2012b) - therefore, we consider the proposed solutions, i.e. (Gorbenko et al. 2005; Nascimento et al. 2012b), as primary studies. Xu (2011) examined challenges in the fields of dependability and security that need to be addressed carefully in order to provide sufficient support to enable service-oriented systems to offer non-trivial Quality of Service guarantees. Then, Xu presents several advanced techniques developed at the University of Leeds to achieve dependability and security in service-oriented systems and applications, including, the solution proposed by Townend et al. (2005). Therefore, only the work by Townend et al. (2005) is included as a primary study. Milanovic and Miroslaw (2007) also highlight the use of techniques to tolerate software faults in SOA, including techniques based on software diversity (e.g. N-Versions); however, no specific solution is proposed in this direction.

4.2.2 Classification of the primary studies

In Additional file 1: Supplementary Table, we present the summary of the design solutions supported by the analysed primary studies. Each primary study was classified as follows:

Y(yes), the design solution is supported by a primary study; **N(no)**, no information at all about the design solution is specified; **U(unknown)** according to the authors the design issue is supported, however, what design solutions are supported cannot be readily inferred.

It is important to emphasise that Additional file 1: Supplementary Table was fashioned to show which design issues have been addressed by existing approaches for FT-compositions. It might be meaningless to rank the primary studies based solely on their 'quantity of *Yes*' since the studies present different purposes (e.g. some solutions are mainly focused on supporting the selection of variant services (Nascimento et al. 2012b; Townend et al. 2005), while other ones are focused on executing variant services (Gonçalves and Rubira 2010; Yuhui and Romanovsky 2008)). Although it's difficult to compare these solutions, Section 4.1 presents non-functional characteristics related to the design solutions supported by the authors - thus supporting researchers' decision making when selecting design solutions more adjusted to different clients requirements.

In the next subsections, we discuss the answers to the two research questions presented in Section 3.3.1 and present the main proposed solutions regarding the design issues (Figure 4). Because of space limitations the summaries of solutions are representative rather than exhaustive.

4.2.3 Selection of variant services

The aims of selecting variants are twofold: *(i)* to increase the probability of selecting variants that are provided by different designs and implementations; *(ii)* to determine an appropriate degree and/or selection of variant services targeting an optimal trade-off between reliability measures as well as performance-related factors such as timeliness, cost and resource consumption. In general, variants have been chosen at different points during the software lifecycle. For instance, they can be chosen at design time by the engineer, configured manually once the software is deployed, or even be discovered and selected at runtime by the software itself.

The solutions by Townend et al. (2005) and by Nascimento et al. (2012b) address the issue of ensuring that variant services are diverse. Townend et al. (2005) aims to detect diverse designs during runtime. This is achieved by monitoring previous results and flow of data from a variant service using interaction provenance in order to reveal evidence that two variants share similar services or workflows. Such evidence may include matching common-mode failures that have propagated back from two variant services. Nascimento et al. (2012b) propose an experimental setup to investigate, from clients' viewpoint and by means of statistical tests, whether variant services present a difference in their outputs and their failure behaviours. Their solution also investigates if and by how much the use of FT-composition improves reliability when compared to a single non-fault-tolerant service (Nascimento et al. 2012b).

Variant services may be chosen using different measurements at different points throughout the execution. Buys et al. (2011) and Dillen et al. (2012) propose a fault tolerance strategy that autonomously changes the amount of redundancy or the selection

of variant services. The architecture proposed by Buys et al. (2011) bases this decision upon the current execution context and clients' requirements at the time of request. They propose a measure to infer combinations of variant services that are, in fact, effective. This measure quantifies the historical effectiveness of each variant service by penalising or rewarding it when it disagrees or complies with the majority decision respectively. Differently to this, Gotze et al. (2008) propose a solution where every atomic service provides information about its dependability attributes and every composite service has to provide additional information about their external services that are used to provide the desired functionality. The calculated dependability attributes and probability values of the resulting composite service are then used to manually optimize the composite service towards the user's expectations (Gotze et al. 2008), differing from the solution by Buys (2011) and Dillen et al. (2012) in which variant services are automatically selected.

The solution by Nascimento et al. (2011), Abdeldjelil et al. (2011,2012), Chen and Romanovsky (2008) and Zheng and Lyu (2008,2010b,2010a) allows the dynamic selection of variant services based on a priority schemes where the client defines requirements in terms of QoS. QoS values are updated by monitoring procedures in (Abdeldjelil et al. 2012; Faci et al. 2011; Nascimento et al. 2011; Yuhui and Romanovsky 2008) and by encouraging users to contribute their individually-obtained QoS information of the target Web services in (Zheng and Lyu 2008,2010b,2010a). The solution by Kotonya and Stephen (2010) and by Mansour and Dillon (2011) support a QoS matching scheme that will prioritise services based on reliability and performance metrics. These metrics are not shared with the client, i.e., a client cannot express preference for specific QoS metrics. The solution by Gorbenko (2005) monitors dependability attributes and selects an appropriate service based on them. However, the motivation behind this solution was to manage service upgrades, i.e. switching from an old version to a new version online when the level of dependability of the new service is acceptable.

4.2.4 Execution of variant services

Both parallel and sequential execution schemes have been addressed within existing approaches. The solution by Nascimento et al. (2012b) does not aim to support multiple execution schemes. The solutions by Gotze et al. (2008), Townend et al. (2005) supports execution of variants; however, they do not specify how variants might be executed in parallel, sequentially or both. Most solutions support both execution schemes (Abdeldjelil et al. 2012; Buys et al. 2011; Dillen et al. 2012; Gonçalves and Rubira 2010; Gorbenko et al. 2005; Laranjeiro and Vieira 2007; Mansour and Dillon 2011; Nascimento et al. 2011; Nourani 2011; Yuhui and Romanovsky 2008; Zheng and Lyu 2008,2010b,2010a) the solution by Looker et al. (2005), Santos et al. (2005) and by Kotonya and Stephen (2010) support the parallel execution of variants. An important characteristic of the execution scheme proposed by Laranjeiro and Vieira (2007) is that it can use functionally equivalent web services that have different interfaces, providing developers with more options to build their solutions (*i.e.* input/output adapters). However, their solution operates on a static connectivity mode, requiring static generation of local proxy classes for each variant service (Laranjeiro and Vieira 2007). The solution by Abdeldjelil et al. (2012) allows for the execution of variant services that present diversity also in their interfaces and results by mapping operations and parameters in the domain ontology.

4.2.5 Judgement on result acceptability

A variety of decision mechanisms were found amongst the target papers. The main advantage of diversity-based solutions is that they allow for variant service responses to be compared using a large choice of voter and acceptance test techniques. For instance, majority, consensus, formal and dynamic voters are addressed by some of the existing approaches for FT-compositions. Acceptance tests (ATs) are also supported; however, as they are strongly application-dependent there are fewer approaches supporting ATs.

Majority voters Majority voters are supported by Nourani et al. (2009,2011), Looker et al. (2005), Nascimento et al. (2011) and Zheng and Lyu (2008,2010b,2010a). The solution by Nourani et al. (2009,2011) supports the implementation of FT-compositions by means of the WS-BPEL. In this sense, the authors present details on the prototype implementation by identifying the BPEL structured activities adopted. The adjudication mechanism is also implemented as web service and invoked from a BPEL process. The authors mention the adoption of voters and adjudicators, including majority, dynamic, consensus, acceptance test and hybrid adjudicator. Nevertheless, no details on voting procedures are provided, except for the majority one. Differently from other solutions, they utilise diversity to allow two majority voting mechanisms to be applied to the same input set. In the BPEL process, if for any reason the voter faces faults or if there is an absence of consensus, the second version of voter is invoked using the same inputs. This avoids the voting process to becoming a single point of failure (Nourani 2009; Nourani and Azgomi 2011).

The Web Service-Fault Tolerance Mechanism (WS-FTM) proposed by Looker et al. (2005) supports the majority voter that allows generic result comparison. Nascimento et al. (2011) and Zheng and Lyu (2008,2010b,2010a) do not present additional information on voting procedures. We emphasize, however, that the solution by Zheng and Lyu (2008,2010b,2010a) supports different fault tolerance strategies and the authors describe in detail a dynamic fault tolerance strategy selection algorithm.

Consensus voters A variety of solutions support consensus voters. Nourani et al. (2009,2011) argues that their solution provides support consensus voters. In the solution by Santos et al. (2005), there is a component responsible for arbitrating the adjudicated output based on the output with the highest number of occurrences. Therefore, we inferred that their decision mechanism is based on the consensus voter. The benefits of diverse components is not only limited to the services, in the solution proposed by Laranjeiro and Vieira (2007) the voter protocol supports two voting mechanisms to be utilized at the same time: a unanimous voter and a consensus voter. Their solution also supports an evaluation mechanism that performs a continuous assessment of the quality of services. During the voting process, if an impasse occurs the QoS values of the variants are used to select one response. According to the authors, the impasse occurs whenever different variant results present the same number of occurrences (Laranjeiro and Vieira 2007). However, in their solution, it is not clear how they judge variant result acceptability when variants are executed sequentially.

Formal voters When dealing with greater variability in variant service interfaces, a straightforward consensus voting mechanism may not suffice. Some solutions accept a

certain amount of variability and agree that a consensus is formed with multiple equivalent results. The solution by Dillen et al. (2012) supports formal consensus voting, called plurality voting in their paper. For each invocation of the scheme, the variant services will be partitioned based on the equivalence of their results. The result associated to the largest cluster will be accepted as the correct result (Dillen et al. 2012). Abdeldjelil et al. (2011,2012) describe a specific voting algorithm, called equivalence vote, that will decide if multiple concurrent service responses are equivalent or not. This is based on a pre-agreed amount of deviation for answers to be considered equivalent. Their algorithm requires the input to be an ordered set of equivalent results and also for the acceptable functional deviation to be specified.

Dynamic voters Dynamic consensus and dynamic majority voters are supported by a number of solutions (Buys et al. 2011; Gotze et al. 2008; Nourani 2011; Nourani and Azgomi 2009; Townend et al. 2005). This may be more suitable for applications where it is acceptable for some variant services to fail and an answer still be returned from the remaining variant services. Due to the intricacies of dynamic decision mechanisms defined in our taxonomy some classifications in our review have been changed from the classifications used in the target papers. The solution by Gotze et al. (2008) define three FT-protocols, namely, *one*, *any* and *majority*. Only the majority operator allows for the enhancement of reliability and availability. The remaining operators are mainly focused on achieving high levels of availability and are out of the scope of this work. They define the majority as an operation that schedules the same request to all defined services. Afterwards this operation uses the results of all services that did not fail and chooses the most common result. Based on this general definition, we inferred that their solution (according to our taxonomy) in fact supports the dynamic consensus voter instead of the majority one that has been specified by the authors (Gotze et al. 2008).

According to Buys et al. (2011), their solution, called A-NVP composite, supports majority voting. However, for similar reasons to the Gotze et al. classification we decided their solution instead supports a dynamic majority voter. Moreover, the response latency of the A-NVP composite is guaranteed not to exceed a maximum response time defined by the client. If no absolute majority could be established before the maximum response time, an exception will be issued to signal that consensus could not be found (Buys et al. 2011). In addition, according to Townend et al. (2005), their solution supports different types of voting algorithms to choose from. However, we inferred that the dynamic consensus voting is the only decision mechanism supported. The voter is dynamic as it can process 0 to n results, where n is the number of executed variant services. Specifically, the voting discards results of any variant service whose weighting falls below a user-defined value and subsequently performs consensus voting on the remaining results. The weighting is based on the confidence of that service returning a correct result.

Acceptance tests Acceptance tests give the client the freedom to specify accepted values according to pre-defined requirements. Nascimento et al. (2011) propose a solution that supports a mechanism responsible for the error-processing technique, which in turn supports acceptance tests, voters and comparisons. The solution by Dillen et al. (2012) has been designed so that acceptance tests are not hardwired within the FT-composition. Instead, ATs can be configured at runtime through a parameterised assertion holding an

XPath expression that will be used to assess if the variant result is acceptable The solution by Mansour and Dillon (2011) propose a scheme to combine variant web services into parallel and serial configurations with centralized coordination. In this case, the broker has an acceptance testing mechanism that examines the results returned from a particular web service. The acceptance test is conducted using the broker, which might be a single point of failure. To increase the reliability of the broker introduced in their systems and mask out errors at the broker level they suggest a modified general scheme based on triple modular redundancy and N-version programming, which also includes a voting algorithm. The ATs could be specified as examining a post-condition or inalterable association with the service. The solution by Zheng and Lyu (2008,2010b,2010a) and by Chen and Romanovsky (2008) supports recovery block strategy (RB), nevertheless, they do not mention any acceptance tests, the adjudicator employed in RB.

Other quality requirements The solution by Gorbenko et al. (2005) supports an adjudicator, whose type is not explicitly specified, that is also responsible for reconfiguration, recovery of the failed releases and for logging the information which may be needed for further analysis (Gorbenko et al. 2005). Extensibility is an important feature, and is important in SOA architecture when the context and available services are constantly changing. In the solution by Kotonya and Stephen (2010) different adjudicators might be plugged into their solutions. Also, new web services may be discovered and combined with the existing set of services. A particularly robust protocol detailed in this paper, Andros, provides a three-step consensus and authentication solution to tolerate Byzantine faults at a trade-off with system resources.

Chen and Romanovsky (2008) claim that although N-Version programming techniques require voting on results, in a real world application, it is not always possible to vote on results received from different services. In this sense, their solution for fault-tolerant SOAs supports well defined extension-points in which voting implementation might be included. The solution by Gonćalves and Rubira (2010) encapsulates the WS-Mediator proposed by Chen and Romanovsky (2008), therefore, it is possible to include voting implementations at extension-points of the WS-Mediator.

4.2.6 SOA specific requirements

The following discusses evidence that the target solution support specific SOA related quality requirements. The quality requirements selected include (i) interoperability capabilities, (ii) autonomic composition of services, (iii) QoS-aware service compositions and (iv) business-driven automated composition.

Since the studied solutions touch different phases of design, their implementation descriptions are not always available. For instance, some proposed solutions were based on abstract models or early stages of development and therefore it may not be appropriate to judge solutions against each other. Some solutions do not aim to present complete implementations as the contribution of the work is on a specific part of the system design. From the 17 primary studies, 7 showed support for all SOA-specific requirements as displayed in Additional file 1: Supplementary Table, 11 showed support for interoperability within their designs, 8 showed support for within their implementations for autonomic composition of services, 10 showed support for QoS-aware service compositions and 9 showed support for business-driven automated composition.

Solutions supporting all SOA specific requirements Seven solutions support all SOA specific requirements (Buys et al. 2011; Dillen et al. 2012; Gonçalves and Rubira 2010; Kotonya and Hall 2010; Nascimento et al. 2011; Santos et al. 2005; Townend et al. 2005; Zheng and Lyu 2010a).

The solution by Buys et al. (2011) is interoperable since it has been explicitly designed as a generic WSDM-enabled utility WS-Resource so as to support a diversity of applications without the need to generate application-specific proxy classes at design time. WSDM is a standard that defines how networked resources can be managed by exposing their capabilities and properties by means of a web service interface. This solution also supports an adaptive fault-tolerant strategy that autonomously tunes the amount of redundancy or dynamically alters the selection of variant services currently employed in the redundancy scheme. That is, their solution supports autonomic composition of services. Moreover, the proposed solution aims to achieve an optimal trade-off between dependability as well as performance-related objectives such as load balancing and timeliness.

Also, application-specific intricacies are taken into account, in that the redundancy dimensioning and variant selection models can be configured by means of a set of user-defined parameters. Therefore, the solutions supports both QoS-aware and business-driven automated compositions.

The FT-composition proposed by Townend et al. (2005) invokes variant services, and results are weighted based on a confidence metric for each service. The weighting algorithms, which are performed dynamically, take into account whether variant services are composed by common shared services and historic data of how often variant outputs agrees with the consensus. Services whose weighting is lower than a user-defined level are eliminated from the voting procedure for the remaining variants. Variant service end-points are specified by the client. This means the solution is not able to dynamically bind new variants. Moreover, the functionalities of the proposed FT-compositions are exposed as web services operations. Therefore, we infer that this solution supports interoperable, QoS-aware, business-driven automated and autonomic compositions.

The solution proposed by Dillen et al (2012) dynamically manages the degree of redundancy of multi-version fault-tolerance mechanisms. The client specifies the parameters of the dependability requirements and its available budget at request level. The system will autonomously select an appropriate degree and selection of variants and integrate them within an appropriate fault-tolerant redundancy scheme. Their solution is also responsible for maintaining a pool of variants of a specific web service and is capable of autonomously deploying additional variants, or removing poorly performing variants.

Therefore, their solution supports all described SOA specific requirements.

Nascimento et al. (2011) proposes a feature model that captures the variabilities and commonalities among software fault tolerance strategies (for instance, recovery blocks, N-version programming and N-self-checking programming). The identified variability is mapped into design decisions represented as variation points into a Product Line Architecture. The authors propose an infrastructure that encompasses both the feature model and PLA and relies on key activities of the autonomic control loop (i.e. collect, analyze, plan and execute) to support dynamic management of software variability. Based on changes of *(i)* user requirements and *(ii)* QoS level, and high-level policies, which are represented as adaptation rules, the control loop decides an appropriate fault tolerance strategy to be executed. Consequently, this infrastructure is responsible for the

dependable mediation logic between service clients and redundant services. Moreover, since functionally equivalent services may appear with completely different function names, input parameters and return types, hence complicating the dynamic discovery of redundant services. To alleviate this difficulty, their solution relies on the Semantic Web (SW) in order to support dynamic provision of redundant services by describing them in terms of semantics. Also the infrastructure itself may be deployed to be remotely accessible via web technology. Therefore, their solution supports all the described SOA specific requirements.

The FT-composition proposed by Santos et al (2005) relies on a set of components and services, some based on OMG's FT-CORBA standard's models and concepts. In order to create the groups of variant services, Santos et al (2005) adopts the service domain concept. A service domain allows aggregation and sharing of multiple web service descriptions (WSDL). The binding information refers to the group, allowing several services to be virtualized as a single service. Their solution is able to dynamically add new variant services and the remove faulty variant services according to predefined rules. In particular, if a variant service presents a fault at the moment of its execution or does not respond within the time limit established in the service configuration, the faulty variant is removed from the service group. In this case, the faulty variant service stays out of the group until its state has been reestablished through the recovery mechanisms. In order to carry out the monitoring and recovery process, variant services must implement predefined interfaces. Therefore, their solution supports all the described SOA specific requirements.

Zheng and Lyu. (2010a) propose an adaptive fault tolerance strategy for automatic system reconfiguration at runtime based on the subjective user requirements and objective QoS information of the target Web services. Users are encouraged to contribute their individually-obtained QoS information of the target Web services by employing user-participation and collaboration. In this way, this solution is able to collect a large quantity of QoS data from the users located in different geographical locations under various network conditions. This data is used to objectively evaluate the target Web services. This solution supports various software fault tolerance techniques and adjusts the optimal fault tolerance strategy based on the overall QoS information and the individually recorded QoS information of the variant services.

The framework by Kotonya and Stephen (2010) supports interoperability capabilities, autonomic composition of services, QoS-aware service composition and and business-driven automated composition. Their framework implements fault tolerance protocols as process models and exposes them as discoverable services. At runtime, the framework provides a service differentiation mechanism based on quality of services. In this way, their solution is able to dynamically instantiate fault tolerance techniques tailored to the specific needs of different clients and contexts (e.g. requirements in terms of QoS).

Solutions supporting partial SOA specific requirements The solution by Nourani et al. (2009, 2011) supports interoperability. Its variant execution scheme and adjudicators are offered as web services, and, in the prototype implementation, invoked in a fault-tolerant BPEL process that leverages variant web services.

The solution by Laranjeiro and Vieira (2007) support QoS-aware and business-driven automated service compositions as in their solution the web services execution sequence

is defined based on the information available for each variant. The variant services are ranked based on evaluation metrics (for instance, response time, availability and correctness of variants) as selected by the programmer. The evaluation metrics might be collected offline (before deployment of variants) and online (during utilization of variants). However, the QoS values are not used to select variant services, since service endpoints are hard-coded. Their solution for fault-tolerant SOAs also supports interoperability capabilities by exposing their main functionalities by means of a proxy web service.

The solution by Chen and Romanovsky (2008), called WS-Mediator, supports Web service resilience-explicit dynamic reconfiguration in order to adapt fault-tolerance techniques and use the available service redundancy. Their solution monitors web services and generates resilience metadata representing dependability attributes, such as response time, failure rate, failure types and so on (Yuhui and Romanovsky 2008). This metadata's structure can be designed for particular application scenarios and the resilience-explicit decision-making mechanism uses it to dynamically select the most dependable web service according to the client's preference. Nevertheless, the solution by Chen and Romanovsky (2008) cannot be directly accessible via web services technology because it is a Java stand-alone package. The solution by Gonćalves and Rubira (2010) extends WS-Mediator (Yuhui and Romanovsky 2008) in order to make it interoperable. Therefore, the solution by Chen and Romanovsky (2008) supports autonomic composition of services, QoS-aware service composition and business-driven automated composition, while the solution by Gonćalves and Rubira (2010) supports these quality requirements and also interoperability capabilities.

The solution by Mansour and Dillon (2011) supports both autonomic composition of services and QoS-aware service composition. The solution develops a model used to improve the reliability of the composite service based on the prediction of the failure rates of individual services. For instance, it allows for execution schemes to be optimized by choosing effective rollback schemes. The focus of the solution is therefore to support self-optimizing autonomic composition of services. The approach subsequently allows for both the increased dependability of the composite service and increase trust from the end-user.

4.3 Discussion

4.3.1 Guidelines for designing fault-tolerant service applications

In order to evaluate the guidelines presented in Section 4.1 we gathered qualitative feedback using a questionnaire. Five graduate developers of service-based applications were given the questionnaire with questions about certain qualitative aspects of the guidelines. Two of the volunteers had experience on software fault tolerance techniques while three had no previous experience on designing fault-tolerant software. The questionnaire had a total of nine questions: (i) one to collect information about the previous experience of the volunteers when developing service-based applications; (ii) one to collect information about the previous experience of the volunteers when developing fault-tolerant software; and (iii) seven questions to collect information about qualitative aspects of the guidelines.

The seven qualitative questions were: (1) *What are the positive aspects of the guidelines?*; (2) *What are the negative aspects of the guidelines?* (3) *Would you use it to develop your next fault-tolerant web service?* (4) *Why?* (5) *Would you recommend the use of the*

presented guidelines to develop fault-tolerant web service? (6) *Why?* (7) *Suggestions.* Except for Questions 3 and 5, which are yes/no questions, the other five qualitative questions had free answers.

All the volunteers agreed that the proposed guidelines make the design process more predictive and systematic. Moreover, according to the two volunteers that already had experience in designing fault-tolerant systems, the activities of the guidelines prevents important aspects of the software fault tolerance be neglected by the software designer. One of the volunteers suggested that the guidelines could be incorporated into a wizard tool to assist the designer in a more effective and easy to follow fashion.

Although the qualitative results are preliminary, they are important as a feedback and show a positive first impression that provides evidence that these guidelines are a potential help for less-experienced developers of fault-tolerant applications.

4.3.2 Systematic literature review

Related to the selection of variant services, we have considered two main issues, to select diverse variants and to determine an appropriate degree and/or selection of variant services. We should emphasize that these two different purposes are complementary. This is because the reliability of fault-tolerant compositions depends upon design diversity of their variant services to increase the probability that they fail on disjoint input spaces. For most of the proposed approaches for FT-compositions there is an underlying assumption that variant services can always be efficiently employed by means of diversity-based techniques (Gotze et al. 2008; Kotonya and Hall 2010; Zheng and Lyu 2008,2010b,2010a). However, Nascimento et al. (2012a) presents an empirical study to investigate whether variant services are able to tolerate software faults. They concluded that the benefits of diversity-based solutions applied to SOAs are not straightforward. Even when variants seem to present design diversity, this diversity might not be sufficient to improve system reliability in case the chosen variants have coincident faults activated on important execution scenarios. That is, the chosen set of variants will impact on the success of the FT-strategy used.

Runtime decisions regarding which variant services are used require trade-offs according to which specific QoS attributes to use, and their feasibility of obtaining them. There are important considerations for delegating QoS responsibility to different components of the architecture. It may be less process intensive to require each atomic service to provide quality measurements of themselves; however, lack of trust or need to ensure data integrity may mean that QoS is monitored from the client. For instance, many approaches measure availability by monitoring the variant service 'heartbeat' - this is certainly feasible with most available services. However, a particular challenge for fault tolerance is to find a feasible way of measuring other QoS attributes, for example, the security of potential services, an issue tackled by Gotze et al. (2008) where various levels of service transparency are taken into account. Moreover, while many solutions provide means of optimizing service selection once the variant services have been chosen, we see there is a growing need to integrate test activities to ensure a specific service as being suitable as a candidate variant service. This is particularly needed with growing capabilities of fault-tolerant SOA in terms of autonomic searching, discovering and selection of variant services.

With respect to the execution scheme, both parallel and sequential execution schemes have been addressed. This is particularly important, as the type of execution scheme

will affect important QoS attributes such as execution time and resource consumption (*Section 4.1*). Related to the decision mechanisms, it is important to notice that the expected behaviour of variant services is likely to affect the complexity of the chosen decision mechanism. For example, to choose among voting algorithms presented in the proposed taxonomy (Figure 4), we should consider the primary concerns surrounding the software's application and details about the output space (*Section 4.1*). As we can observe in the proposed classification (Additional file 1: Supplementary Table) there are still some adjudicators not addressed by current approaches for FT-compositions, e.g., median, mean voters and acceptance tests. One can claim that existing solutions might be easily extended.

On one hand, when looking beyond the implementation of solely the decision mechanism, we can also find interesting architectural solutions that provide additional functionality at the same point. For instance, by defining key extension points or ability for pluggable FT strategies enhances the flexibility and interoperability of fault-tolerant SOA design. Many publications that discuss this do not explicitly describe how to extend their solutions, therefore, the reader is unable to determine what interfaces must be implemented when inserting a custom adjudicator (Kotonya and Hall 2010; Laranjeiro and Vieira 2007; Nascimento et al. 2011; Yuhui and Romanovsky 2008). In fact adjudication procedures are marginally described. For example authors do not specify which type of variant outputs their solutions are able to process. Also, they don't specify how to navigate through elements and attributes in messages returned by the variant services in order to adjudicate the acceptability of specific fragments from these messages (Dillen et al. 2012; Kotonya and Hall 2010; Nascimento et al. 2011; Santos et al. 2005). In other words, most of the authors do not specify clear guidelines on the reuse and, in particular, customization of their decision mechanisms in practical settings.

In addition, related to the decision mechanism, many FT-protocols within diversity-based fault tolerance solutions frequently selected results based on properties other than the actual response values, such as response time or likelihood of failure. This perhaps reflects the reality that even when reliability of results is uncertain, the fastest response time remains one of the main sought-after service qualities. Alternatively, it highlights the need for future research to address these challenging issues. Finally, it is interesting to observe that although various design issues and respective design solutions related to software fault tolerance techniques have been supported, they are spread among existing approaches for FT-compositions. That is, there is no a single solution able to cope with conflicting client requirements by employing at the same time a wide variety of schemes to select and execute variant services and to determine the adjudicated result from the variant services. There is a lack of solutions able to bring out a set of closely related fault tolerance techniques based on design diversity in close accordance with customers' requirements and high-level policies (e.g. to adopt a fault tolerance technique based on parallel execution scheme for better response time).

The classification of the primary studies according to SOA-specific requirements in Additional file 1: Supplementary Table provides a useful starting point to understanding the capabilities of specific solutions. From the 17 primary studies, 7 showed support for all SOA-specific requirements. In fact, design decisions chosen to support one SOA-specific requirement often addressed the others at the same time. For instance, mechanisms that gather QoS data can be utilised to select appropriate services. Consequently, the

mechanisms to collect this information are often utilized to satisfy all requirements relating to the composition of services.

Many solutions provided QoS-aware service compositions. However, the way in which quality of service was measured may have been based on multiple properties such as response time, availability and correctness of variants or alternatively solely by monitoring the availability of a variant. As mentioned previously, these differences between solutions with respect to their underlying mechanisms for SOA-specific features impact on the capabilities of the SOA to satisfy the remaining design issues such as how to select, execute and judge variants.

Finally, we noticed that the field of defining fault-tolerant service compositions is relatively recent, since the analyzed solutions present illustrative examples to evaluate their feasibility. To the best of our knowledge there are no commercial or open source solutions for implementing fault-tolerant service composition based on design diversity.

5 Threats to validity

We identified some possible threats to validity (Wohlin et al. 2000) and the measures we took to reduce the risks.

5.1 Internal validity

In terms of internal validity, our study is based on 17 papers that matched our criteria (*Section 3.3.2*). This number is not high, nevertheless, it is representative of this area of research (Burrows et al. 2010; Kitchenham et al. 2009). To mitigate this risk, we adopted a search strategy that aims to detect as much of the relevant literature as possible (Kitchenham and Charters 2007). Despite this, the size of the sample should be kept in mind when assessing the generality of our results.

5.2 Construct validity

We identified two threats to construct validity: the study selection and data extraction are error-prone activities. Related to the study selection, this activity was performed by one of the researches at two different points in time, thus reducing the risk of having the inclusion/exclusion criteria applied inconsistently. Related to the data extraction, data might have been extracted in an inconsistent manner and to reduce this risk, as suggested by Kitchenham and Charters (2007), all primary studies were assigned to one of the researchers, responsible for extracting the data (Kitchenham and Charters 2007). Another researcher was asked to perform data extraction on a subset of primary studies chosen at random (for instance, on 6 studies). Data from the researches were compared and disagreements were resolved by consensus among researchers.

5.3 Conclusion validity

We identified one threat to conclusion validity, which is the reliability of the taxonomy itself used to classify the primary studies. To mitigate the risks of employing an inadequate taxonomy, before it was built, we had analysed the domain knowledge of software fault tolerance techniques based on design diversity in depth (e.g. (Elmendorf 1972; Horning et al. 1974; Kim 1984; Laprie et al. 1990; Lee and Anderson 1990; Lyu 1996; Pullum 2001; Scott et al. 1987)).

6 Related work

We are not familiar with any work that surveys diverse fault-tolerant SOAs. As a consequence, we address, in turn, work related to literature review of fault tolerance techniques in general.

Garcia et al. (2001) present a comparative study of exception handling mechanisms for building dependable object-oriented software. The authors define a taxonomy to help address main basic technical aspects for a given exception handling proposal. By means of the proposed taxonomy, the authors survey various exception mechanisms implemented in different object-oriented languages, evaluates and compares different designs. Our classification of software fault tolerance solutions is also based on a general taxonomy of design issues. However, compared to their work, we do not provide a rating of the primary studies according to a quality assessment.

Carzaniga et al (2008) identify some key dimensions upon which they define a taxonomy of fault tolerance and self-healing techniques in order to survey and compare the different ways redundancy has been exploited in the software domain. These are the intention of redundancy (deliberate or opportunistic), the type of redundancy (code, data, environment), the nature of triggers and adjudicators that can activate redundant mechanisms and use their results (preventive - implicit adjudicator or reactive or-implicit/explicit adjudicator), and lastly the class of faults addressed by the redundancy mechanisms (Bohrbugs or Heisenbugs). The proposed taxonomy is used to classify well known techniques, for example, N-version programming, exception handling and data diversity. The concepts presented in their taxonomy and the ones presented in the taxonomy we employed are orthogonal. In fact, our classification is performed in lower level of abstraction.

Ammar et al. (2000) propose a survey of the different aspects of system fault tolerance and discuss some issues that arise in hardware fault tolerance and software fault tolerance. In this context, the authors distinguish information, spatial and temporal redundancy; present the three fundamental concepts of fault tolerance (i.e. failure, error, fault); describe the four steps of fault tolerance (i.e. error detection, damage assessment, error recovery, and fault removal) and relate these to the differences of redundant techniques for handling hardware as well as software faults. According to the authors, since redundancy may be used under a variety of forms to achieve fault tolerance, the design of a fault-tolerant system involves a set of trade-offs between redundancy requirements (imposed by the need for fault tolerance) and requirements of economy (economy of the process, and the product) (Ammar et al. 2000). The authors also emphasize that program fault tolerance is no panacea, like almost everything in software engineering. We refer to their work for an interesting discussion on reasons to support this claim.

Florio and Blonda (2008) present a survey of linguistic structures for application-level fault tolerance (ALFT). The authors emphasize the importance of employing appropriate structuring techniques to support an adequate separation between the functional and fault tolerance concerns. They claim the design choice of which fault tolerance provisions to support can be conditioned by the adequacy of the syntactical structure at 'hosting' the various provisions, called *syntactical adequacy*. Moreover, offline and online (dynamic) management of fault tolerance provisions and their parameters may be an important requirement for managing the fault-tolerant code in an ALFT, called *adaptability*. These three properties, separation of concerns, adaptability and syntactical

adequacy are referred as the *structural attributes* of ALFT. The structural attributes are adopted to classify and analyse a number of ALFTs, including, recovery blocks and n-version programming. This classification is also orthogonal to the classification we have provided.

7 Conclusion

Due to the low cost of reusing existing functionally equivalent services, called variant services, several approaches based on design diversity exist to support fault-tolerant Service-Oriented Architecture (SOA). These solutions operate in the communication between clients and variant services, which are structured in fault-tolerant compositions. Regarding fault tolerance based on software diversity, three major design issues need to be considered, namely, selection of variants; variant execution schemes; and judgement on results acceptability. These design issues may be realised by different design solutions. Different design decisions involve different measures of quality requirements (e.g. memory utilisation, execution time, reliability, financial costs and availability). With respect to service-oriented computing, it is well known that SOA systems *(i)* exhibit highly dynamic characteristics, and changes in the quality of services are likely to occur frequently and *(ii)* should support conflicting user requirements. Therefore, effective SOA solutions should address these design challenges.

In this paper we define a general taxonomy for these design challenges, their solutions, and functionalities required for the aggregation of multiple services into a single composite service. We also compare current design solutions and give an overview of their effectiveness. Based on this information and by means of systematic literature review method, we present a comprehensive survey of existing solutions for fault-tolerant service composition. The solutions were classified according to the elements of the proposed taxonomy and the checklist they support, thus facilitating the process of choosing from existing solutions by a SOA architect, according to specific needs of each SOA-application and execution environment. The classification guidelines support decisions related to four important aspects of fault-tolerant SOA-based applications: *(i)* failure mode, by defining the ways a failure can occur; *(ii)* fault latency, by defining details related to the failure behaviour, such as the interval between the moments of fault occurrences; *(iii)* fault assumptions, guiding the architect on the analysis of error detection and recovery mechanisms and the identification of restrictions related to the services available; and *(iv)* comparison amongst existing design solutions, by means of a systematic literature review which provides valuable information about the characteristics of classical software fault tolerance techniques.

Additional file

Additional file 1: Supplementary Table S1. Classification of the Primary Studies.

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
ASN and RB planned the systematic review, carried it out and analyzed its results. ASN, RB and FC analyzed threats to the validity of the performed systematic review and the measures we took to reduce the risks. PHSB and ASN specified the design decision guidelines to realize fault tolerance in Service-Oriented Architecture (SOA). The format of the manuscript

was decided by PHSB, ASN and RB. The manuscript was prepared by PHSB, corrections and reviews are made by RB, FC and CMFR. All authors read and approved the final manuscript.

Acknowledgements
We would like to thank the anonymous referees, who helped to improve this paper. This research was partially supported by UOL (www.uol.com.br), through its UOL Bolsa Pesquisa program, process number 20120217172801. Cecília is supported by CNPq (305331/2009-4) and FAPESP (2010/00628-1). Fernando is supported by CNPq/Brazil (306619/2011-3), FACEPE/Brazil (APQ-0395-1.03/10 and APQ-1359-1.03/12), and by INES (CNPq 573964/2008-4 and FACEPE APQ-1037-1.03/08). Rachel is supported by EPSRC grant reference EP/K002465/1.

Author details
[1]Institute of Exact Sciences and Biology, Federal University of Ouro Preto, Ouro Preto, MG, Brazil. [2]Institute of Computing, University of Campinas, Campinas, SP, Brazil. [3]Department of Computer Science, University of Bath, Bath, UK. [4]Informatics Center, Federal University of Pernambuco, Recife, PE, Brazil. [5]Institute of Computing, Federal University of Alagoas, Maceió, AL, Brazil.

Received: 11 December 2013 Accepted: 11 October 2014
Published online: 14 November 2014

References

Abdeldjelil H, Faci N, Maamar Z, Benslimane D (2012) A diversity-based approach for managing faults in web services. In: Proceedings of the IEEE 26th International Conference on Advanced Information Networking and Applications. IEEE Computer Society, Los Alamitos, CA, USA. pp 81–88

Ammar HH, Cukic B, Mili A, Fuhrman C (2000) A comparative analysis of hardware and software fault tolerance: impact on software reliability engineering. Ann Software Eng 10(1–4):103–150

Anderson T, Barrett PA, Halliwell DN, Moulding MR (1985) Software fault tolerance: An evaluation. IEEE Trans Software Eng SE-11(12):1502–1510

Arlat J, Kanoun K, Laprie JC (1988) Dependability evaluation of software fault-tolerance. In: Digest of Papers of the 18th International Symposium on Fault-Tolerant Computing (FTCS'18). Society, Washington, DC, USA. pp 142–177

Avizienis A, Laprie JC, Randell B, Landwehr C (2004) Basic concepts and taxonomy of dependable and secure computing. IEEE Trans Dependable Secure Comput 1(1):11–33

Blough DM, Sullivan GF (1990) A comparison of voting strategies for fault-tolerant distributed systems. In: Proceedings of the 9th Symposium Reliable Distributed Systems (SRDS'09). IEEE Computer Society, Washington, DC, USA. pp 136–145

Broen RB (1975) New voters for redundant systems. J Dyn Syst Meas Contr 97(1):41–45

Burrows R, Garcia A, Taiani F (2010) Coupling metrics for aspect-oriented programming: A systematic review of maintainability studies. In: Maciaszek L, Gonzalez-Perez C, Jablonski S (eds). Evaluation of Novel Approaches to Software Engineering. Springer, Berlin. pp 277–290

Buys J, De Florio V, Blondia C (2011) Towards context-aware adaptive fault tolerance in soa applications. In: Proceedings of the 5th ACM International Conference on Distributed Event-Based System (DEBS'11). ACM, New York, NY, USA. pp 63–74

Cardozo ESF, Araújo Neto JBF, Barza A, França ACC, da Silva FQB (2010) Scrum and productivity in software projects: a systematic literature review. In: Proceedings of the 14th International Conference on Evaluation and Assessment in Software Engineering (EASE'10). British Computer Society, Swinton, UK, UK. pp 131–134

Carzaniga A, Gorla A, Pezzè M (2008) Handling software faults with redundancy. In: de Lemos R, Fabre JC, Gacek C, Gadducci F, ter Beek MH (eds). WADS, Lecture Notes in Computer Science, vol. 583. Springer, Berlin. pp 148–171

Daniels F, Kim K, Vouk MA (1997) The reliable hybrid pattern: a generalized software fault tolerant design pattern. In: Proceedings of the 4th Conference of Patter Languages of Programming Conference (PloP'97). Washington University, St. Louis, MO, USA. pp 1–9

Di Giandomenico F, Strigini L (1990) Adjudicators for diverse-redundant components. In: Proceedings of the 9th Symposium on Reliability in Distributed Software and Database Systems (SRDS'90). IEEE Computer Society, Washington, DC, USA. pp 114–123

Dillen R, Buys J, Florio V, Blondia C (2012) Wsdm-enabled autonomic augmentation of classical multi-version software fault-tolerance mechanisms. In: Ortmeier F, Daniel P (eds). Computer Safety, Reliability, and Security. Springer, Berlin. pp 294–306

Eckhardt DE, Lee LD (1985) A theoretical basis for the analysis of multiversion software subject to coincident errors. IEEE Trans Software Eng SE-11(12):1511–1517

Eckhardt DE, Caglayan AK, Knight JC, Lee LD, McAllister DF, Vouk MA, Kelly JPJ (1991) An experimental evaluation of software redundancy as a strategy for improving reliability. IEEE Trans Software Eng 17(7):692–702

Elmendorf WR (1972) Fault-tolerant programming. In: Proceedings of the 2nd IEEE International Symposium on Fault Tolerant Computing (FTCS'2). pp 79–83

Faci N, Abdeldjelil H, Maamar Z, Benslimane D (2011) Using diversity to design and deploy fault tolerant web services. In: Proceedings of the 20th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'11). IEEE Computer Society, Washington, DC, USA. pp 73–78

Florio VD, Blondia C (2008) A survey of linguistic structures for application-level fault tolerance. ACM Comput Surv 40(2):6:1–6:37

Garcia AF, Rubira CMF, Romanovsky AB, Xu J (2001) A comparative study of exception handling mechanisms for building dependable object-oriented software. J Syst Software 59(2):197–222

Gärtner FC (1999) Fundamentals of fault-tolerant distributed computing in asynchronous environments. ACM Comput Surv 31(1):1–26

Gonçalves EM, Rubira CMF (2010) Archmeds: an infrastructure for dependable service-oriented architectures. In: Proceedings of the 17th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'10). IEEE Computer Society, Washington, DC, USA. pp 371–378

Gorbenko A, Kharchenko V, Popov P, Romanovsky A (2005) Dependable composite web services with components upgraded online. In: Lemos R, Gacek C, Romanovsky A (eds). Architecting Dependable Systems III. Springer-Verlag, Berlin. pp 92–121

Gorbenko A, Kharchenko V, Romanovsky A (2009) Using inherent service redundancy and diversity to ensure web services dependability. In: Butle M, Jones C, Romanovsky A, Troubitsyna E (eds). Methods, Models and Tools for Fault Tolerance. Springer, Berlin. pp 324–341

Gorbenko A, Romanovsky A, Kharchenko V, Tarasyuk O (2012) Dependability of service-oriented computing: time-probabilistic failure modelling. In: Avgeriou P (ed). Software Engineering for Resilient Systems. Springer, Berlin. pp 121–133

Gotze J, Muller J, Muller P (2008) Iterative service orchestration based on dependability attributes. In: Proceedings of the 34th Euromicro Conference on Software Engineering and Advanced Applications (SEAA'08). IEEE Computer Society, Washington, DC, USA. pp 353–360

Hilford V, Lyu MR, Cukic B, Jamoussi A, Bastani FB (1997) Diversity in the software development process. In: Proceedings of the 3rd Workshop on Object-Oriented Real-Time Dependable Systems (WORDS,97). IEEE Computer Society, Washington, DC, USA. pp 129–136

Horning JJ, Lauer HC, Melliar-Smith PM, Randell B (1974) A program structure for error detection and recovery. In: Proceedings of an International Symposium on Operating Systems: Theoretical and Practical Aspects. Springer, London, UK. pp 171–187

Huhns MN, Singh MP (2005) Service-oriented computing: key concepts and principles. IEEE Internet Comput 9(1):75–81

Jorgensen M, Shepperd M (2007) A systematic review of software development cost estimation studies. IEEE Trans Software Eng 33(1):33–53

Kim KH (1984) Distributed execution of recovery blocks: An approach to uniform treatment of hardware and software faults. In: Proceedings of 4th the International Conference on Distributed Computing Systems (ICDSC'84). IEEE Computer Society, Washington, DC, USA. pp 526–532

Kitchenham B, Charters S (2007) Guidelines for performing systematic literature reviews in software engineering. Tech. Rep. Technical Report EBSE 2007-001. Department of Computer Science, University of Durham

Kitchenham BA, Mendes E, Travassos GH (2007) Cross versus within-company cost estimation studies: a systematic review. IEEE Trans Software Eng 33(5):316–329

Kitchenham BA, Pearl Brereton O, Budgen D, Turner M, Bailey J, Linkman S (2009) Systematic literature reviews in software engineering - a systematic literature review. Inform Software Tech 51(1):7–15

Knight JC, Leveson NG (1986) An experimental evaluation of the assumption of independence in multiversion programming. IEEE Trans Software Eng 12(1):96–109

Kotonya G, Hall S (2010) A differentiation-aware fault-tolerant framework for web services. In: Maglio PP, Weske M, Yang J, Fantinato M (eds). Service-Oriented Computing. Springer, Berlin. pp 137–151

Laprie JC, Béounes C, Kanoun K (1990) Definition and analysis of hardware and software-fault-tolerant architectures. Computer 23(7):39–51

Laranjeiro N, Vieira M (2007) Towards fault tolerance in web services compositions. In: Proceedings of the 2nd International Workshop on Engineering Fault Tolerant Systems (EFTS'07). ACM, New York, NY, USA

Lee PA, Anderson T (1990) Fault tolerance: principles and practice. 2nd edn. Springer-Verlag New York, Inc., Secaucus

Looker N, Munro M, Xu J (2005) Increasing web service dependability through consensus voting. In: Proceedings of the 29th annual International Conference on Computer Software and Applications (COMPSAC-W'05). IEEE Computer Society, Washington, DC, USA. pp 66–69

Lyu MR (1996) Handbook of Software Reliability Engineering. Inc., Hightstown

Lyu MR, Chen JH, Avizienis A (1994) Experience in metrics and measurements of n-version programming. Int J Reliab Qual Saf Eng 1(1):41–62

Mansour H, Dillon T (2011) Dependability and rollback recovery for composite web services. IEEE Trans Serv Comput 4(4):328–339

McAllister DF, Vouk MA (1996) Handbook of software reliability engineering. McGraw-Hill, Inc., Hightstown. pp 567–614. chap Fault-tolerant Software Reliability Engineering, http://dl.acm.org/citation.cfm?id=.239425239466

Milanovic N, Malek M (2007) Service-oriented operating system: A key element in improving service availability. In: Proceedings of the 4th International Symposium on Service Availability (ISAS '07). Springer, Berlin. pp 31–42

Nascimento AS, Rubira CMF, Lee J (2011) An spl approach for adaptive fault tolerance in soa. In: Proceedings of the 15th International Software Product Line Conference (SPLC'11). pp 1–8

Nascimento AS, Castor F, Rubira CMF, Burrows R (2012a) An empirical study on design diversity of functionally equivalent web services. In: Proceedings of the 7th International Conference on Availability, Reliability and Security (ARES'12). ACM, New York, NY, USA. pp 236–241

Nascimento AS, Castor F, Rubira CMF, Burrows R (2012b) An experimental setup to assess design diversity of functionally equivalent services. In: Proceedings of the 16th International Conference on Evaluation and Assessment in Software Engineering (EASE'12). IET, Herts, UK. pp 177–186

Nascimento AS, Rubira CMF, Burrows R, Castor F (2013) A model-driven infrastructure for developing product line architectures using cvl. In: Proceedings of the 7th International Conferences on Self-Adaptative and Self-Organizing Systems (SBCARS'13). IEEE Computer Society, Washington, DC, USA

Nourani E (2011) A new architecture for dependable web services using n-version programming. In: Proceedings of 3rd International Conference on Computer Research and Development (ICCRD'11). IEEE Computer Society, Washington, DC, USA. pp 333–336

Nourani E, Azgomi MA (2009) A design pattern for dependable web services using design diversity techniques and ws-bpel. In: Proceedings of the 6th International Conference on Innovations in Information Technology (IIT'09). pp 290–294

Papazoglou MP, Heuvel WJ (2007) Service oriented architectures: approaches, technologies and research issues. Int J Very Large Data Bases 16(3):389–415
Papazoglou MP, Traverso P, Dustdar S, Leymann F (2006) Service-oriented computing research roadmap. In: Dagstuhl Seminar Proceedings 05462. Universidad de Talca, Talca. pp 1–29
Papazoglou MP, Traverso P, Dustdar S, Leymann F (2007) Service-oriented computing: state of the art and research challenges. Computer 40(11):38–45
Pullum LL (2001) Software fault tolerance techniques and implementation. Artech House, Inc., Norwood
Randell B (1975) System structure for software fault tolerance. In: Proceedings of the 1st International Conference on Reliable Software. pp 437–449
Saglietti F (1992) The impact of voter granularity in fault-tolerant software on system reliability and availability. In: Kersken M, Saglietti F (eds). Software Fault Tolerance. Springer-Verlag, Berlin. pp 199–212
Santos GT, Lung LC, Montez C (2005) Ftweb: a fault tolerant infrastructure for web services. In: Proceedings of the 9th IEEE International EDOC Enterprise Computing Conference (EDOC '05). IEEE Computer Society, Washington, DC, USA. pp 95–105
Scott RK, Gault JW, Mcallister DF (1987) Fault-tolerant software reliability modeling. IEEE Trans Software Eng SE 13(5):582–592
Shin K, Lee YH (1984) Error detection process: model, design, and its impact on computer performance. IEEE Trans Comput 33(6):529–540
Townend P, Groth P, Xu J (2005) A provenance-aware weighted fault tolerance scheme for service-based applications. In: Proceedings of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05). IEEE Computer Society, Washington, DC, USA. pp 258–266
Trivedi KS, Grottke M, Andrade E (2010) Software fault mitigation and availability assurance techniques. Int J Syst Assur Eng Manage 1(4):340–350
Vouk MA, Mcallister DF, Eckhardt DE, Kim K (1993) An empirical evaluation of consensus voting and consensus recovery block reliability in the presence of failure correlation. J Comput Software Eng 1(10):364–388
Wilfredo T (2000) Software fault tolerance: a tutorial. Tech. Rep. Technical Report NASA/TM-2000-210616, National Aeronautics and Space Administration (NASA)
Williams BJ, Carver JC (2010) Characterizing software architecture changes: A systematic review. Inform Software Tech 52(1):31–51
Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2000) Experimentation in Software Engineering: An Introduction. Kluwer Academic Publishers, Norwell
Xu J (2011) Achieving dependability in service-oriented systems. In: Jones CB, Lloyd JL (eds). Dependable and Historic Computing. Springer, Berlin. pp 504–522
Yuhui C, Romanovsky A (2008) Improving the dependability of web services integration. IT Professional 10(3):29–35
Zheng Z, Lyu MR (2008) Ws-dream: a distributed reliability assessment mechanism for web services. In: Proceedings of the International Conference on Dependable Systems and Networks. IEEE Computer Society, Washington, DC, USA. pp 392–397
Zheng Z, Lyu MR (2010a) An adaptive qos-aware fault tolerance strategy for web services. Empir Software Eng 15(4):323–345
Zheng Z, Lyu MR (2010b) Collaborative reliability prediction of service-oriented systems. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10). pp 35–44

doi:10.1186/s40411-014-0013-7
Cite this article as: Nascimento *et al.*: **Designing fault-tolerant SOA based on design diversity.** *Journal of Software Engineering Research and Development* 2014 **2**:13.

Submit your manuscript to a SpringerOpen journal and benefit from:

► Convenient online submission
► Rigorous peer review
► Immediate publication on acceptance
► Open access: articles freely available online
► High visibility within the field
► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com