



COMPUTING SCIENCE

Architecting Fault Tolerant Systems

Henry Muccini and Alexander Romanovsky

TECHNICAL REPORT SERIES

No. CS-TR-1343

July 2012

Architecting Fault Tolerant Systems

H. Muccini and A. Romanovsky

Abstract

Building trustworthy (dependable) systems is a major challenge faced by software developers. To this end, various fault tolerance mechanisms have been developed by researchers and used in industry. Unfortunately, more often than not these solutions ignore earlier development phases - most importantly, the architecture design to exclusively focus on the implementation instead. This creates a dangerous gap between the requirement to build dependable (and fault tolerant) systems and the failure to address these issues at any stage preceding the implementation step. Software Architecture has been widely accepted as a way to achieve a better software quality while reducing the time and cost of production. While typical architectural specifications model only the normal behaviour of the system, ignoring the abnormal ones, several approaches have recently been developed which break the wrong pattern.

The aim of this paper is to survey the existing approaches to architecting fault tolerant systems, offering its readers a clear picture of the state of the art research in this emerging area. This survey is built on developing a two-dimensional classification of the existing solutions: the first dimension is based on the traditional software engineering characteristics while the second one uses fault tolerance-related parameters. The paper analyses the major trends and identifies possible directions for future research.

Bibliographical details

MUCCINI, H., ROMANOVSKY, A.

Architecting Fault Tolerant Systems
[By] H. Muccini, A. Romanovsky

Newcastle upon Tyne: Newcastle University: Computing Science, 2012.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1343)

Added entries

NEWCASTLE UNIVERSITY

Computing Science. Technical Report Series. CS-TR-1343

Abstract

Building trustworthy (dependable) systems is a major challenge faced by software developers. To this end, various fault tolerance mechanisms have been developed by researchers and used in industry. Unfortunately, more often than not these solutions ignore earlier development phases - most importantly, the architecture design to exclusively focus on the implementation instead. This creates a dangerous gap between the requirement to build dependable (and fault tolerant) systems and the failure to address these issues at any stage preceding the implementation step. Software Architecture has been widely accepted as a way to achieve a better software quality while reducing the time and cost of production. While typical architectural specifications model only the normal behaviour of the system, ignoring the abnormal ones, several approaches have recently been developed which break the wrong pattern.

The aim of this paper is to survey the existing approaches to architecting fault tolerant systems, offering its readers a clear picture of the state of the art research in this emerging area. This survey is built on developing a two-dimensional classification of the existing solutions: the first dimension is based on the traditional software engineering characteristics while the second one uses fault tolerance-related parameters. The paper analyses the major trends and identifies possible directions for future research.

About the authors

Dr Henry Muccini is an Assistant Professor with Dipartimento di Informatica, University of L'Aquila, Italy. Henry works on software engineering focusing on architectural languages, architecture-based analysis and architecting wireless sensor networks. More information about henry can be found here web site is www.henrymuccini.com.

Alexander (Sascha) Romanovsky is a Professor in the Centre for Software and Reliability, Newcastle University. His main research interests are system dependability, fault tolerance, software architectures, exception handling, error recovery, system structuring and verification of fault tolerance. He received a M.Sc. degree in Applied Mathematics from Moscow State University and a PhD degree in Computer Science from St. Petersburg State Technical University. He was with this University from 1984 until 1996, doing research and teaching. In 1991 he worked as a visiting researcher at ABB Ltd Computer Architecture Lab Research Center, Switzerland. In 1993 he was a visiting fellow at Istituto di Elaborazione della Informazione, CNR, Pisa, Italy. In 1993-94 he was a post-doctoral fellow with the Department of Computing Science, the University of Newcastle upon Tyne. In 1992-1998 he was involved in the Predictably Dependable Computing Systems (PDCS) ESPRIT Basic Research Action and the Design for Validation (DeVa) ESPRIT Basic Project. In 1998-2000 he worked on the Diversity in Safety Critical Software (DISCS) EPSRC/UK Project. Prof Romanovsky was a co-author of the Diversity with Off-The-Shelf Components (DOTS) EPSRC/UK Project and was involved in this project in 2001-2004. In 2000-2003 he was in the executive board of Dependable Systems of Systems (DSoS) IST Project. He has been the Coordinator of the Rigorous Open Development Environment for Complex Systems (RODIN) IST Project (2004-2007). He is now the Coordinator of the major FP7 DEPLOY Integrated Project (2008-2012) on Industrial Deployment of System Engineering Methods Providing High Dependability and Productivity.

Suggested keywords

DEPENDABILITY
SOFTWARE ARCHITECTURES
FAULT TOLERANCE

Architecting Fault Tolerant Systems

Henry Muccini and Alexander Romanovsky

Abstract

Building trustworthy (dependable) systems is a major challenge faced by software developers. To this end, various fault tolerance mechanisms have been developed by researchers and used in industry. Unfortunately, more often than not these solutions ignore earlier development phases - most importantly, the architecture design to exclusively focus on the implementation instead. This creates a dangerous gap between the requirement to build dependable (and fault tolerant) systems and the failure to address these issues at any stage preceding the implementation step.

Software Architecture has been widely accepted as a way to achieve a better software quality while reducing the time and cost of production. While typical architectural specifications model only the normal behaviour of the system, ignoring the abnormal ones, several approaches have recently been developed which break the wrong pattern.

The aim of this paper is to survey the existing approaches to architecting fault tolerant systems, offering its readers a clear picture of the state of the art research in this emerging area. This survey is built on developing a two-dimensional classification of the existing solutions: the first dimension is based on the traditional software engineering characteristics while the second one uses fault tolerance-related parameters. The paper analyses the major trends and identifies possible directions for future research.

Index Terms

Software Architecture, Fault Tolerance

I. INTRODUCTION

While it is no longer acceptable to argue against the need to incorporate fault-tolerance means into software, it is only recently that software engineering of fault tolerant systems has become an active research area recognised by both contributing communities, software engineering and dependability (see, for example, recent workshops on Architecting Dependable Systems [1], Engineering Fault Tolerant Systems [2], and Engineering Resilient Systems [3]). For a very long time, each community treated the other as something on the fringes. More often than not, researchers and developers working on software engineering left the issues of dealing with faults until the

H. Muccini is with the Department of Computer Science, University of L'Aquila, Italy. E-mail: henry.muccini@univaq.it

A. Romanovsky is with the Center for Software Reliability, Newcastle University, Newcastle upon Tyne - UK E-mail: alexander.romanovsky@ncl.ac.uk

very end of development, focusing on ensuring the normative system behaviour. Similarly, developers of advanced fault tolerance schemes did not tend to propose solutions for engineering fault tolerance starting from the earlier development phases. Arguably, this has been one of the reasons for a considerable number of failures caused by malfunctioning fault tolerance means.

Let us consider several examples of *poorly engineered fault tolerance*:

- according to Flaviu Cristian's report of field experience, up to two thirds of system failures in telephone switching systems used in the 80s were due to design faults in exception handling or recovery algorithms [4];
- the failure of the Ariane 5 launcher was caused by improper reuse of component exception handlers [5];
- the Interim Report on Causes of the August 14th 2003 Blackout in the US and Canada clearly shows that the problem was mostly caused by badly designed fault tolerance, including various architectural issues: poor diagnostics of component failures, longer-than-estimated time for component recovery, failure to involve all necessary components in recovery, inconsistent system state after recovery, failures of alarm systems, etc. [6];
- in their report of typical patterns of exception handling misuse and abuse in five customer and one proprietary J2EE applications, IBM researchers refer to them as "bad coding practice". It was found, for example, that one in ten classes swallows exceptions without doing anything about them [7];
- in a frequently cited paper, the authors conducted an experiment that shows that in a 10 million LOC real-time embedded control system, misused exception handling results in the introduction of 2-3 bugs per 1 KLOS [8];
- it has been recently reported that in the eight .NET assemblies (which represent the application, library and infrastructure levels), over 90% of exceptions that can be thrown by the code are not documented [9].
- It has been shown [10] that exception handling design in industrial applications typically exhibits poor quality independently of the underlying programming language or the application domain.

It is true that advances made since the 70s have included a plethora of fault tolerance mechanisms, a good understanding of the basic principles of building fault tolerant software, and the dedication of a considerable fraction of requirements analysis, run-time resources, development efforts and code to ensuring fault tolerance. And yet we still cannot say that fault tolerance is always trustworthy. It tends to be the least understood, documented or tested part of the system, which is poorly designed, misused and left until too late in the development process, seldom introduced in a systematic, disciplined or rigorous way, and often not suitable for the specific situations in which it is applied (in his famous paper on exception handling [4] F. Cristian draws this conclusion regarding the exception handling code, but we believe this is true for fault tolerance in general).

Recently, however, a number of studies have been conducted aiming to understand *where and how fault tolerance can be integrated in the software life-cycle* (e.g., [11], [12], [13]). It has been recognised that different classes of faults, errors and failures are identifiable during different phases of software development, and that it is therefore essential that fault tolerance is addressed at different phases of the software process, such as requirements, high-level (architectural) and low-level design.

The emerging research area which specifically focuses on **architecting fault tolerant systems** has, in the last few years, gained recognition among both academia and industry. This is due to the fact that the introduction of

fault tolerance at the architecting phase has the clear benefits of allowing developers to make good decisions, very early into the process, about what redundant resources to use, how to use them efficiently and to establish if the chosen recovery strategy will be successful. Making fault tolerance an explicit concern addressed during this phase, however, raises a number of challenging issues which the researchers working on architecting fault tolerant systems need to address. Some initial overviews of the state of the art in this area are reported in [14].

We believe it is the right time to **systematically analyse this area and to compare and contrast the existing work**. Coming as we do from different backgrounds (software architecture and fault tolerance), we have found it extremely fruitful and instructive to work together on this survey, trying to bring knowledge and traditions from both domains into it. The core of this work is a two-dimensional classification based on parameters which originate in both software architecture and fault tolerance. The survey is set to achieve two chief aims: first, to analyse a carefully selected set of approaches using these parameters and, secondly, to describe the major contributions and trends in terms of these parameters.

The paper is organized as follows. First, some suggestions are made as to how different readers' needs could be met in Section I-A. A brief outline of the fault tolerance and software architecture concepts is given in Section II. The procedure for study selection is described in Section III, the classification framework is presented in Section IV, while the application of the framework to the selected papers is introduced in Sections V and VI. Section VII provides the final analysis, summarising the lessons learned and outlining the directions of future work in the area. Section VIII concludes the survey with a brief summary.

A. Suggestions for Readers of this Survey

Sections II and IV will provide sufficient explanation of the main bulk of this paper for readers from backgrounds other than the software engineering or dependability communities. If the reader comes from one of the two communities, he/she may wish to read Section II more closely to familiarise him/herself with the basic concepts related to the work of the other community, and Section IV for a detailed explanation of the meaning of each parameter. Sections V and VI contain the main results of this study.

This survey will help practitioners to find specific solutions to the problem of architecting fault tolerant systems. Their work should start with the definition of the fault tolerant requirements. At the next step, software architects (working with the specification of a fault tolerant architecture which addresses these requirements) and the dependability experts (asked to find an appropriate fault tolerant solution at the architectural level) would be able to use the paper to arrive at a shared understanding of the terminology, problems and solution space. This would create a common language for their discussions. Based on the results of our study, we expect that they will be able either to identify an existing solution that would meet the requirements or to develop a new solution driven by the existing ones.

II. BACKGROUND

A. Software Architecture

Software Architecture (SA) [15] is an established software engineering area which focuses on the overall organization of a large software system using *abstractions* to express the logical coordination structure of complex and distributed systems. The emphasis in SA is on capturing the *system structure* (i.e., architecture topology) by identifying architectural components and connectors, and the *system behaviour* designed to meet system requirements by specifying how components and connectors are intended to interact [16]. While there is no universal agreement about what software architecture specification comprises, the concepts common to most definitions of software architecture are components, connectors, channels, configurations, interfaces and ports. Properties and constraints can be attached to architectural elements or to the overall system configuration. A coherent set of constraints and rules defines an architectural style.

To specify SAs, informal box-and-line notations have been replaced or supplemented by *formal* and rigorous Architecture Description Languages (ADLs) and by UML-based notations [17]. While UML-based notations for SA modelling are increasingly widely applied in industrial projects, domain specific ADLs are more frequently utilized in specific domains (like consumer electronics and avionics). Software architecture *views and viewpoints* are also growing in use as a technique for describing an architecture from a variety of specific stakeholders' perspectives [18]. Architectural *modes* which are viewed as different types of system behaviour in execution that depend on different operating conditions, environment and stimuli [19] are also more widely used in practice.

Documenting software architecture is not, however, the only direction of research in the area. Recently, a significant effort has been made to analyse a particular architecture and to generate a skeletal code out of the architectural specification. A variety of validation and verification techniques have been introduced to either assess how a SA conforms to requirements or to verify if the final system conforms to architectural decisions, as demonstrated by the many conventions devoted to the topic [20], [1], [21], [22]. Once the quality and dependability of a SA is assured, it can be used to generate a (skeletal) code, employing (semi-) automated processes [23], [24]. If this architecture models only *normal* behaviours of the system, the code generated from it will be unable to tolerate faults. As a consequence, the system may fail in unexpected ways if any faults occur. SA descriptions have been also integrated in industrial software development processes, as shown in [25], [26] and tools have been proposed in order to make specification and analysis rigorous.

B. Fault Tolerance

Generally, dependability is attained by using four means [27]: fault prevention, tolerance, removal and forecasting. Clearly, in practice a combination of all of these is necessary to ensure the required dependability level. All of them

are centred around the concept of fault. In our survey we follow the dependability terminology from [27]¹, which introduces a causal chain of dependability threats. In this chain, a system failure to deliver its service is caused by an erroneous system state, which is in turn caused by a triggered fault. That means that faults can be silent for some time and that their triggering (activation) does not necessarily cause immediate failure. Errors are typically latent, and the aim of fault tolerance is to detect and deal with them before they make systems fail.

This survey focuses on fault tolerance means that are used to avoid service failures in the presence of faults. The essence of fault tolerance (FT) is in detecting errors and carrying out the subsequent system recovery. Generally speaking, during the system recovery two steps need to be conducted: *error handling* and *fault handling*.

Error handling can be done in one of the following three ways: *backward error recovery* (sometimes called rollback), *forward error recovery* (sometimes called rollforward) or *compensation*. Backward error recovery returns the system into a previous state (which is assumed to be correct); the typical techniques employed to do this are checkpoints, recovery points, backup, restart, transaction abort etc. Forward error recovery moves the system into a new correct state; this type of recovery is typically carried out by employing exception handling techniques (found in many programming languages, such as Ada, Java, C++, etc.). There has been a considerable amount of work on defining exception handling mechanisms suitable for different domains, development and modelling paradigms, types of faults, execution environments, etc. (see, for example, [28]). It is worth noting here that, generally speaking, forward error recovery is more general than backward error recovery, as the former ensures that the system is recovered by moving it into any correct state (which may be a previous one or not). To conduct compensation, it is necessary to ensure that the system contains enough redundancy to mask errors by adjudicating the execution results and system states. Various replication and diversity techniques fall into this category. A wide range of software diversity mechanisms, including recovery blocks [29] and N-version programming (NVP) [30], have been developed and successfully used in industry.

The nature of fault handling is very different as it is intended to rid the system from faults to avoid causing new errors in a later execution. It starts with fault diagnostics, followed by the isolation of the faulty component and system reconfiguration. After that, the system typically needs to be reinitialized to continue its execution. Fault handling is usually much more expensive than error handling.

An area of particular interest to this survey is development of the *atomic action* mechanisms used for handling exceptions in systems in which components cooperate. Forward and backward error recovery in these system relies on cooperative exception handling, which involves several components. [31] introduced recursive system structuring using nested atomic actions and resolution of exceptions concurrently raised in an action. The follow-up work on Coordinated Atomic actions (CA actions) [32] extended this scheme to allow dealing with resources shared (or competed for) by several actions using the mechanisms of the ACID (atomicity, consistency, isolation and durability) transactions [33] and to explicitly support action abort as one of the possible exceptional outcomes. Another relevant

¹The authors are fully aware that there are different definitions of the terms fault, error, dependability and fault tolerance, which results in multiple frameworks for reasoning about this domain. The survey uses the terminology proposed in the Avizienis et al. paper to clearly define the boundaries of this work.

work [34] put forward the concept of the ideal (or idealised) fault tolerant component as a general blueprint for building components with well-defined fault tolerance (exceptional) interfaces and clearly separated fault tolerance component behaviour. The choice of the specific error detection, error handling and fault handling techniques to be used for a particular system is directly related to and depends upon the underlying fault assumptions. For example, replication techniques are typically used to tolerate hardware faults, whereas software diversity is employed to deal with software design bugs.

III. PROCEDURE FOR STUDIES SELECTION

The methodological approach used for identifying the papers relevant for this study makes use of both an *ad-hoc* (manual) literature review, and a *systematic* (automatic) literature review. We used both manual and automatic searches as a way to maximize the retrieval of relevant papers (as advocated in e.g., [35]).

First, we formulated some research questions. Then, in the ad-hoc review, the two authors (who have different backgrounds and specialised knowledge in software architecture and fault tolerance, respectively) selected relevant papers by looking at their own communities and networks and, more generally, at software engineering/architecture and dependability conferences and events. The selection has been carried out according to inclusion and exclusion criteria defined and finalized during the ad-hoc review. Conversely, the systematic literature review (SLR) was carried out to analyse, in a more systematic way, the body of knowledge in the field. The SLR uses the set of papers produced through the ad-hoc review as a *pilot* for testing the correctness of the SLR protocol.

A total of 75 papers have been selected and surveyed in this study, 39 coming from the ad-hoc review, 36 more from the SLR.

The rest of this section formulates the research questions (Section III-A), defines how the ad-hoc study has been performed (Section III-B), describes the SLR protocol used to refine the initial set of papers (Section III-C), and some threats to validity (Section III-D).

A. Research Question Formulation

This study focuses on papers on fault tolerance at the architectural level, which involves approaches to and solutions for tolerating those faults (errors, exceptions, etc.) that are caused by failing components, connectors and other architectural elements, and affect the whole system.

In order to specify the aim of our research, a research question (with sub-questions) were formulated, as described below:

Research Question: What are the approaches to architecting a FTS?

- Sub-question: What are the architectural methods, and processes leading to a FTS?
- Sub-question: Which architectural style/pattern is more appropriate for a FTS?

They are used to drive the papers selection process.

B. Ad-hoc Study

In the ad-hoc review, the paper selection process was driven by the authors' experience of the field and web-based searches using the most relevant search engines. We searched into the main conferences and journals on Dependability (DSN - IEEE/IFIP Int. Conference on Dependable System and Networks, LADC - Latin-American Symposium on Dependable Computing, IEEE TDSC - Trans. Dependable and Secure Computing), those on Software Engineering (ICSE - Int. Conference on Software Engineering, ESEC/FSE - European Software Engineering and Foundations of Software Engineering Conference, FSE - Foundation of Software Engineering, IEEE TSE - Transactions on SE, ACM TOSEM - Transactions on SE and Methodology, JSS - Journal on Software and Systems), as well as through those on both Software Architecture and Dependability (WICSA - Working Conference on Software Architecture, CBSE - Int. Symposium on Component-based Software Engineering, QoSA - Quality of Software Architecture, ROSATEA - Int. Workshop on the Role of Software Architecture for Testing and Analysis, WADS - Int. Workshop on Architecting Dependable Systems). To supplement our findings, we ran searches on Google and GoogleScholar.

As a way to drive the selection process, we incrementally defined inclusion and exclusion criteria, according to the research questions.

To be included, a paper needs to propose novel architectural principles, elements and styles, development processes and specification languages for reasoning about fault tolerance and its specific steps² at the architectural design stage. These would allow developers to explicitly represent abnormal system behaviour and state, and to express decisions on how to use redundant structures for system recovery.

Papers not validated by the computer science community are excluded: only peer-reviewed sources are included, while technical reports and PhD theses are not. Papers exclusively focusing on design and implementation, for example, on fault tolerance design patterns or various fault tolerant middleware and protocol stacks are excluded, too. This is consistent with our overall aim of analysing architectural approaches which are designed to make applications fault-tolerant. Papers which do not explicitly mention fault tolerance, in general, or its parts (such as error detection and system recovery) are excluded.

Table I provides inclusion and exclusion criteria. An article is selected if it satisfies *all* the pre-defined inclusion criteria and is excluded if it fulfills *any* of the pre-defined exclusion criteria.

As a result of this ad-hoc study, 39 papers were selected that match the identified inclusion and exclusion criteria.

²as defined in [27]

Inclusion Criteria	Exclusion Criteria
I1: Articles which discuss software fault tolerance and software architectures.	E1: Articles that do not focus on software, but only on, say, hardware or networked systems, for example “Multijunction fault tolerance architecture for nanoscale cross bar memories” [36]. E2: Articles which discuss fault tolerance at the implementation level only, or articles about architectures in general. E3: A study which is marginally related to fault tolerant systems and software architectures.
I2: One of the main objectives of the study is to present architectural languages for fault tolerant systems and exceptional architectural behaviour, methodologies, processes and approaches to architecting a FTS, or architectural styles that support the development of FTSs.	E4: Articles which do not present methods, approaches, styles or experience involving the use of fault tolerance at the architectural level of abstraction.
I3: A study that is in form of a scientific paper.	E5: Articles written in languages other than English. E6: Articles not validated by the Computer Science community, such as technical reports and PhD thesis. For instance, “Self-assembly for discreet, fault-tolerant, and scalable computation on internet-sized distributed networks” (PhD thesis) [37].

TABLE I: Inclusion and Exclusion Criteria

C. The Systematic Literature Review

The Systematic Literature Review (SLR) is a methodology created to standardize the methods of analysing research in particular scientific fields. Kitchenham [38] defines the SLR as ”a means of evaluating and interpreting all available research relevant to a particular research question, topic area or phenomenon of interest”.

The SLR can use *interpretive synthesis*, which aims to obtain specific-context knowledge through research articles without any background or initial data-set, or an *integrative review*, in which the reviewing process is based on a background of knowledge in the area of interest, which makes it possible to draw up an initial list of (pilot) papers. In this work, we use the initial pool of papers selected in the ad-hoc study to run an integrative review process.

The systematic review process includes a number of steps. Following [38], [39], we defined our SLR protocol by defining a suitable *search string* (used to match the set of research questions in the best possible way), a *time period* (adequate to capture as many relevant papers as possible), a list of *web search engines* (used to automatically identify a list of papers, potentially related to the review research topic), and a *studies selection process* (used to select those articles that better fits our research questions).

1) *Search String*: Based on the study research questions, we identifies some some relevant keywords and combined them in order to form the study research question. The search string we identified and used in the SLR is: (“*architecture*” OR “*architecting*” OR “*architectural*”) AND (“*fault tolerance*” OR “*fault tolerant*” OR “*exception handling*”). Although other relevant keywords had been identified, they were discarded in order to have a query that would be both strict and as complete as possible.

2) *Time Period*: In order to minimize the risk of missing relevant papers in the area, we took into account all articles published from January 1, 1995 to July 1, 2010³. This time period was carefully selected in order to cover as many relevant publications as possible.

3) *Web Search Engines*: The search string was run in the following search engines: ACM The Guide to Computing Literature⁴, ISI Web of Science, and SpringerLink. The ACM - The Guide search engine includes all the articles retrievable from ACM Portal Digital Library, IEEE Computer Society, and Elsevier.

4) *Studies Selection*: The studies selection process was started by putting the selected search string in the web search engines we had chosen, in order to identify the articles whose title or abstract include this search string. The selection process initially returned the titles of 3433 articles. Then, articles have been pre-processed to delete all duplicates, PhD theses and Technical Reports. The pre-processing stage returned 2540 non-duplicated, peer-reviewed papers. After that, the inclusion and exclusion criteria are applied when reading the introduction and abstract of the 2540 non-duplicated papers. 210 papers were chosen as a result of this stage. Then the selected papers are read in full. Those matching the inclusion criteria are selected to become the primary studies. A total of 59 primary studies were selected as a result of reading the full text of the 210 papers. Of these 59 papers, 23 were already part of the ad-hoc review.

Table II summarizes the selection process stages.

³*Comment: We sincerely apologize since this study covers articles published only till July 2010. Some problems have greatly delayed our submission. Being aware of the research area, we may confirm that the results and findings are still valid. As soon as the paper will be accepted for publication, we will carefully update the list of selected papers.*

⁴<http://portal.acm.org/guide.cfm>

Articles	ACM - The Guide	ISI - Web of Science	Springer Link	Sum	% Off
Total articles	1730	1201	502	3433	-
Articles pre-processed	1232	822	486	2540	-26%
Articles selected by Title and Abstract	111	69	30	210	-75.5%
Primary Studies	27	20	12	59	-71.9%

TABLE II: Papers' selection steps

It is worth noting that the SLR has been run through two iterations. During the first iteration, only 14 of the 39 pilot studies could be found by the systematic protocol. We carefully refined the search string and the web search engines (that are those presented above) in order to improve the coverage of pilot studies. Eventually, 23 of the 39 pilot studies could be found through the SLR.

In the end, 75 papers were selected for our study: 59 coming from the SLR, plus 16 more that were selected in the ad-hoc review (but not included in the SLR primary studies). The selected papers are listed in Appendix A.

To avoid having to classify different published versions of essentially the same research, we grouped all papers written by the same authors within the same research framework into what we call *approaches*. Figure 1 lists the 44 approaches we used to group the 75 selected papers. The table reports the approach number, the papers pertaining to a certain approach, and their main keywords (the most relevant being in bold).

D. Threats To Validity

According to [40], “*threats to validity are influences that may limit our ability to interpret or draw conclusions from the study's data.*”. Three main threats shall be considered: construct, internal, and external validity.

Construct Validity. Construct validity refers to the relevance of the studied parameters with respect to the research questions [41]. In order to maximize the construct validity we carefully tested our systematic review protocol and its search string. Starting from the research questions described in Section III-C, we defined an initial protocol and search string (presented in [42]). Accordingly, we initially run a pilot study and successively checked the ability of our search string to select relevant papers (i.e., those identified during the ad-hoc review). As discussed in [42], we revised our protocol and search string in order to maximize the coverage. The final search string has been carefully analyzed by the two co-authors, plus a colleague of us (Patricia Lago) working on the software architecture domain. Moreover, since both the ad-hoc and the systematic review have been designed and executed by the same persons, the chance of misinterpreting theoretical concepts is minimized.

Approach ID	Paper ID	keywords
A01	P01 P10 P55	SIMPLEX and ORTEGA architectures for real-time control systems
A02	P02	Framework for architectural upgrade
A03	P03	Runtime reconfiguration and middleware support
A04	P04	Exception handling patterns , from SA style to design patterns
A05	P06 P61	Meta-level architectures, exceptional patterns
A06	P08	Atomic actions for component integration, wrappers
A07	P09	Architectural Upgrade
A08	P13	Component replication and replacement, redundancy
A09	P05 P18	Architectural mismatch , co-operative architectural style, co-operative connectors
A10	P12 P16	Architectural mismatch
A11	P19	DRIP MDA method, CA action
A12	P21	Self-adaptable architectures
A13	P23	Reconfiguration -based fault tolerance
A14	P24	Architectural framework for handling multiple classes of faults
A15	P60 P25 P71	Performability and fault tolerance model
A16	P26	Formal specification pattern , layered architecture
A17	P11 P14 P15 P46 P17 P22	iC2C (C2 Style and Idealised FT Component Model), FaTC2, iCOTS
A18	P29	Exception handling and middleware
A19	P07 P27 P30	MDCE+ process (Idealised FT Component Model, from requirements to code & testing)
A20	P20 P32	Web services, availability, dynamic reconfiguration
A21	P35	SA reconfiguration , CA action
Approach ID	Paper ID	keywords
A22	P37	SA dynamic reconfiguration
A23	P38	Fault Tolerance and Testing
A24	P33 P34 P64 P67 P40 P58	iFT Elements : Specification, Verification, and Validation
A25	P28 P31 P36 P39	Aereal framework, Alloy , ACME
A26	P63 P42 P49	HoFE
A27	P41	GUARDS upgradable architecture for real-time dependable systems
A28	P43	Mechanisms for modeling FT services in SOA
A29	P44	Fault tolerance architectural aspect oriented modelling
A30	P45 P74	A generic fault tolerant SA for incorporating coordinated error recovery
A31	P47	A fault tolerant SA based on CA actions, cooperative EH, component level exceptions
A32	P48	Generic multi-level reference architecture
A33	P50	Use of aspects for designing fault tolerant architectures
A34	P51	OO design augmented with special (FT-specific) architectural properties
A35	P52	ComponentGOP framework for run-time reconfiguration of graph-based architectures
A36	P53	Tile architectural style
A37	P54 P70	Fault tolerance tactics and patterns
A38	P56	Architectural elements with redundancy schema
A39	P57	Power system information architecture
A40	P59	AQUA architecture
A41	P62	Survey on FT CORBA
A42	P65	Fault-tolerant service-based distributed systems
A43	P66	Reflective FT software architecture
A44	P68	Algebra of objects , crash failure

Fig. 1. Surveyed Approaches

Internal Validity. Internal threats to validity are mostly related to bias introduced by the study designers and/or executors. In this work, the ad-hoc and the systematic review have been conducted by both co-authors, with the support of an external expert on systematic reviews (Patricia Lago) and of a master student (Andrea Florio). A review protocol has been carefully defined in order to minimize the chance of a bias. Precise inclusion and exclusion

criteria, as well as research questions and study selection criteria have been defined. The review protocol has been carefully designed by the co-authors. Then, the review protocol has been executed by both co-authors (the first co-author being running the first stages, and complemented by the other co-author in the latest stages).

With respect to the design of the systematic review, a potential threat may affect our results. In fact, we cannot guarantee that a paper relevant to our review is always selected. While we tested and improved the protocol and the search string, and we had a colleague of us double checking the final search string and review process, it can still happen that a relevant paper is missing simply because its abstract does not contain the keywords defined in our search string.

External Validity. External threats to validity relate to how much the results of the study can be generalized and whether there exist enough evidence. This study includes only academic research papers, peer-reviewed by researchers in the Computer Science and/or Dependability communities. This study, thus, does not include industrial approaches for architecting fault tolerant systems. We may assume that solutions, methodologies, and styles identified by the research community may represent those appearing in industrial contexts. However, since solutions for architecting fault tolerant systems presented in other forms other than scientific peer-reviews papers were not considered in our study, the completeness of the results might be threatened.

IV. THE CLASSIFICATION FRAMEWORK

The classification framework proposed in this paper allows to compare the existing proposals for handling fault tolerance at the software architecture level.

As this survey serves two distinct and usually loosely related communities (namely, those of software architecture and fault tolerance), the classification framework proposed here takes into consideration two main viewpoints: the perspective of a software architect, who is interested in modelling fault tolerance concepts at the architectural level and wants to learn how to architect a fault tolerant system; and that of a fault tolerance specialist, who wants to learn how traditional low-level fault tolerance techniques can be applied at the architectural level. Our classification framework is therefore built around two respective types of parameters.

Starting with the set of research papers selected in Section III, this survey will classify each of them with respect to each of the two sets of parameters. Since the domain of architecting fault tolerant systems is an emerging area, it is only natural that it heavily relies on the concepts inherited from its ancestors. The decision to use both classifications was driven by the fact that the two communities have different vocabularies, expertise, domain specific knowledge, and ontologies; the survey is thus intended to provide a useful reference framework for both communities, yet without requiring specialised knowledge in both domains. While keeping the two sets of parameters separate may not seem the natural choice, our work on the survey has convinced us that this is a useful approach as it captures the features of the surveyed approaches which are non-overlapping and orthogonal. Bringing together the two sets of parameters creates a powerful common language which can be understood by researchers from both communities.

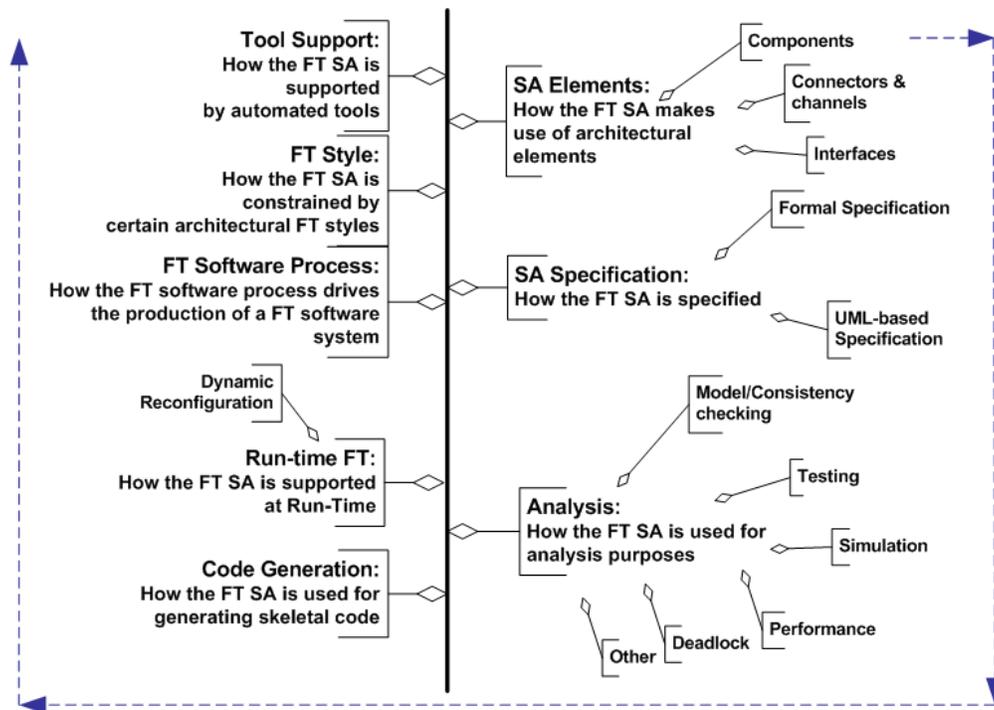


Fig. 2. Software Architecture Viewpoint

A. Classification Parameters from a Software Architecture Viewpoint

As outlined in Section II-A, software architecture specifies the high-level structure and behaviour of interacting components. Research on software architecture mainly investigates how to model an architecture, how the architectural specification can be analysed and then used for generating the application skeletal code, architectural styles and architecture-based software processes. The first set of parameters proposed in this work allows examination of how the existing approaches for architecting fault tolerant systems cover the main core concepts in software architecture. The selected software architecture parameters are described below and illustrated in Figure 2:

- *SA Elements*. As outlined in Section II-A, a software architecture is typically represented as an assembly of constituting elements, such as components and connectors, linked via interfaces.

The classification framework will investigate how the main architectural elements, that is `components`, `connectors and channels`, and `interfaces`, have been enhanced, extended, adapted or revised by the existing approaches in order to tolerate faults.

- *SA Specification*. As thoroughly discussed in many papers (e.g., [17], [43]) a software architecture can be specified through formal languages, model-based notations and box-and-line sketches.

The classification framework will identify what kinds of notation are used in the existing approaches for specifying FT SA, focusing on `formal` and `UML-based` notations.

- *Analysis.* A software architecture is designed in order to meet certain quality requirements. To this end, a range of analysis techniques have been proposed to allow software engineers to assess the software architecture and to evaluate its quality with respect to these requirements (e.g., [16], [44]). Model and consistency checking, testing, simulation, performance analysis and deadlock analysis are the most investigated analysis techniques at the architectural level.

The classification framework will cover the most common analysis techniques applied to fault tolerant architectures.

- *Code Generation.* A software architecture usually acts a bridge connecting the requirement phase to the system implementation. As long as the architectural model complies with certain quality expectations, it may be used to drive low-level design and to generate (skeletal) code.

The classification framework will identify the approaches which support skeletal code generation from FT architecture specifications.

- *Run-Time.* Traditionally, software architectures have been specified as unchangeable artefacts. Recently, however, the focus has been shifting to dynamically evolving architectures, capable of adapting during the system execution (e.g., self-adaptive, self-repairing, service-oriented architectures).

The framework will analyse how the principles of and solutions for building dynamic and evolvable architectures have been used to enhance ways of tolerating faults at the architectural level.

- *FT Software Process.* Software architecture is an important phase in a typical software engineering development process, which needs to be strongly connected to requirements, low-level design and implementation.

The framework will highlight how FT SA models presented in the surveyed papers are integrated in a more generic FT software development process.

- *FT Style.* According to [45], an architectural style is “*a set of design rules that identify the kinds of components and connectors that may be used to compose a system or subsystem, together with local or global constraints on the way the composition is done*”. There have been a number of architectural styles proposed: client-server, pipe-and-filter, layered, blackboard, C2, and many others.

This framework will identify and discuss new architectural styles introduced for dealing with FT architectures.

- *Tool Support.* Developing tools for supporting the specification, analysis, and implementation of fault tolerant software architectures is an important contribution to bridging the gap between academic research and industrial practice.

Our framework will identify the tools made available by the surveyed approaches, enabling automated support for specifying, analysing and implementing FT architectures.

B. Classification Parameters from the Fault Tolerance Viewpoint

This section defines a set of fault tolerance parameters used for classification and comparison of the existing approaches, summarised in Figure 3. All solutions under consideration explicitly deal with faults and/or errors by applying specialised fault tolerance mechanisms at the architectural level. These techniques are to be applied by the

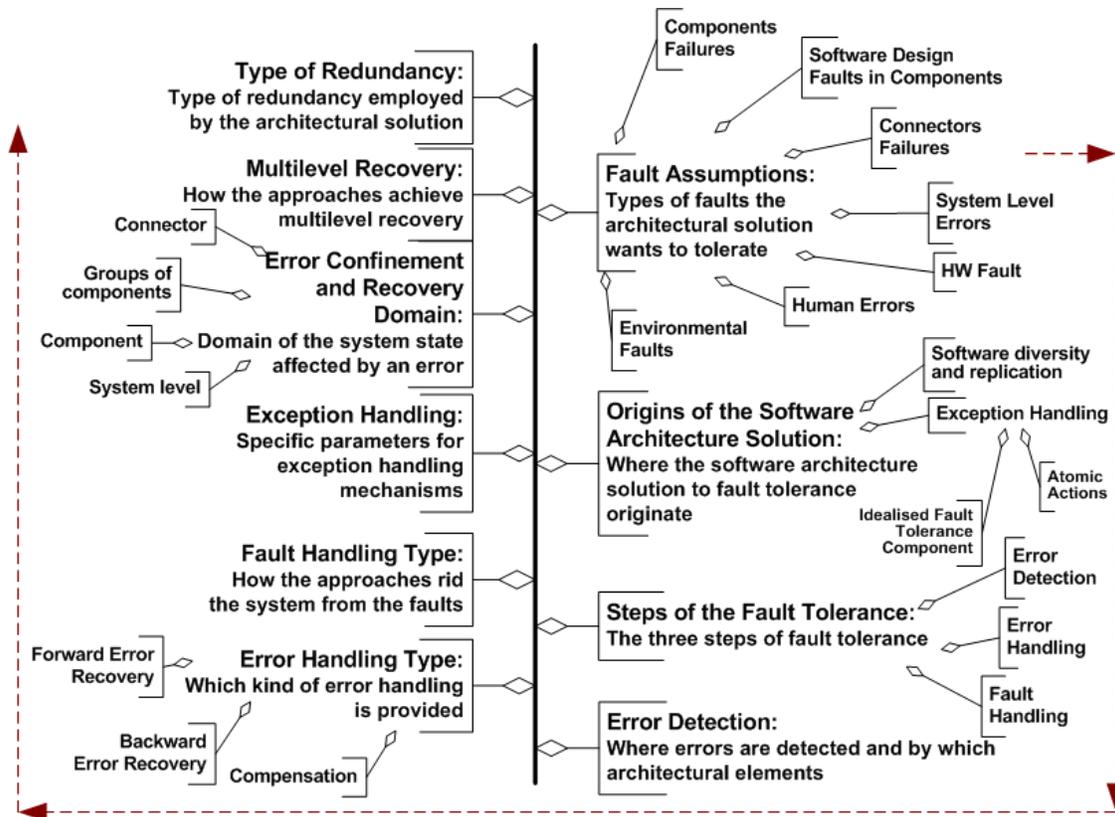


Fig. 3. Fault Tolerance Viewpoint

architects as part of system development. The choice of the solution depends on the types of faults which can and need to be represented at the architectural level, on the fault tolerance requirements and on the redundant resources which are included during architectural design.

- *Fault assumptions.* A clear statement of fault assumptions is in the core of fault tolerance as they define both the faults to be tolerated by a specific fault tolerance mechanism and the faults which can never happen during system execution. Correct system execution includes the execution of the fault tolerance mechanism and as such defines the foundation on which the mechanism is built. Fault assumptions are crucial in choosing the fault tolerance mechanism to be applied. Typical examples of fault assumptions are hardware faults, software design faults, environmental faults, operator mistakes and interaction faults.

In this survey fault assumptions define the types of faults that the proposed architectural solutions are designed to tolerate. These assumptions are typically expressed in terms of the architectural level at which we reason about the systems, i.e. in terms of components, connectors and configurations (e.g. failures of the individual components and connectors, and architectural component mismatches).

- *Origins of the software architecture solution.* Development of various fault tolerance techniques at the design

and implementation levels has been an active area of research since mid-60s.

All fault tolerance techniques developed for the software architecture level originate from these known techniques. The framework will place each architectural solution in the context of the original fault tolerance techniques, such as software diversity (*N-version programming* and *recovery blocks*), exception handling (Idealised FT component model and Coordinated Atomic actions), replication, etc.

- *Steps of fault tolerance.* The three steps of fault tolerance are error detection, error handling and fault handling (see Section II-B).

Every architectural approach discussed in the survey supports techniques of one or more of these types. The paper will describe each of them in these terms.

- *Error detection.* Fault tolerance always starts with detecting erroneous conditions. When the discussed architectural solution provides error detection, this work will describe where exactly errors are detected at the architectural level, by which architectural elements and how error detection is conducted.
- *Error handling type.* There are three general types of error handling: forward error recovery, backward error recovery and compensation.

When the proposed architectural approach provides error handling, the paper will explain which type of error handling it implements.

- *Fault handling type.* This parameter defines the way in which the proposed mechanism is designed to rid the system of faults.

The most typical approach to doing this at the architectural level is by reconfiguring the system, e.g. by removing or replacing the faulty architectural elements.

- *Exception handling.* Forward error recovery is typically implemented at the application level using appropriate exception handling mechanisms. Exception handling is the most general and effective approach to recovering from errors.

When the proposed architectural approach employs forward error recovery this work will analyse it using specific parameters developed for comparing exception handling mechanisms at the architectural level. These parameters have been defined by building on our previous classifications of exception handling mechanisms in programming [46] and include the following: (*i*) where exceptions are defined (e.g. the component interfaces, configuration), (*ii*) what exception handlers do, (*iii*) where handlers are attached, (*iv*) how exceptions are propagated, (*v*) which continuation model is used (resumption or termination), (*vi*) how the approach deals with concurrent exceptions, and (*vii*) what functions as exception handling scopes.

- *Error confinement and recovery domain.* A clear understanding of the domain of the overall system state which can be affected by an error is crucial for the success of recovery. Some fault tolerance solutions clearly define such domains and ensure error confinement, whereas others rely on a set of assumptions or impose a set of rules for programmers to follow in order to guarantee that errors are always contained.

This work will clearly identify the error confinement and recovery domain for each approach surveyed, doing

this in software architecture terms. Some of the examples of the structuring units of error confinement and error recovery are a `component`, an action involving several cooperating components, a `connector`, an architectural level exception handling scope, etc. For each approach, the architectural elements involved in recovery will be defined.

- *Multilevel recovery.* Fault tolerance needs to be recursive and to rely on recursive system structuring. This is crucial for localising recovery and reducing the overall system complexity. Multilevel recovery can be achieved by creating nested scopes or multiple system layers.

Where appropriate, this paper will discuss how the proposed architectural approaches achieve multilevel recovery.

- *Type of redundancy.* All fault tolerance mechanisms use redundancy.

For each solution the paper will identify the type of redundancy employed in terms of software architecture. The typical examples of redundancy are replicated component, additional connectors and components, extended component interfaces, etc.

V. PARAMETERS BY PARAMETERS VIEW: SOFTWARE ARCHITECTURE VIEWPOINT

This section analyses how surveyed *approaches* for handling FT SAs can be categorised in terms of the classification framework which captures the software architecture viewpoint, proposed in Section IV-A. Figure 4 summarises the classification results.

A. SA Elements

Software architectures describe software systems as consisting of components, connectors and interfaces required to glue the various architectural elements together. To make individual constituting elements, a set of them, and/or the entire architecture tolerant to faults, architectural elements are extended, enhanced or revised. This section will analyse how architectural elements are revised in order to manage fault tolerance at the architectural level. It is composed of three subsections: Section V-A1 focuses on components, Section V-B analyses connectors, while Section V-B1 shows how interfaces have been used in FT architectures.

1) *Components in Fault-tolerant Architectures:* Introducing *components* with fault tolerance features at the architectural level is the solution adopted in many of the surveyed approaches - see Figure 4. The overall idea is to enhance the traditional concept of component in order to make it more resistant to faults or capable to manage exceptions internally. A great variety of proposals have been put forward that can be categorised into four major classes, based on the use of *idealised fault-tolerant components*⁵, *redundant and diverse components*, *domain-specific components*, and *dedicated types of components*.

⁵In this study *idealised fault-tolerant component/connector* will be used to refer to what is sometimes called ideal and sometimes idealised component/connector

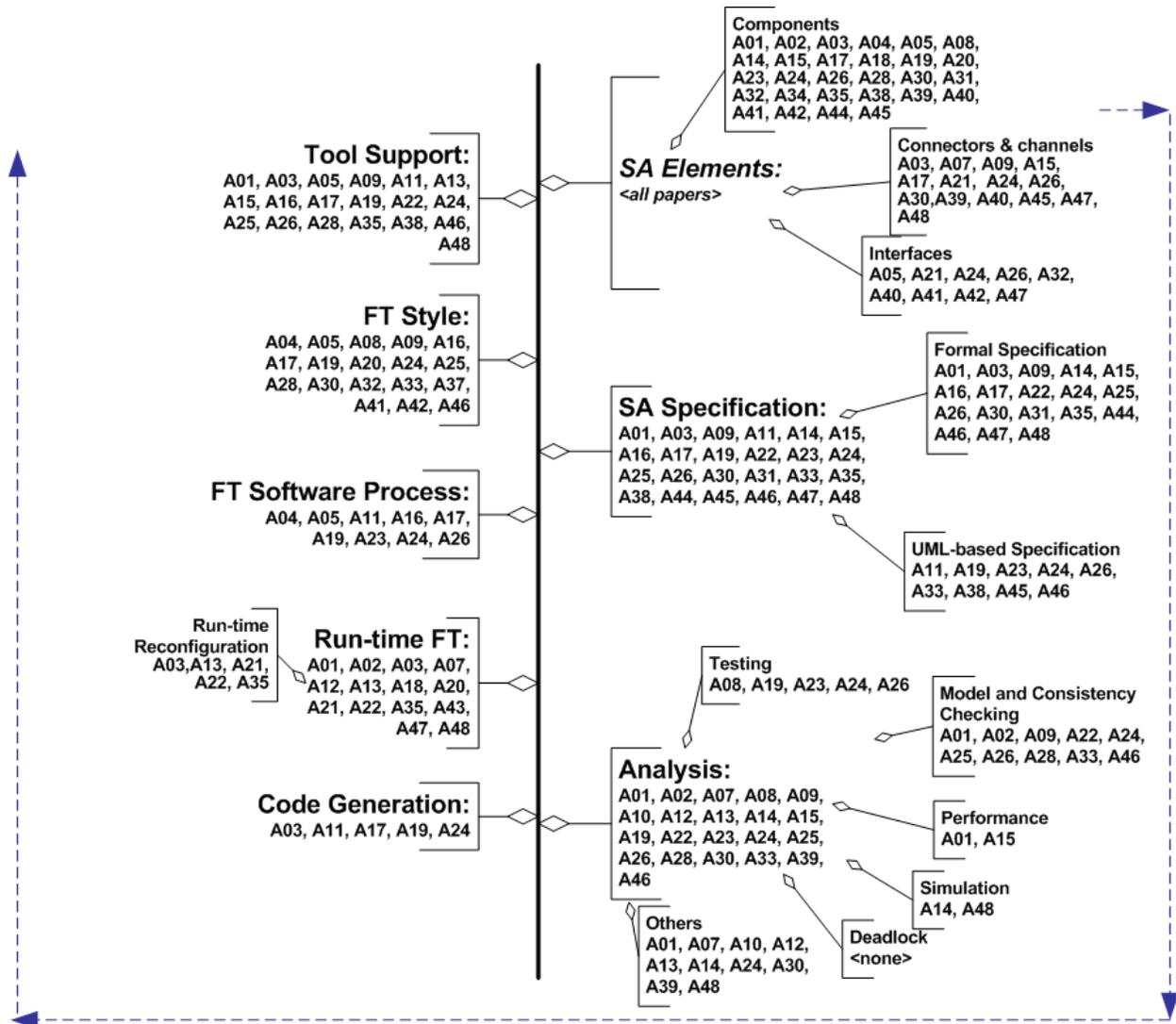


Fig. 4. The Software Architecture Viewpoint Classification

Idealised Fault-tolerant Components. The solution adopted by some of the papers in this category adapts the *ideal fault-tolerant model* [34] for building components with well-defined fault-tolerant (exceptional) interfaces and clearly separated fault tolerance component behaviours. This solution is implemented in A17, A19, A23, A24, A30 and A32, for a variety of purposes. Offering the most comprehensive treatment of the subject, A17 is the most representative approach in the set. In the A17 papers, a C2 [47] component⁶ is replaced with an idealised C2 component (iC2C), and commercial off-the-shelf (COTS) software components are put into protective wrappers in order to create idealised C2 COTS components (iCOTS). The concept of idealised fault-tolerant component is

⁶A C2 component is a generic architectural element that needs to meet certain structural and communication constraints, as defined in the C2 architectural style

used in approaches A19, A23, and A24 to incorporate exception handling in all phases of software development, to test fault-tolerant architectures, and to define idealised fault-tolerant elements, respectively. Approach A30 uses a different implementation of a fault-tolerant component (called `FTComponent`): an `FTComponent` may run in parallel with other `FTComponents` and may compete for shared resources. Exceptions concurrently raised by different `FTComponents` are managed by a coordinating component. In Approach A32 fault-tolerant objects (used to represent components) have to adhere to the standard interface characteristics of an idealised fault-tolerant component.

Using Redundant and Diverse Components. A large number of the approaches include fault tolerance features into components using some form of redundancy or diversity. These are A01, A02, A08, A14, A18, A26, A38 and A40.

All of them use *redundant* components. A01 uses analytic redundancy so that each component is internally composed of high-assurance and high-performance subsystems. In A02 different versions of the existing components are run in parallel, and this diversity is used to ensure system reliability during both system execution and upgrade. A08 stores components with similar characteristics (therefore replaceable) in a *redundant component array* (RAIC). A RAIC controller handles the dynamic updating of components, supporting a variety of fault tolerance strategies. A18 uses the RAIC concept proposed in A08 in order to design middleware that supports exception handling. A14 uses COTS component replication and majority voting for error detection and masking. A26 uses fault tolerant components with a crash failure semantics by employing various redundant/diverse components and decider components (for adjudicating or voting). In A38 architectural components (and more generally, elements) can have a redundancy schema attached to make them fault-tolerant. This is done using redundancy schemas, so that elements behave as fault-tolerant units. A40 introduces components called AQuA and Proteus, which are composed of replicated dependability managers and object factories. These are replicated, and groups of objects are created to manage a fault-tolerant architecture.

Domain-Specific Components. In certain approaches, domain-specific concepts (such as objects, web services and actors) are treated as fault-tolerant components; these are A28, A34, A35, A41, A42, A44 and A45.

In A34, A41 and A45 various types of (or abstractions of) *objects* are used to specify fault-tolerant components. In A34 a fault-tolerant element is a large-grain object (i.e. a component). In A41 the concept of *object group*, an abstraction introduced to ensure replication and failure transparency, is applied to provide a replication of CORBA objects. The metalevel architecture approach presented in A45 uses fault tolerance metaobject classes to describe fault-tolerant architectural components.

A28 extends the SOA metamodel proposed in [48], adding six types of components to introduce fault tolerance mechanisms into *service-oriented architectures*. In A35 *local programs* (which may be executed on several processors) are extended to become components, each associated with an interface. In A42, a web service is seen as a component which implements two extra interfaces to make the web service fault-tolerant. A44 proposes the use of *actors* (formally defined in an algebra of actors) to describe architectural components with crash failures

mechanisms.

Dedicated Components. This section will focus on approaches in which a set of dedicated components (types) are introduced to make the overall architecture fault-tolerant. These are A05, A15, A20, A31, A32 and A39.

In A05, four different components are presented to describe a generic software architecture that integrates sequential and concurrent exception handling: *Exception, Handler, Exception Handling Strategy* and *Concurrent Exception Handling Action*. The MAMA (model for availability management architectures) model put forward in A15 considers four types of components: *application tasks, agent tasks, manager tasks* and *processors*. In A20 four types of component are used to implement a crash-failure semantics and fault tolerance in Web Services. A31 uses user-defined and built-in components and channels for the specification of concurrent fault-tolerant systems. Three main types of component are considered: the *participant, coordinator* and the *external* one. The coordinator and external components are built-in and thus pre-defined, and used for exception handling coordination; by contrast, the participant component is user- defined, but includes built-in components.

In A32 a multi-level reference architecture for structuring fault- tolerant architectures is presented. This is composed of four layers: the *application, system support, low system* and *OS* levels, with each layer consisting of types of component specific to it. A39 proposes an information architecture for power systems, whose main components are the *operations control centre* and the *distribution substations*, connected by diverse fault-tolerant configurations.

In a class of their own, approaches A03 and A04 extend the traditional definition of component specification. In A03 components (as well as connectors) can raise exceptions; the “configuration exceptions” specification describes conditions that lead to occurrences of exceptions. A similar approach is taken in A04, where local handlers are attached to the component that raises the exception.

B. Connectors in Fault-tolerant Architectures

Software *connectors* have also been used to make architectures fault-tolerant. Similarly to the classification above, connectors used in fault-tolerant architectures can be grouped into *idealised fault- tolerant connectors, redundant and diverse connectors, domain-specific connectors* and *dedicated connectors*.

Idealised Fault-tolerant Connectors. Approaches A17, A24, and A30 introduce idealised fault-tolerant connectors by defining specific fault-tolerant connectors to connect the normal and abnormal parts of a set of idealised fault-tolerant components. A17 introduces specialised C2 connectors called *IC2C_bottom, IC2C_internal* and *IC2C_top*, and rules concerning how normal and exceptional information flows through them. For systems using COTS components, another two connectors have been introduced (*upper_detector* and *lower_detector*) to connect a COTS component with that responsible for handling abnormal activities. A24 introduces the concept of idealised fault-tolerant architectural element, allowing for subsequent instantiation of idealised fault-tolerant components and

connectors. A30 uses connectors to connect an `out_port` of one `FTComponent` with an `in_port` of another `FTComponent`.

Using Redundant and Diverse Connectors. Approaches A03, A07, A26, A39, A40 and A47 use redundant and diverse connectors to architect fault-tolerant systems. A03 proposes enriching connector specifications with explicit exception specification and handlers. A07 defines the concept of multi-versioning connector (MVC): an MVC is essentially a wrapper (similar to that in A17 for software components) that contains different versions of certain components, and is used to monitor the execution of several versions, to record the history of version execution and to dynamically remove less reliable versions or replace an old version with a new one. A26 uses halt-on-failure connectors to link redundant halt-on-failure components. The architecture proposed in A39 uses redundant communication links, implemented using various communication technologies for achieving redundancy. The AQuA architecture in A40 uses a set of Gateway connectors to provide fault tolerance by implementing various communication schemes and replication protocols. An AQuA gateway is responsible for finding a set of replicated objects that can implement a request, pass them the parameters, invoke their methods, and return the results. A Gateway is a complex connector composed of a naming service object, a handler factory (static and dynamic), a DII processor, and handlers for AQuA objects. A47 proposes a "Specific Fault-tolerant Connector". It is implemented as a specialised component that can capture Web Service interactions and perform built-in FT actions (assertions, recovery, monitoring). These connectors can use identical or equivalent service replicas available.

Domain-Specific Connectors. There is only one approach, A45, which uses domain-specific connectors. It proposes a metalevel architecture that runs metaobject protocols; these act as connectors between objects and metaobjects.

Dedicated Connectors. Approaches A09, A15, A21 and A48 use sets of connector types in order to manage fault tolerance at the architectural level. A09 uses co-operative connectors to represent collaborative activities between components. A co-operative connector integrates the concepts of *conversations* and *transactions*, and ensures that information is confined to localise the impact of a change. The model for availability management architectures (MAMA), which is put forward in A15, considers three different types of connectors: *alive-watch*, *status-watch* and *notify*. A21 proposes using complex connectors that are composed of an *Application* and a *Configuration* component, and an *Integrator* connector. The PRISM-MW architectural middleware in A48 allows the architect to define different types of `FTConnector`. Two types of `FTConnector` are defined in the paper: the `BestEffortFTConnector`, which enables the synchronisation of all active backups, and the `RealTimeFTConnector`, to be used when a shorter response time to the client is crucial.

1) *Interfaces in FT Architectures:* Some of the surveyed papers introduce dedicated *interfaces* to support the architecting of fault-tolerant systems, as shown in Figure 4. These can be grouped into two major classes: *interfaces for idealised fault-tolerant components* and *domain-specific interfaces*.

Interfaces for Idealised Fault-tolerant Components. In defining an idealised fault-tolerant component, approaches A21, A24 and A32 introduce certain interfaces which involve required or provided access points to the normal or abnormal parts of the idealised component. A21, for example, uses two main interfaces, the *Application services* and the *Configuration services*, in order to keep a clear distinction between architectural representation and reconfiguration, and to reflect the fact that this architecture consists of the Application and Configuration components. A24 defines two provided and two required interfaces to external services and exceptions. A32 extends what is done in the approaches discussed above by including two new interfaces: the *exceptional responses* interface to deal with degraded services, and the *abort exceptions* to notify the environment about a successful state restoration after a fault.

Domain-Specific Interfaces. A05, A26, A40-42 and A47 introduce domain-specific interfaces. In A05, the software architecture proposed for sequential and concurrent exception handling includes five different interfaces. The Exception component implements three public interfaces: the *IRaising*, *IGetInformation* and *IUpdateInformation*. The Handler component implements the private interface *IInvocation*, while the Exception Handling Strategy component conforms to the private interface *ISearcher*. Finally, the Concurrent Exception Handling Action component implements the public interface *ICooperation*. In A26, the HoFE (Halt-on Failure Element) abstraction defines two types of interfaces: the *I_HoFE_Prov*, which determines a set of operations provided by the HoFE, and the *I_HoFE_Req*, which specifies operations required by the HoFE for implementing its behaviour. In A40, there are three types of interfaces between the Dependability Manager in AQuA and other components in the AQuA architecture: the interface with the QoS Observer/Requester, the interface with the Advisor Observer, and the interface to the Host Observer/Controller components. A41 discusses the types of interface supported by FT CORBA implementations. In A42 the interface of a web service is extended to deal with checkpointing and rollback management. In A47 interfaces are generated from the definitions of the abstract operations of Web Services. The concept of Web Service equivalence is introduced to allow several valid interfaces to be generated from the equivalent Web Services.

C. SA Specification

Formal languages and model-based notations (such as UML - Unified Modelling Language) are the main means to formally or semi-formally specify a software architecture. Formal languages may be general purpose or designed specifically to describe software architectures (in which case they are referred to as Architecture Description Languages - ADLs). This section will first present an analysis of *formal languages* used to describe FT SA, and then *UML-based* notations, based on the parameters identified in Figure 2.

1) *Formal Specifications:* FT architectures are formally described using *ADLs* or *general purpose formal languages*.

Formal Description using ADLs. Approaches A01, A17 and A25 use or extend existing ADLs. A01 uses the

WRIGHT ADL (as is) to model the SIMPLEX architecture for dependable and evolvable process-control software-intensive systems. In A17, the Idealised C2 Component model is introduced and specified by extending the C2 architectural language. A25 uses the ACME architectural language to specify the architectural model.

Approaches A03 and A15, instead, define new ADLs, specifically designed for specifying fault-tolerant architectures. A03 extends a generic architectural language with constructs to model exceptions and their handlers. This approach can be applied to any existing ADL. A15 defines a new ADL called MAMA-dl, which combines a language for describing MAMA fault management architectures with a Fault-Tolerant Layered Queuing Network specification.

General Purpose Formal Specification Languages. Approaches A09, A14, A16, A22, A24, A26, A30, A31, A35, A44, A46, A47 and A48 use general purpose formal languages to describe fault-tolerant architectures.

A16, A24 and A26 use the formal *B-Method*, which supports state-based system specification by defining various types through mathematical sets. A16 uses the B-Method to specify a layered system. Abstract specifications are refined so as to smoothly incorporate reasoning about fault tolerance into the software development process. Both A24 and A26 combine the B-Method (to specify architectural elements, interfaces and exception types) and the CSP algebra (to specify architectural scenarios).

Approaches A24, A26, A31 and A44 use a *process algebraic formal language* (the CSP algebra in case of the first three). While A24 and A26 combine a CSP specification with the B-Method, A31 uses the CA Actions to structure normal and exceptional behaviours, employing CSP to combine behaviours for both user-defined and built-in components and channels. A44 extends an algebra of actors with mechanisms to model and detect crash failures (of actors). Primitives, such as the *ping* statement, are added for this purpose. An extended algebra of this type is used to specify distributed software architectures.

Some papers in A24 and A46 use specification languages (UPPAAL and Promela, respectively) intended for *the specification and checking of reactive systems*. In A24 the general architecture of iFTE, defined as its components, connectors and their interactions, is specified in UPPAAL using extended timed automata: the interactions between architectural elements are assumed to be blocking request/reply, and represented as synchronous channels. A formal Promela model of a FT architecture is proposed in A46; this is model-checked with the SPIN model checker.

In A14 the Stochastic Activity Networks, a variant of Stochastic Petri Nets with a graphical representation, are used to model the proposed methodology for handling multiple classes of faults in COTS- and Legacy- based applications. A22 uses the Modal Action Logic specification language to specify components in terms of attributes, actions and axioms. The Modal Action Logic specification is then systematically translated into finite state models, by extending the LTSA approach and the FSP algebra [49]. A30 makes use of Object-Z, a formal language based

on set theory and predicate logic, to describe an FTA. Such specification includes global types, FTComponent, Connector, CoordinatingComponent, ShareResource, and FTSystem class schemas. In A35 the architecture of component-based distributed software is defined by a logical graph. A47 defines DeWel, a Dependable Web service Language, to specify connectors for unreliable Web Services. A48 introduces DeSi, a visual environment that supports the specification, analysis and manipulation of a distributed software system deployment architecture.

Approach A09 makes use of extended time automata diagrams in order to specify the behaviour of architectural components.

2) *UML-Based Specifications*: Model-based notations based on the UML are used in approaches A11, A19, A23, A24, A26, A33, A38, A45 and A46.

Different sets of UML models have been used in different approaches. UML Class, Sequence Diagrams and Activity diagrams are used throughout the entire life-cycle to model fault tolerance from requirements to architectural and low-level design in A19. Component and State diagrams are used in A23 to specify normal and exceptional behaviour. A24 and A26 make use of (stereotyped) Component and Sequence diagrams to describe the four main idealised fault-tolerant architectural elements (iFTEs), enforcing the principles associated with the ideal fault-tolerant component model. A38 uses a UML profile to define ADL components, connectors and configurations with their stimuli, general and redundancy properties.

Approaches A11, A33, and A46 take a slightly different view. A11 is a model-driven engineering approach to architecting fault-tolerant systems. The approach uses a UML-based notation for modelling Coordinated Atomic actions and an MDA development support tool. Defining the strategies to be used for making the SA fault-tolerant, aspect-oriented modelling and model weaving in A33 are used to link a *base* model (i.e., the architecture without fault tolerance means) with the selected fault tolerance patterns, both specified using UML stereotyped diagrams. The integrated architecture and dependability models generated with the application of model weaving are represented as UML diagrams. In A46 fault-tolerant architectures conforming to selected FT styles are drawn using UML profiles. Component diagrams are used to describe both FT styles and FT architectures. Sequence diagrams are used to describe how FT components behave according to a FT style.

Approach A45 presents a model-based notation, not based on UML. It makes use of static and dynamic diagrams, taken from the Business Object Notation domain, to describe system classes and metaobjects (i.e., components), and their relationships and composition; dynamic diagrams represent the system behaviour in terms of exchanged messages.

D. Analysis

An interesting body of work has been developed to propose techniques for analysing fault-tolerant architectures. The analysis techniques in the surveyed approaches appear to belong to several areas: *model and consistency checking, testing, simulation, performance analysis, deadlock analysis, and others*.

1) *Model and Consistency Checking*: Model and consistency checking of FT SA is addressed in approaches A01, A09, A22, A24, A25, A26, A28 and A46 (and partially in A02 and A33).

A variety of model-checking engines have been used for model and consistency checking in the above approaches. The CSP-FDR model-checking engine has been used to validate the SIMPLEX architecture compliance to selected requirements in A01. The UPPAAL model checker is used in both A09 and A24. In A09 the safety of the normal and abnormal system behaviour is analysed by checking the co-operative architecture using timed automata and the UPPAAL model checker. A24 models idealised fault-tolerant elements in the UPPAAL language, and their correctness is verified by the UPPAAL model checker. The Labelled Transition System Analyser (LTSA) model checker is used in A22 for formal SA specification and model checking. The ProB model checker is used by both A24 and A26. A24 proposes a systematic verification and validation process, enabling the consistency checking of the formal B-Method model, the verification of general and specific abnormal scenarios violations, and the verification of user-defined properties. A26 enables the formal verification of iFTE and HoFE FT architectures, making it possible to identify behavioural inconsistencies, and assessing the consistency of interactions between architectural elements. Alloy [50], the modelling and analysis language created by Daniel Jackson, is used in A25 for validating the software architecture and exception model specified in the ACME architecture description language. The normal architectural specification is enriched with an exception flow view; the model thus developed is then converted to Alloy and submitted to Alloy analysis. The Bogor model checker is used in A28 for analysing FT SOA architectures. The SPIN model checker is employed in A46 for analysing whether and to what extent a fault-tolerant architecture which follows a particular fault-tolerant style satisfies both generic and application specific properties.

In A02 the Constraint Evaluator component is used to analyse the component version output with respect to domain constraints. A33 introduces a dependability model (obtained by model weaving of a normal architecture with FT strategies) to be used for analysis purposes.

2) *Testing*: *Testing* techniques are applied in approaches A08, A19, A23, A24 and A26.

The most relevant work is found in A19, A24 and A26. All of them provide *unit testing* strategies. A24 and A26 also provide *integration testing* of fault-tolerant architectures. A19 presents the MDCE+ methodology aimed at defining *unit testing* of the exceptional activities in component-based software systems. Testing and development activities are run in parallel in the development process. A24 performs unit, integration, and scenario-based testing at various levels of abstractions, and uses a grey-box testing strategy. A formal representation of the iFTE (idealised Fault-tolerant Element) is used to generate an execution sequence graph used for testing. A

scenario-based strategy also allows the generation of both unit and integration test cases for specific behaviour related to either functionalities or error handling. In A26, unit and integration testing is generated for HoFE architectures: unit test cases are produced for each HoFE, while integration test cases are generated to identify any mismatches between the architectural elements. Architectural scenario-based robustness testing is also introduced for testing single failures and stressful conditions.

A23 deals with *conformance* testing of the system implementation with respect to fault tolerance requirements. *Just-in-time* testing is applied to the architecture of redundant arrays of independent components in A08.

E. Performance Analysis

Performance analysis is applied in A15 (and partially in A01). In A15, the MAMA model is transformed into a knowledge propagation graph, and the Fault-Tolerant Layered Queuing Network (FTLQN) model into a fault propagation graph. Certain performability algorithms are then used to analyse these graphs to compute the architecture expected reward rate. In the Simplex architecture proposed in approach A01, hardware and software fault tolerance is ensured by using analytic redundancy which combines two software versions, a high-assurance and a high-performance one. The architecture provides a guarantee of the existing level of performance in case of software failure.

F. Simulation

Model simulation is briefly discussed in A14 and A48 (and analysed in related work). In A14 fault tolerance modelling is conducted using Stochastic Activity Networks which are simulated by employing the MOBIUS tool. The XTEAM environment, a suite of architecture description language extensions and transformations, implements a reliability estimation technique by simulating each single component. The output is used by the Desi's replication decision algorithms in A48.

G. Others

Analysis techniques proposed in other approaches include *monitoring* (A01, A07, and A13), *mismatch and compositional analysis* (A10, A12), *dependency analysis* (A24), *proofs* (A30), and *reliability* (A39).

Monitoring of fault-tolerant architectures is described in A13, and partially in A01 and A07. A13 describes how the Lira framework monitoring and reconfiguration capabilities can be used for monitoring critical events (including system errors and component failures) and reconfiguring the component-based system accordingly. In A01 monitoring techniques are employed to establish whether a run-time change in the Simplex architecture is safe or not. In A07 a specialised Multi-Version Connector monitors the execution of several components' versions, collects the history of version execution and dynamically removes less reliable versions or replaces the old version

with the new one.

Mismatch and *compositional* analysis are performed in A10 and A12 respectively. A10 proposes an approach for tolerating mismatches by using error detection and system recovery. In A12 the authors focus on component integration and misbehaviours caused by dependencies violation. They use self-adaptation to recover the system from misbehaviours.

Dependency analysis is used in A24 to support the testing phase. More specifically, dependency analysis is used to impose an order on how components are integrated, thus facilitating fault localisation. Three types of dependency chains are considered: *affected-by*, *affects*, and *related*. A dependency matrix is used to represent the relationships among architectural elements.

The *proof* process is used in A30 to demonstrate that fault-tolerant properties of FTA are satisfied. This is done through the Object-Z reasoning rules.

A39 considers various configurations of the power system information architecture. Their *reliability* analysis shows that the different solutions employed in the approach provide very similar reliability results.

H. Code Generation

Among the surveyed approaches A03, A11, A17, A19 and A24 provide support for code generation starting from a fault-tolerant architectural specification. Each of them presents an initial stage and prototypal tools.

A11, A19 and A24 share the use of *Java* as a reference programming language for the earlier specified fault-tolerant SA.

A11 introduces the CORRECT MDA approach together with the CAA-DRIP [51] framework, for automatically generating Java skeletal code from Coordinated Atomic actions. In A19 the authors propose an exception handling mechanism (at the implementation level) to support an explicit separation of normal and exceptional activities; the Java programming language is used to implement components, while the meta-object protocol (MOP) is proposed to implement the functions of the exception mechanisms. While the code generation phase is not automated, the authors offer clear directions as how to implement normal and exceptional behaviours defined in the fault-tolerant SA. Java and MOP are also used in A24, where a normal class signals an exception, which is intercepted by the MOP that will find an adequate exception handler in the exceptional class of this component. The exceptional classes are hierarchically organized, allowing subclasses to inherit handlers from their superclasses and, consequently, permitting the reuse of exceptional code.

A03 introduces an initial version of the Aster prototype. It supports the systematic mapping of architectures to their implementation using a middleware-based solution. The run-time support for exception handling is provided

by three main components running on top of a CORBA-compliant middleware. A17 proposes the FaTC2 object-oriented framework for implementing the iC2C architecture-level exception handling approach. FaTC2 extends the Java version of the C2.Fw framework.

I. Run-Time FT & Dynamic Reconfiguration

As illustrated in Figure 4, run-time architectures and dynamic reconfigurations are dealt with in a great deal of approaches. They can be classified as follows: *run-time management via middleware, exception handling for managing run-time evolution, upgrades, run-time reconfiguration, web services and reflection.*

Run-Time Management via Middleware. Approaches A03, A18 and A48 make use of a *middleware* infrastructure to deal with the run-time management of fault-tolerant architectures. A03 introduces the Aster environment aiming at providing an implementation-level support for architecture reconfiguration via a middleware architecture. The *interceptor facility of CORBA* is used for enabling component instances reconfiguration. In A18 the authors propose an approach to exception handling in component composition at the architectural level with the support of middleware. The middleware monitors the system execution at run-time and enforces the handling of exceptions (according to an exception model). The approach is based on the PKUAS J2EE-compliant middleware and makes use of the *component array, replacement, replica and reboot* mechanisms provided by it. A48 presents the Prism-MW *architectural* middleware platform that provides advanced run-time FT facilities (i.e., service discovery, replication and failover facilities).

Exception Handling for Managing Run-Time Evolution. Approaches A03, A12, and A18 use *exception handling* as a means for managing run-time evolution. A03 describes how to extend a generic architecture specification by introducing a configuration exception specification and exception handlers. When an exception is caught, the architecture is dynamically reconfigured using the ASTER framework. A12 proposes DeEvolve, a component-based self-adaptable P2P architecture which extends the traditional concept of adaptation by conducting explicit error detection and exception handling at the architectural level. The exception handling model in A18 describes when to handle exceptions and what strategies to employ to do so. The PKUAS J2EE-compliant middleware monitors the system execution and enforces the handling of exceptions (according to the exception model).

Upgrades. Approaches A01, A02 and A07 consider *upgrading systems* and their resilience. A01 introduces the Simplex architecture which ensures system fault tolerance and offers a reliable, safe and easy way of upgrading a system while it is in operation. Software *evolution* and *change management* are the essential aspects of this approach. A02 proposes the Hercules framework which maintains the previous components versions running in the course of upgrading the system so as to tolerate errors in the newly introduced version. A07 extends the work initially proposed in A01 to ensure fault tolerance of a system under upgrade by dynamically connecting old and new releases of off-the-shelves components.

Web Services. Approaches A20 and A47 are specific to *Web Services*. A20 proposes a pattern for improving web services availability and dynamic reconfiguration. By dealing with architectural elements of four types, this pattern captures the main functionalities specific to the dynamic composition of Web services. A47 uses the IWSD (Infrastructure for Web Services Dependability) to provide FT support services to run connectors transparently.

Reflection. Approach A43 analyses the benefits of *reflective architecture* for implementing adaptivity at the architectural level (i.e., separation of concerns, flexible support for late binding and intrinsic support for run-time adaptation). A four-layered reflective architecture for fault tolerance is briefly outlined.

Run-time Reconfiguration. Approaches A03, A13, A21, A22 and A35 consider run-time reconfiguration of fault-tolerant architectures. In A03 this reconfiguration is a way to handle architecture-level exceptions. In A13 the system is reconfigured according to monitored data. In A21 reconfiguration is specified using Coordinated Atomic actions to achieve a clear separation of the application and reconfiguration activities. Approach A22 uses the Modal Action Logic specification language for specifying and reasoning about dynamically-reconfigurable systems. In A35, the ComponentGOP framework enables run-time reconfiguration of graph-based architectures for tolerating faults. Component GOP supports both *programmed* and *ad-hoc* dynamic reconfiguration. update primitives. For ad-hoc reconfiguration, the Configuration Manager component offers the user a way to control the execution of a component-based distributed system. Crash detection and system reconfiguration are implemented as part of ComponentGOP. The Consistency Maintenance component is responsible for ensuring the component-based system consistency during dynamic reconfiguration.

J. FT Software Process

Ways of enforcing a fault-tolerant architecture in the software life-cycle are analysed in nine approaches: A04, A05, A11, A16, A17, A19, A23, A24 and A26. Those approaches can be classified into three major classes: *support for some specific phases in the process, architectural refinement, and V&V process.*

Support for some Specific Phases in the Process. Approaches A04, A05, A19 and A23 link the fault-tolerant architecting phase with other software development phases. A04 and A05 both focus on two main software development phases: architectural and low-level design. In A04 a software architecture for developing dependable systems is presented (based on two architectural styles and introducing atomicity, exception handling and coordinated error recovery at the architectural level) and then refined through a set of design patterns. A05 provides a systematic approach to incorporating exception handling during the detailed design stage by refining (through design patterns) the general components of the proposed architecture. The *Exception, Handler, Exception Handling Strategy* and *Concurrent Exception Handling Action* patterns are presented in order to map architectural decisions and exception handling guidelines at the low- level design. A19 covers the entire software development process,

from requirements to code. This approach focuses on how to specify normal and exceptional requirements, how to use this information to drive the component specification and design phase, and how to implement the system using a Java-based framework. The proposed software process is based on the Catalysis process. (A05 applies a similar strategy to the UML Components Process). A23 starts from Use Case diagrams, detects exceptions from use cases, and uses this information to guide the architectural design stage.

Architectural Refinement. Approaches A16 and A24 aim at *refining* abstract architectural specifications into more detailed ones. A16 proposes a specification pattern that can be recursively applied to formally specify exception raising and handling at each architectural layer. The goal is to start from an abstract specification in B, the formal notation in the core of the B method, and then gradually add lower layers by refinement. A24 uses UML (Component and Sequence diagrams) to specify the FT architecture, and generates B- Method and CSP specifications. This process extends the MDCE+ process presented in A19.

V&V Process. Approaches A24 and A26 include verification and validation means as an integral part of the proposed software development process. A24 presents a full process including the specification, verification and validation of FT architectures described using the iFTE paradigm. The B Method and CSP specifications are used to drive the verification and validation process (through testing). A26 presents an iterative, recursive and incremental process for developing fault-tolerant architectures which would also enforce the role of architectural abstractions. The process is presented in the context of the iFTE and HoFE architectural abstractions: iFTE implements error handling based on the exception handling mechanism, while HoFE is based on crash failure. The process considers a UML and a formal specification of the architecture, and its verification through unit and integration testing.

Other approaches. A11 is a Model Driven Architecture-based approach, in the DRIP Catalyst process, where Coordinated Atomic actions are specified using a UML-profile, and Java code is generated automatically from this specification. A17 discusses an exception handling system which adds fault tolerance to component-based systems at the architectural level. By combining two architectural styles, the iC2C and iCOTS styles are created. The ALEX framework supports the transition from an architecture modelled according to such styles into Java code.

K. FT Style

Interesting and relevant work has been done in introducing fault tolerance styles and design patterns for architecting fault-tolerant systems. The existing approaches can be classified into four categories: *approaches that reuse a library of styles or combine the existing styles, domain-specific fault-tolerant styles, styles that support the idealised fault-tolerant component model and others.*

Approaches that Reuse a Library of Styles or Combine the Existing Styles. These include A04, A17, A30, A33 and A46.

Approaches A04, A17 and A30 *combine existing styles* so to architect FT systems. In A04 the Idealised Fault-Tolerant Component and Role-based Collaboration styles are combined in order to produce a dependable software architecture. The Exception, Handler, Exception Handling Strategy, Reflective Role and Competitive Collaboration patterns are used to implement a dependable software architecture. A17 integrates the C2 and the Idealised Fault-Tolerant Component (IFTC) style in order to create an Idealised C2 Component style. The Fault-tolerant Architecture (FTA) presented in Approach A30 combines fundamental architectural styles, such as pipe-and-filter, repository and object-oriented organization.

Approaches A33 and A46 architect FT systems by using a *library of existing styles*. A33 uses a library of fault tolerance patterns in order to generate a fault-tolerant architecture. A46 models fault-tolerant mechanisms as styles by using profiled UML component and sequence diagrams; a library of FT styles is provided, based on combinations of provided FT components or connectors.

Domain-Specific Fault-tolerant Styles. These are presented in approaches A28, A41 and A42. In A28 the SOA style is modified in order to add fault tolerance mechanisms to SOA systems. To this end, a SOA metamodel provided elsewhere is extended. In A41, FT replication styles for FT CORBA are presented. The work focuses on *active* and (warm and cold) *passive* replication. A42 uses a web service (WS) FT style, in which a fault local to a WS is managed internally, but if the WS is unable to do so the failure is immediately propagated to the *Global Fault Manager*. The SOAP layer is extended to provide message logging, replay and acknowledgement capabilities. A *Fault Detector* has the responsibility to identify the fault.

Styles that Support the Idealised Fault-tolerant Component Model. These are discussed in approaches A04, A17, A19, A24 and A32. A04 uses the Idealised Fault-Tolerant Component style (combined with the Role-based Collaboration styles) in order to produce a dependable software architecture. A17 presents the Idealised C2 Component style, which treats the normal activities in the way it is done in the C2 style and the abnormal activity according to the IFTC style. A19 proposes a systematic approach to incorporating exception handling in all phases of software development, from requirement to implementation. At the architecture level, the idealised fault-tolerant component model style is used to produce a fault-tolerant SA. A24 introduces the Idealised FT architectural component (`iFTComponent`), the Idealised FT architectural connector (`iFTConnector`), and in general, the Idealised FT architectural element (`iFTE`). IFTE has its own style which prescribes the way components and connectors inside it are integrated. A32 proposes an architectural pattern called *generic software fault tolerance - GSFT*, which combines the structural characteristics of an idealised fault-tolerant component with the extensibility and flexibility of an object-oriented approach. This pattern includes an *FT interface* object, an *FT controller*, the software *variants* and *adjudicators* abstract classes, and can manage fault tolerance in concurrent systems.

Other styles. Styles which are not covered by the three categories above are presented in approaches A05, A08, A09, A16, A20, A25 and A37. A05 presents an exception handling architectural model (composed of four main components), systematically incorporating exception handling during the detailed design stage by refining the general components of the proposed architecture through design patterns. The *Exception, Handler, Exception Handling Strategy* and *Concurrent Exception Handling Action* patterns are presented in order to map architectural decisions and exception handling guidelines at the low-level design. A08 introduces RAIC (redundant arrays of independent components) as a way to achieve higher dependability and enhance the desirable properties of the component-based systems. The RAIC architectural style links redundant components and the controller. A09 introduces a co-operative architectural style to enable component adaptation in evolving dependable systems. A formal specification pattern that can be recursively applied to formally specify exception raising and handling at each architectural layer is introduced in A16. A20 proposes an architectural pattern based on two main principles: components implement crash-failure semantics, and the system supports dynamic reconfiguration. A25 introduces the concept of exceptional styles. The Aeral framework provides a basic architectural style (expressed in ACME) which can be extended in order to create new exceptional styles. A37 analyses the impact of certain FT tactics on the architectural pattern. This approach is different from the others surveyed in that, instead of providing new FT patterns, the authors i)analyse how several fault-tolerant tactics impact several of the most common architectural patterns, and ii)consider if and how this information helps software architects to incorporate fault-tolerant measures into the system architecture in the best possible way.

L. Tool Support

Some of the approaches provide tool support, which serves a variety of purposes. We group the existing tool support in the following classes: *tools for supporting the specification of FT SA*, *tools for supporting the analysis of FT SA*, and *tools for supporting the realisation of FT SA*.

Specification. Tools for supporting the *specification* of FT SA are presented in approaches A11, A15, A16, A19, A22, A24, A25, A26, A38 and A48.

Approaches A15, A22 and A25 use architecture description languages to provide tool support for the specification of FT SA. A15 provides a parser which accepts a description of various components (processors, application tasks, agent and manager tasks), of connectors interconnecting the components in the management architectural view and of the software functional dependencies as well as relevant performance and reliability definitions as inputs, and produces the MAMA and FTLQN models as output. A22 makes use of the LTSA⁷ tool for specifying and reasoning about dynamically-reconfigurable, multi-component systems with mechanisms for specifying normal and abnormal behaviour, and recovery actions (fault handling) by reconfiguration. A25 uses the ACMESTudio toolset⁸ to specify architectures in the ACME architectural language.

⁷<http://www.doc.ic.ac.uk/ltsa/>

⁸<http://www.cs.cmu.edu/acme/AcmeStudio/index.html>

Approaches A16, A24 and A26 use the B method and related tools⁹ to formally model FT architectures.

While they do not provide their own specific tool, approaches A11, A19, A24, A26 and A38 all provide UML profiles that are supported by any UML modelling tool.

A48 uses the DeSi visual environment to specify and analyse distributed software system architectures.

Analysis. Tools to support the automated analysis of FT architectures are presented in approaches A01, A09, A22, A24, A25, A26, A28 and A46. A01 uses the CSP-FDR model-checking engine¹⁰ to validate the SIMPLEX architecture compliance to selected requirements. In A09 and A24 the UPPAAL model checker analyses the safety of normal and abnormal system behaviours. A22 employs the Labelled Transition System Analyser (LTSA) model-checking engine associated with the FSP algebraic specification language. A24 and A26 both use the ProB¹¹ model-checking engine. In A28 FT SOA architectures are analysed with the help of the Bogor model checker. A25 makes use of Alloy [50] for validating the software architecture and exception model specified in the ACME architecture description language. The SPIN model checker is employed in A46 for examine if and how a fault-tolerant architecture, written according to a particular fault-tolerant style, satisfies both generic and application specific properties.

In A15 Layered Queuing Network tools are used to implement performability analysis. A48 refers to the XTEAM simulation environment for analysing DeSi specifications.

Tools for Supporting the Realisation of FT SA. Java-based support for architecting FT SA is proposed in approaches A05, A11 and A17. A11 uses the CORRECT tool to automatically generate Java code for the normal and exceptional behaviours, with the code generation process discussed in detail in [51]. Rather than generating Java code, A05 and A17 make use of Java-based frameworks: A05 presents a Java-based prototype of the proposed exception handling approach, based on previous work on the Guaran meta-object protocol, while A17 defines the FaTC2 Java framework, which extends the earlier C2.fw [47] Java framework for dealing with exceptional behaviours.

Frameworks and middleware for architecting fault-tolerant systems are proposed in approaches A03, A13 and A35. Both A03 and A35 use or extend CORBA. The Aster prototypal middleware in A03 extends the CORBA middleware in order to deal with exceptional behaviours, while a prototypal implementation of the ComponentGOP framework in A35 is implemented on top of CORBA, with Java and VisiBroker. As for A13, it builds on top of the Lira framework for reconfiguration-based fault tolerance in distributed systems.

M. Software Architecture Viewpoint Summary

Figure 5 summarises our analysis from the software architecture viewpoint.

⁹see, for example, <http://www.bmethod.com/php/outils-b-en.php>

¹⁰<http://www.fsel.com/>

¹¹<http://www.stups.uni-duesseldorf.de/ProB>

	SA concepts			SA Specification		Analysis							Run-time		Tool support		
	Components	Connectors & Channels	Interfaces	Formal Spec.	Model-based	Testing	M&C Check	Performance	Simulation	Deadlock	Others	Code Gen.	Run-time FT	Run-time Reconfig		FT Process	FT Style
A01	x			x			x	x			x						x
A02	x						x						x				
A03	x	x		x								x	x	x			x
A04	x														x	x	
A05	x		x												x	x	x
A06																	
A07		x									x		x				
A08	x					x										x	
A09		x		x			x									x	x
A10										x							
A11					x						x				x		x
A12											x		x				
A13											x		x	x			x
A14	x			x					x		x						
A15	x	x		x				x									x
A16				x											x	x	x
A17	x	x		x							x				x	x	x
A18	x												x				
A19	x				x	x						x			x	x	x
A20	x												x				x
A21		x	x										x	x			
A22				x			x						x	x			x
A23	x				x	x									x		
A24	x	x	x	x	x	x	x			x	x				x	x	x
A25				x			x									x	x
A26	x	x	x	x	x	x	x								x		x
A27																	
A28	x						x									x	x
A29																	
A30	x	x		x							x					x	
A31	x			x													
A32	x		x													x	
A33					x		x									x	
A34	x																
A35	x			x									x	x			x
A36																	
A37																x	
A38	x				x												x
A39	x	x									x						
A40	x	x	x														
A41	x		x													x	
A42	x		x													x	
A43													x				
A44	x			x													
A45	x	x			x												
A46				x	x		x									x	x
A47		x	x	x									x				
A48		x		x					x		x		x				x

Fig. 5. The SA Viewpoint

- *SA elements.* The most common ways of extending components and connectors for dealing with fault-tolerant architectures are idealizing them (by explicitly designing an exceptional component or connector dedicated to managing exceptional behaviours) or creating redundant and diverse elements (through redundant arrays of components/connectors, multi-versioned components and connectors, or redundancy schema), or simply creating architectures from ad hoc components and connectors. It can be seen from this summary that a fault-tolerant component does not necessarily need to be accompanied by a fault-tolerant connector, or the other way round. A few of the approaches, however, create specialised interfaces for managing fault-tolerant architectures. Some of the proposed solutions also involve legacy COTS components, ultimately creating a fault-tolerant architecture from legacy components which do not tolerate faults.
- *SA Specification.* Many of the approaches rely on a formal or model-based specification of the fault-tolerant architecture. Most of the proposed specification languages are based on the existing general-purpose formal languages or notations (CSP, Stochastic Petri Nets, Queuing Network, the B method, the C2 architectural language, the Finite State Process algebra, Acme). A few Architecture Description Languages are, however, used as is or adapted for specifying fault-tolerant architectures. This seems to suggest that most of this work has been conducted by researchers who tend to be closer to the fault tolerance community rather than the architectural one. This is partially confirmed by looking at the authors' backgrounds and where the papers are published. It is also worth noting that while model-based specifications of software architectures are widely used in practice today, most of the existing notations for dealing with fault-tolerant architectures are still based on formal languages. In most cases, model-based specifications rely on UML, and more specifically on UML profiles built for including fault tolerance concepts into traditional UML diagrams.
- *Analysis.* Creating a fault-tolerant architecture is not the only issue; analysing its correctness and consistency is important as well. As shown in this study, many approaches carefully check the architecture conformance to fault-tolerance requirements or properties, or generate test specifications for checking the functional quality of the architecture developed. Consistency and model checking is the most commonly practised approach to analysis; these mostly rely on formal specifications of the fault-tolerant architecture. Most of the analysis techniques also use or adapt the existing analysis engines: the FDR model checker, the UPPAAL model checker, the LTSA labelled transition system analyser, the Charmy model-checking features, and Alloy. The focus here is on defining and proving the properties required for the correct functioning of the fault tolerance mechanism, including the consistency and correctness of the chosen solution. Testing is sometimes conducted in combination with model checking.
- *Code generation.* A small number of approaches describe preliminary work on supporting the automated derivation of source code from fault-tolerant architectures. For a number of reasons, this is a challenging area of research. Code generation requires a precise and explicit specification of the system architecture, and this is provided only by a certain number of approaches (note that all the approaches which support code generation make use of formal or model-based techniques for specifying the fault-tolerant architecture). In general, code generation driven by architectural specifications is still a fairly underexplored research area. Among more than

a hundred architectural description languages existing today, only a few (e.g., Fujaba [24], ArchJava [23], and JavaA [52]) support code generation from the specification of normal (i.e., non-fault tolerant) architectures. It is thus clear that more research and tools are needed in the fault tolerance domain, for generating fault-tolerant code from architectural specifications. Our understanding is that only few approaches are mature enough to be applied in practice.

Two major approaches to code generation use specialised middleware and code generation of Java classes. The proposed middleware is specialised for targeting architectural level recovery by reconfiguration and systematic mapping of the architectural solution down to its implementation. In the Java code generation process the model is transformed into a set of application classes supported by specialised class libraries. All proposals support exception handling.

- *Run-Time*. Many approaches deal with fault-tolerant architectures at run-time. Some of them support architectural upgrade (i.e., ensuring fault tolerance of system under upgrade by using compensation), while others implement fault handling by run-time reconfiguration through middleware support, focus on run-time replacement according to the chosen recovery strategy, or guarantee fault tolerance during system reconfiguration.
- *FT Software Process*. Only a few approaches cover the entire process of architecting fault-tolerant systems. Some approaches describe how a fault-tolerant architecture can guide the low-level design process. Only a few approaches start from the requirement specification stage, and only one covers the entire process from requirements specification to coding. The main focus is either on supporting exception handling (including the idealised fault-tolerant component model and Coordinated Atomic actions) from the requirement to the design phase, or on defining specialised architectural styles which enforce exception handling and guide the transition to design.
- *FT Style*. A considerable number of approaches deal with styles or patterns for architecting fault-tolerant systems. While some of these specifically propose architectural styles for fault-tolerant systems (such as the iC2C style and the idealised fault-tolerant style), others employ architectural styles for meta-level architectures, or focus on creating a bridge between architectural specifications and low level design by using architectural and design patterns. Some of the research conducted aims to introduce co-operative architectural styles for defining error recovery that involves several components.
- *Tool Support*. Only a few approaches are supported by automated tools. Some of those tools are libraries to be used to implement a fault-tolerant system: they either support a model-driven development, or provide primitives for implementing architectures which follow the idealised fault-tolerant component model. Other approaches allow some form of code generation from architectural specifications or offer specialised middleware that supports run-time architectural reconfiguration. There are tools (LTSA and AEREAL) that are employed for validation of fault-tolerant architectures. Most of those tools, however, are at a prototypal phase or no longer supported. To the best of our knowledge, only a few tools are still being developed and improved.

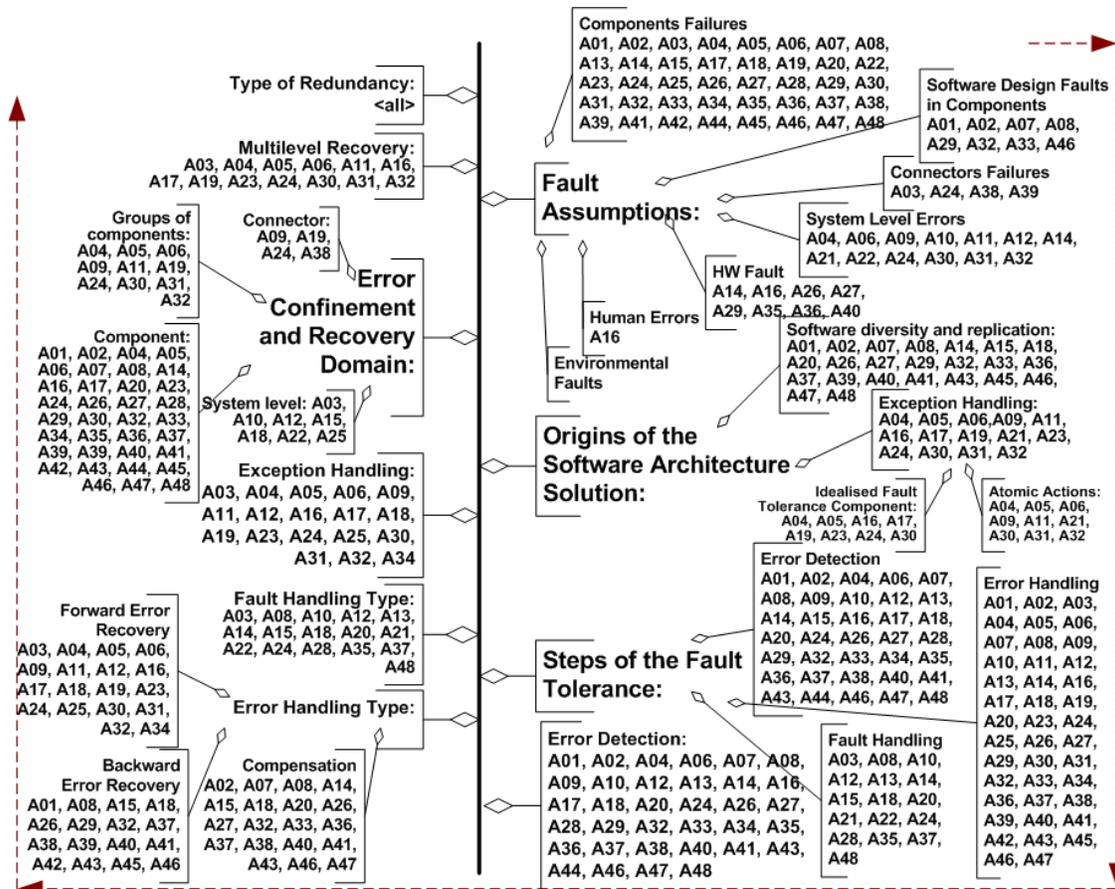


Fig. 6. The Fault Tolerance Viewpoint Classification

VI. PARAMETERS BY PARAMETERS VIEW: FAULT TOLERANCE VIEWPOINT

This section analyses the selected approaches following the fault tolerance classification parameters defined in Section IV-B. The results of this analysis are summarised in Figure 6.

A. Fault Assumptions

Fault assumptions define the types of faults that the proposed architectural solution is intended to tolerate. Our main focus here is on describing the assumptions in terms of software architectures. The approaches offer a good coverage of the types of faults they can tolerate: some of them deal specifically with the failures of architectural elements (components or connectors), others are intended for system (configuration or architectural) level exceptions and inconsistencies, while the remaining few deal with hardware faults, human errors and those detected during architectural reconfiguration.

Component failures, which represent errors at the architectural level, are addressed in the majority of the approaches (see Figure 6). These can be grouped into five major categories:

- Approaches A03-A06, A14, A15, A17-A19, A22-A25, A30-A32 and A34 deal with the exceptions explicitly reported by individual components when they are not able to deliver the expected results; it is very often the case that the component interfaces are extended with the descriptions of the exceptions they can signal (e.g. A06 and A25).
- Approaches A13, A15, A20, A22, A26, A27, A29, A34, A35, A44 and A48 focus on failures of individual components with well-defined semantics of their failure behaviour, such as crash, value or Byzantine failures. Even though several of these deal with value failures, only approach A34 specifically requires that components have pre- and post-conditions associated with them. Some of these approaches deal with both crash and value failures (A26 and A27).
- Some approaches are not specific about the failure behaviour of the components, assuming only that components can fail and leaving the assumption particulars and the choice of error detection measures till later design phases (A28, A35-A39, A41, A42, A44, A45, A46, A47).
- Several approaches (A01, A08, A29, A32, A33 and A46) deal with general types of design faults in component software (bugs) (see Figure 6, item *Software Design Faults in Components*).
- Two approaches (A02 and A07) explicitly focus on software design faults in the new releases of the components.

Connectors failures are explicitly addressed by approaches A03, A24, A38 and A39. These failures are propagated by connectors to a higher-level architectural context in which they are dealt with. For example, approach A03 introduces the concept of configuration exception handling to support system reconfiguration when exceptions are explicitly propagated by individual components or connectors. To do this systematically, components and connectors interfaces specify the exceptions these elements can propagate when they cannot deliver the service expected from them.

Approaches addressing *system level inconsistencies and errors* deal with abnormal situations which affect the architectural level and involve several interlinked components. A04, A06, A09, A11, A21, A24 and A30-A32 assume that such errors emerge in the course of a defined set of components explicitly cooperating. Approach A10 specifically focuses on general solutions for dealing with architectural level mismatches. Approaches A12, A14 and A22 deal with general types of abnormal behaviour exhibited at the system level.

Many of the approaches deal with several types of faults: for example, approach A22 covers individual component failures and system level errors. Approach A29 provides tolerance of both design bugs in components and hardware crashes.

A relatively small number of approaches focus specifically on *hardware faults* (A14, A16, A26, A27, A29, A35, A36, and A40) but even when doing so they always consider the implications at the level of software architecture and propose solutions in the software architecture context. A14 and A40 deal with the hardware-induced software faults and host crashes, whereas A16 deals with service failures, where services are viewed as a combination of software and hardware on which this software is located. The approaches discussed do not explicitly deal with the environmental faults. This is because they usually assume that the environment (e.g. sensors) is represented by components and the component faults are therefore used to represent the faults in the environment. Approach A27

is different in this respect as it introduces special architectural measures for dealing with sensor failures.

B. The Origins of the Software Architecture Solution

It is only natural that the majority of the architectural solutions discussed here rely on the existing fault tolerance mechanisms which were initially developed for the design or implementation phases. It is clear, at the same time, that as some of these mechanisms (e.g. found in the middleware) deal with the implementation level problems by using implementation level means, they do not have any counterparts at the architectural level.

By now a number of the major fault tolerance mechanisms have been moved to the architectural level and presented in terms of components, connectors and system configurations (see Figure 6, item *software diversity and replication*). A01 uses the idea of recovery blocks with the special decision logic implementing the acceptance test and with the two alternates (the high-performance and the high assurance ones); the initial architecture supporting only parallel execution of the alternates was later extended to allow their sequential execution, closely adhering to the original idea of recovery blocks. Approaches A28 and A42 apply the idea of recovery blocks for architecting Web Service composition, A29 supports both recovery blocks and distributed recovery blocks [53]. Approaches A02, A07 and A33 rely on the ideas of N-version programming. Approach A27 uses the idea of N-modular redundancy, whereas A08 and A47 use a combination of recovery blocks, N-version programming and N-modular redundancy. A20 is similar to the N Self-Checking Programming technique for software fault tolerance [54]; however it does not employ two levels of comparison since this scheme does not deal with design bugs. A14 uses replication of the COTS components and the majority voting on component results. A15 employs several known replication strategies at the architectural component level: primary-standby, active and load-balancing redundancy. A40 relies on a different set of replication strategies, including first-pass, voting and passive. A26, A32 and A46 develop general architectural frameworks which allow N-version programming, recovery blocks or component replication to be applied. A48 is based on active replication (without voting) combined with heartbeats used to detect replica failures, after which the system is reconfigured.

The majority of these approaches use architectural level *exception handling* (see Figure 6) as it provides structuring support for application-specific recovery. The *idealised fault tolerant component* blueprint initially proposed for designing fault tolerant systems [34] has been extensively used for developing various architectural solutions (A04, A05, A16, A17, A19, A23, A24 and A30). The ideas of *atomic actions and Coordinated Atomic actions* were applied at the architectural level in approaches A04-A06, A09, A11, A21 and A30-A32 to support cooperative recovery involving several cooperating components and to structure systems using nested units of component cooperation. Note here that A04 and A05 introduce an extra (orthogonal) architectural view for reasoning about component cooperation. Approach A21 uses Coordinated Atomic actions for structuring system reconfiguration during fault handling.

C. Steps of Fault Tolerance

The three general steps of fault tolerance which each approach can provide are *error detection*, *error handling* and *fault handling*.

Several approaches cover all three steps (A08, A10, A14, A20, A24 and A37). Approach A08 is based on the RAIC (redundant array of independent components) architecture which uses several diverse components to detect errors, recover from them and, if necessary, decommission failing components. Approach A24 introduces an idealised fault tolerant element architectural style, which incorporates features for error detection, error recovery by exception handling and localised reconfiguration at the level of individual connectors and components.

The majority of approaches deal with error detection and error handling (e.g. A01, A26, A27, A38). Several approaches (A05, A11, A19, A23, A25, A30-A32 and A34) only deal with error handling (all of them propose solutions for handling exceptions at the architectural level without discussing how to detect errors or how to handle faults causing these errors) while two approaches (A21 and A22) focus exclusively on fault handling and, more specifically, on reconfiguring systems to rid them of faults.

A number of approaches (A15, A28, A35 and A48) detect component failures and, without trying to recover from these system errors, move directly to fault handling by system reconfiguration. For example, A15 uses a fault propagation graph to define the part of the architecture that needs reconfiguring.

D. Error Detection

Error detection means are an element in the majority of the architectural solutions discussed here - see Figure 6. These solutions offer a wide variety of ways in which errors can be detected at the architectural level.

Approaches A01, A08, A24, A26, A28, A29, A32, A37, A46 and A4 use specially designed acceptance tests to check the results of diversely designed components (alternates). For example, A28 employs a dedicated monitoring component to check these results, A29 describes acceptance tests as component aspects, and A47 uses executable assertions for detecting component (web service) failures.

Approaches A02, A07, A08, A14, A20, A26, A27, A32, A33, A36-A38, A40, A41, A43, A46 and A47 employ *compensation schemes* which compare multiple execution results for detecting errors. This is used for tolerating both hardware faults and software design bugs. For example, in approach A07 a dedicated multi-version connector monitors the execution of several component releases and records its history. By contrast, approach A26 proposes a style which includes a decider component for result adjudication when versions are executed sequentially (recovery blocks) or in parallel (compensation).

Some approaches explicitly *assume that component errors are detected by some extra architectural elements* (connectors and/or components) as part of the architectural style for developing idealised components capable of detecting errors. In this case these components detect errors themselves and the corresponding exceptions are defined in the component interfaces as part of the software architecture description (A04, A16, A17, A24).

Several approaches (A15, A35, A46, A48) describe *specific error detection functionalities* on which they rely to detect failures of components: watchdog, heartbeat, timeouts and pings. For example, A15 specifically links these

functionalities with dedicated connectors.

Approach A34 offers an *application-specific solution* that uses pre- and post-conditions to detect component failures. To ensure the effectiveness and the practicality of this solution, error recovery is here attached to FT components, large-grain objects specifically constructed to contain errors.

In approach A44 *failure detectors* are introduced at the architectural level to make decisions about component failures. Approach A06 explicitly introduces a protective wrapper, a redundant architectural element intended for detecting errors coming from the component or sent to the component. Approach A10 offers general solutions for detecting errors caused by architectural mismatches using additional information associated with components.

System level error detection is used in approaches A06, A09, A12 and A13. For example, A13 uses a dedicated monitoring infrastructure called Lira, which collects information about critical events at the architectural level. In addition to acceptance tests and result comparison, approach A08 uses just in time component testing to ensure better coverage of error detection.

E. Error Handling Type

The application-specific error handling in the form of *forward error recovery* which uses exception handling is clearly the dominant approach - see Figure 6. Considering that this type of recovery is the most general and effective, this is understandable. Moreover, from architects point of view, the assumption that architectural elements can be rolled back for recovery does not seem to be appropriate; the main reason for this can be either that they often deal with the COTS components or that this assumption is too strong/restricting to be accepted at the earlier steps of system development. This is, for example, explicitly stated by the authors of A46, who clearly understand that it is not always possible to develop checkpointing for the black box COTS components or to expect that such components come with this functionality implemented.

Several approaches use various forms of *compensation* (A02, A07, A08, A14, A15, A18, A20, A26, A27, A32, A33, A36-A38, A40, A41, A43, A46 and A47) - they employ redundant (diverse or replicated) versions of software components to mask errors. All of them run several versions in parallel and adjudicate their results to define the correct ones. Approach A20 employs redundant sources of information of the Internet (Web Services) and four dedicated architectural elements (called Bridge, Comparator, ServiceBroker and FT_Registry) which ensure comparison of results and, if necessary, dynamic reconfiguration (i.e. fault handling) when some of the elements fail. Some of these schemes support various forms of compensation, for example, A08 allows a combination of N-version programming and triple-modular redundancy depending on the configuration chosen. Approach A33 supports architecting double modular redundancy and N-version programming schemes using aspect-oriented architecture design.

Approaches A01, A08, A15, A18, A26, A29, A32, A37-A43, A45, and A46 rely on *recovery blocks* and execute redundant versions sequentially with some sort of acceptance test which checks version results. It is interesting that some of them allow versions to be run in parallel (A01 and A40) so they use the results of the first version that passes the acceptance test as the correct ones.

A number of approaches (A08, A15, A18, A26, A32, A37, A38, A40, A41, A43, A46, A47) introduce general architectures within which *both compensation and backward error recovery* can be implemented depending on the specific fault assumptions and the redundant resources available. Some are even more general and support forward error recovery as well: A32, A37, A43, A46. The levels of detail in these proposals are very different. Some of them introduce only general architectures (e.g. the metaobject one in A43) and discuss their potential applicability for various types of recovery in general terms. Others offer much more detailed architectural frameworks for selecting specific architectural recovery styles (e.g. A46). A37 discusses various solutions for extending the existing architectures with fault tolerance by integrating various fault tolerance tactics into them.

Approaches A02 and A07 both employ previous releases of components as a means for tolerating faults in the new release using diversity; in addition to this, A07 also supports architectural undo when the states of all releases are moved back (A07 being the only approach explicitly supporting backward error recovery).

It is interesting to note that approach A18 supports general system level exception handling, giving as the specific examples the compensation-based approaches: the RAIC defined in A08 and component replication.

Approach A10 introduces a general framework for dealing with errors caused by architectural mismatches and as such supports all types of error handling.

F. Fault Handling Type

With system (re)configuration being in the core of software architecture, it should not come as a surprise that a considerable number of approaches offer support for fault handling by reconfiguration - see Figure 6.

These approaches defining interesting and unusual combinations of error recovery and fault handling mechanisms are briefly presented below.

Approach A08 uses the RAIC architecture that, in addition to error recovery by compensation, supports the decommissioning of underperforming components and inclusion of new components into the array (with their state modified to be identical with the rest). All this activity is conducted by a specialised RAIC controller.

Approach A12 supports various exception handling strategies which can provide both error recovery and system reconfiguration for fault handling. The latter is done by using a set of available components (by adjusting their parameters or by building new configurations of them). Approach A03 explicitly introduces reconfigurations as configuration level exception handlers.

Approach A13 introduces multi-level reconfiguration of assemblies of COTS components. System-level error detection is conducted by monitoring certain system-level events, which is followed by run-time fault diagnosis and fault handling by reconfiguration. Three reconfiguration levels are defined: the component, composition and global levels, so that if it is not possible to handle a fault at a lower level an attempt is made to do so at a higher one.

Approach A15 introduces a managing layered architecture in which components can fail and be repaired and different reconfiguration policies are defined. It supports replica reconfiguration by adding, repairing or removing individual replicas. The components to be involved in system reconfiguration are defined by the fault propagation

graph, which is to be developed as part of system architecture. The focus of this work is on defining modelling techniques for analysing systems which support fault handling.

Exception handling approach A18 allows some forms of fault handling by supporting description of the so-called "induced failures" defining components, which are to be treated as failed after an exception is raised and handled. This approach combines system level exception handling and fault handling strategies. It considers the RAIC, component reboot and component replacement as special cases.

In addition to error recovery by the compensation approach, A20 supports dynamic reconfiguration using a registry which allows the system broker to dynamically find redundant and available web services which can be deployed to replace the failed ones. Approach A21 uses Coordinated Atomic actions for architectural reconfiguration as part of fault handling, which allows us to tolerate system level faults during reconfiguration by employing cooperative exception handling. Approach A22 defines a theory for modelling and verification of various dynamic reconfiguration strategies. Approach A24 introduces localised reconfiguration as a possible way of conducting error recovery; this is executed at the level of individual connectors and components.

Approach A35, built on top of component architectures (such as CORBA, DCOM and EJB), reconfigures a system by replacing failed components with their replicas, ensuring that all messages are redirected to the new components.

As was mentioned before, approach A37 supports integration of fault tolerance tactics into the existing system architectures. Some of these tactics support fault handling: removing components from the service, resynchronisation of the failed component and its reintegration into the system.

Approach A48 implements architectural level reconfiguration of the replica set for each service component, so that these replicas can be run in an active replication scheme without voting.

G. Exception Handling

In order to summarise exception handling architectural approaches, it is necessary first to look into how exceptions are propagated. This is typically done after an error is detected either by architectural elements (components, connectors) or by the system/architecture itself. Approaches A03-A06, A16, A17, A19, A23-A25, A30-A32, and A34 assume that *individual components signal exceptions* to the level of architecture. Approaches A03, A24 and A25 extend this idea to individual connectors. Approaches A09 and A18 allow abstract exceptions to be signalled at the architectural level, without assuming that they come from either individual components or connectors. These approaches are applicable when the system architecture includes a glue code responsible for composing components into a system. Approach A06 assumes that components are wrapped by a dedicated wrapper component that represents component behaviour in the context of the system architecture, in which case it is clear that this component should be able to signal architectural-level exceptions.

Our second classification criterion establishes where exceptions are handled in the architecture. Approaches A04, A05, A16, A17, A19, A23-A25, A30 and A34 support *exception handling by individual components*, whereas approaches A24 and A25 allow not only component-level but also connector-level exception handling. Approach

A34 uses the architectural concept of an FT component, a large grained unit which has exceptions and exception handlers associated with it.

A considerable number of approaches support *cooperative exception handling* which involves sets of cooperating components: A04-A06, A09, A11, A19, A24, A30-A32. Let us consider A04 as an example. This approach defines two orthogonal dimensions in which exception handling is conducted: (i) exceptions can be defined in the component interface, in which case they are handled by the caller component and (ii) exceptions can be raised by cooperating components which play a role in the same collaboration, in which case a cooperative recovery involving all the roles is conducted. Approach A30 supports two level exception handling - at the component and at the cooperative action levels. Some approaches support system level exception handling without associating it specifically with any architectural elements: A06, A12 and A18. For example, in approach A12 exception handling strategies are defined during component (peer services in this case) composition, and they can include various architectural level activities, including changing the attributes of individual components as well as connections between components and system reconfiguration. These specific cases of error recovery reveal the similarity between this approach and approach A03, which explicitly defines configuration-level exception handling, so that reactions to architectural level exceptions are defined in terms of system reconfiguration activities.

Some approaches use stylised architectures or a set of guidelines to represent exception handling at the architectural level. Two major general stylised approaches are the idealised fault tolerant component (A04, A05, A16, A17, A19, A23, and A24) and Coordinated Atomic actions (A04-A06, A09, A11, A30-A32). An example of the former is approach A17, in which each component is architected as a style with two components and three connectors, with a specialised component called *AbnormalActivity* designed to conduct exception handling. All of the approaches relying on Coordinated Atomic actions use some form of concurrent exception resolution at the architectural level to deal with exceptions concurrently raised at the system level by action participants. For example, approach A09 supports cooperative recovery of several COTS components involved in cooperation at the architectural level by introducing a cooperating connector that contains all cooperative recovery and deals with concurrent exceptions by resolving them when necessary.

It is interesting to note here that all exception handling approaches without exception use the termination model [55], which is clearly much more suitable than the resumption model for the architectural level reasoning about systems.

Many approaches (e.g. A15, A37 and A46) do not support full-fledged exception handling even though all exceptions propagated from components trigger recovery actions, as they allow only predefined types of handling based on backward error recovery or compensation.

H. Error Confinement and Recovery Domain

The majority of the architectural error recovery approaches assume that *individual components confine errors*. For example, in approaches A02 and A07 errors are confined to the component (i.e. its new release). Approach A24 extends this to connectors and to sets of cooperating components. Connectors are considered as units of confinement

and recovery in approaches A09, A19 and A24. In A09 an action of cooperating components is architected as a specialised cooperating connector which has several roles, one for each component, so that recovery always involves groups of components.

There are several approaches (A04-A06, A09, A11, A19, A24, A30-A32) which use *cooperative recovery* involving several cooperating components at the architectural level (the `Groups of Components` item in Figure 6). In this case, the architecture of the systems is built using structuring units of component cooperation as the first class entities. These approaches deal with both error confinement and consistency of recovery involving several components which work together not only during normal system execution, but during recovery as well. Approach A04 relies on two architectural views for confinement and recovery: (i) the individual component view, in which components confine errors so that recovery is conducted at a higher level by the caller, and (ii) the collaboration view, in which cooperative recovery can be conducted by involving several components which play a role in the same collaboration. Approaches A05, A30 and A32 support two types of recovery: local recovery at the component level and cooperative recovery, which involves several cooperating components inside an action, allowing two types of recovery domains: individual components and actions (groups) of cooperating components.

Approaches A03, A10, A12, A15, A18, A22 and A25 do not assume any general ways of system structuring for confinement and recovery and thus rely on application-specific decisions architected for each specific exception (the `System level` category in Figure 6). For example, in approach A18 error confinement and recovery areas are defined by the exception strategy description, individually for each system level exception. In a similar way, approach A25 allows any element of the architecture to be involved in recovery if there is an explicit exception flow linking it with the source of exceptions (another component).

I. Multilevel Recovery

The idea of defining recursive structuring units of system architecture is crucial for dealing with system complexity and for fault tolerance. Introducing these units allows architects to define error confinement and recovery domains and to ensure that when a recovery in such a unit is not possible, the responsibility for recovery is passed on in a systematic and well-defined way to the containing unit. This structuring is referred to as multilevel recovery at the architectural level.

Unfortunately, the vast majority of the approaches do not introduce multilevel recovery. Yet there are several which support it in various innovative ways. As was mentioned above, approaches A05, A30 and A32 allow *two level error recovery*: when recovery is not possible at the component level, the responsibility for recovery is transferred to the level of a group of cooperating components. Approaches A06 and A31 take this idea further by supporting nested system structuring using the concept of *Coordinated Atomic actions* which involves nested sets of cooperating components. Approach A11 only focuses on architecting systems using nested Coordinated Atomic actions, based on understanding the simplest action as an action with only one participant which executes a set of operations bracketed at an ACID transaction.

The architectural approaches which follow the *Idealised fault tolerant component* blueprint (A04, A05, A16 A17,

A19, A23, A24 and A30) support layered system structuring for Recovery, treating each component as a recovery domain; when this recovery is not possible, the responsibility for recovery is passed on to the caller component using exception propagation.

Approach A03 introduces an interesting way of nested architecture structuring by incorporating a mechanism for defining subconfigurations and nested reconfigurations into the configuration level exception handling. Subconfigurations describe a subset of components and connectors whose interaction may lead to an exception. Reconfigurations define handlers, so that each of them can define a new configuration, which in its turn may include nested specifications of reconfigurations.

J. Types of Redundancy

All architectural level fault tolerance approaches use architectural redundancy. Each specific case defines a specially tailored combination of redundant resources. A clear understanding of the redundancy which each scheme uses often offers a deep insight into how a scheme functions and its benefits or weaknesses. Several typical examples are discussed below.

Many approaches assume that component and connector specifications are extended with definitions of the exceptions which they can propagate (for example, A03, A04, A12, A15, A24, A30).

Approach A03 uses redundant configurations for exception handling at the architectural level.

In approach A06 an additional component, i.e. component wrappers, is introduced as a glue code responsible for local error detection and recovery as well as cooperative recovery.

Approach A26 relies on an architectural style which links together redundant components and a decider component to ensure the fail stop semantics of the composed element.

Approaches A01, A02, A07, A08, A14, A15, A20, A26, A27, A29, A32, A33, A35, A36, A37-A41 and A45-A48 use diverse/replicated components together with some adjudicating and controlling mechanisms (often architected as specialised connectors) to manage them. For example, approach A01 uses the so-called analytical redundancy in two versions: a high-assurance and a high-performance one. A02 and A07 use several releases of a component. A45 proposes a metaobject architecture called FRIENDS, which manages replicas of the base level components. Approach A47 defines the concept of an equivalent service to allow components to be used as diverse implementations.

Each of the approaches that follow the IFTC blueprint defines a style which consists of a set of specialised connectors and components that together ensure the idealised behaviour of the application component. For example, in approach A17 each component is implemented as a style which consists of two components and three connectors.

To support architectural level exception handling, approach A12 uses redundant links between components, redundancy at the system level to allow adjustment of the underperforming components and redundant exception handling code at the system level. Approach A13 uses the following redundancy: the Lira infrastructure for monitoring (error detection) and reconfiguring, and a decision maker for fault diagnosis. Approach A25 employs a number of redundant features: additional information related to exceptions associated with components, exception

flow views that link components during handling using a special type of connectors called *duct* (this effectively offers a redundant view) and an additional code for exception handling attached to each component.

K. Fault Tolerance Viewpoint Summary

Figure 7 summarises our analysis from the fault tolerance viewpoint.

- *Fault assumptions.* The assumptions are typically expressed in terms of software architecture. The majority of the approaches focus on dealing with various failures of individual components (these include exceptions propagated by components when they cannot deliver the required service). There is some earlier work (before 2003) on dealing with software design faults by employing diverse implementations of components. No approaches directly deal with faults in the environments; the main reason for this, in our view, is that the environment of the system is typically dealt with by wrapping all sensors or actuators into regular components, so that failures of those components represent environmental faults.
- *Steps of fault tolerance.* Several approaches deal with all three steps: error detection, error handling and fault handling. The vast majority of the approaches support error handling, as their main purpose is to ensure that the system continues to provide the service by dealing with errors. Fault handling by reconfiguration is another very popular approach; this can be explained by the fact that dealing with system configuration and reconfiguration is a concern typically addressed at the architectural level, so adding fault handling to this functionality is a very natural step.
- *The origins of the software architecture solution.* All the approaches discussed in the survey originate in the previous work on software-implemented fault tolerance. Nearly all major fault tolerance techniques have been moved to the software architecture domain (recovery blocks, N-version programming, conversation, replication, exception handling). Unlike the mainstream fault tolerance, only a few approaches rely on checkpointing, which has limited applicability for COTS components. The role of exception handling at the architectural level is growing as many researchers realise that this is the most suitable mechanism for achieving effective recovery during system architecting. At the same time, there is a growing realisation that the Internet can offer a rich source of replicated and diversely-implemented components (services).
- *Error detection.* All error detection solutions can be divided into two categories. Some of them assume that the components themselves detect errors and inform the architectural level (e.g. by raising exceptions). Approaches of the other type use additional architectural-level solutions, such as wrappers, glue code, connectors, monitoring infrastructure.
- *Error handling type.* Forward error recovery is the predominant solution: the majority of the approaches use various forms of exception handling for handling errors. Compensation is quite popular as it can be naturally expressed in terms of software architectures (e.g. by employing diverse or redundant components), and although it requires substantial component-level redundancy, it can clearly be successfully used at the architectural level. Only a few approaches support backward error recovery, based on the assumption that components provide the save/restore state functionality.

	Fault Assumptions				Origins of the SA solution		Steps of the Fault			Error Handling Type				Error Confinement and				Type of Redundancy						
	HW Fault	Components Failures Sw Design Faults in Components	Connectors Failures	Human Errors	System Level Errors	Environmental Faults	Sw Diversity and replication	Exception Handling - Atomic Actions	Exception Handling - Idealised F. T. comp	Error Detection	Error Handling	Fault Handling	Error Detection	Compensation	Forward Error Recovery	Backward Error Recovery	Fault Handling Type		Exception Handling	System Level	Component	Groups of Components	Connector	Multilevel Recovery
A01	x	x				x			x	x		x		x					x					x
A02	x	x				x			x	x		x	x							x				x
A03		x	x						x	x		x			x	x	x	x						x
A04		x			x		x	x	x	x		x			x	x	x		x	x	x	x	x	x
A05		x					x	x		x					x	x	x		x	x	x	x	x	x
A06		x			x			x	x	x		x			x	x	x		x	x	x	x	x	x
A07	x	x				x			x	x		x	x											x
A08	x	x				x			x	x		x	x	x	x			x		x				x
A09					x			x	x	x		x			x	x	x				x	x		x
A10					x				x	x		x	x					x	x					x
A11					x			x	x	x					x	x	x				x			x
A12					x				x	x		x	x		x	x	x		x					x
A13		x							x	x		x	x					x						x
A14	x	x			x	x			x	x		x	x	x				x		x				x
A15		x				x			x	x		x	x	x				x		x				x
A16	x			x			x		x	x		x			x	x		x		x				x
A17		x					x		x	x		x			x	x		x		x				x
A18		x				x			x	x		x	x	x	x	x		x	x	x				x
A19		x					x		x	x					x	x					x	x		x
A20		x				x			x	x		x	x	x				x		x				x
A21					x			x				x						x						x
A22		x			x							x						x		x				x
A23		x					x			x					x	x		x		x				x
A24		x	x		x		x		x	x		x	x		x	x	x		x	x	x	x	x	x
A25		x								x					x	x		x		x				x
A26	x	x				x			x	x		x	x	x					x					x
A27	x	x				x			x	x		x	x						x					x
A28		x							x	x		x	x					x		x				x
A29	x	x	x			x			x	x		x			x					x				x
A30		x			x		x		x						x	x		x		x	x			x
A31		x			x			x		x					x	x		x			x			x
A32		x	x		x			x	x	x		x	x	x	x	x		x		x	x			x
A33		x	x			x			x	x		x	x							x				x
A34		x							x	x		x			x	x				x				x
A35	x	x							x			x	x					x		x				x
A36	x	x				x			x	x		x	x						x					x
A37		x				x			x	x		x	x	x				x		x				x
A38		x	x						x	x		x	x	x					x			x		x
A39		x	x			x				x					x					x				x
A40	x					x			x	x		x	x	x					x					x
A41		x				x			x	x		x	x	x					x					x
A42		x								x					x					x				x
A43						x			x	x		x	x	x					x					x
A44		x							x			x								x				x
A45		x				x			x	x				x						x				x
A46		x	x			x			x	x		x	x	x						x				x
A47		x				x			x	x		x	x							x				x
A48		x				x			x			x	x					x		x				x

Fig. 7. The FT Viewpoint

- *Exception handling.* In the surveyed approaches, exceptions can be raised either by the individual architectural elements (components or connectors) or by the dedicated additional software located at the architectural level, which detects errors. After exceptions have been raised, they can be propagated to and handled at a number of destinations: individual components or connectors, sets of cooperating components, architectural configurations. To assist architecting systems with exception handling, many approaches introduce specialised exception handling styles. These are typically based on either the earlier idea of the idealised fault tolerant component or on the concept of Coordinated Atomic actions.
- *Error confinement and recovery domain.* The typical approach assumes that errors are confined to individual components (although only a few of them explicitly state this or discuss how to ensure that this assumption holds). In some cases this allows the architecture recovery to be reduced to the recovery of individual components. This solution works only if we do not need to ensure the consistency of several interacting components while recovering one of them, and when there are guarantees that the erroneous information does not leave the components. Such local recovery is cheaper than that which involves several components, but in many systems in which components are tightly coupled the only possible solution is to involve a set of cooperating components in recovery. Approaches which provide this solution often introduce architectural level structuring (e.g. scopes, actions) which precisely defines the sets of components to be involved in cooperative recovery.
- *Multilevel recovery.* The majority of the approaches do not provide multilevel recovery. The few that do this can be categorised into two major groups, those which support nested structuring following the ideas of Coordinated Atomic actions, and those which rely on the layered architecture using the ideas of idealised fault tolerant component. There are two solutions, however, that do not fall under either of these categories: the first one introduces nested subconfigurations and the second one provides only two level recovery: at the levels of components and actions.
- *Fault handling types.* The majority of fault handling mechanisms support various forms of architectural level reconfiguration that ensures that the faulty component is removed. This approach fits into the software architecture view extremely well. Some of the existing solutions extend exception handling with fault handling functionalities. There are several interesting solutions proposed, including multilevel reconfiguration, definition of reconfiguration policies, and introduction of a dedicated reconfiguration layer. Unfortunately, the functionality for conducting fault diagnostics is explicitly introduced in just a few approaches.
- *Type of redundancy.* The approaches under consideration use a broad variety of redundant resources. Each approach uses a specially tailored combination of extra components, connectors, inter-component links and component interfaces. Thus, many employ diverse or replicated components with some adjudicating and controlling components. All of the surveyed approaches need extra time for conducting fault tolerance activities.

VII. CONSIDERATIONS AND RESEARCH RECOMMENDATIONS

The investigation of how to specify and analyse fault tolerant architectures started in around 1996 with the CMU/SEI SIMPLEX architecture for ensuring fault tolerance in evolving hardware and software systems. Since 2001 there has been a steady stream of research papers, mostly presented in workshops and conferences. Between 2001 and 2003 the main focus was on architectural styles and patterns for fault tolerant architectures. Since 2004, however, there has been a noticeable growth in the area: new contributors joining in, a new interest in the topic, indicated by a larger number of papers published, and a new maturity of the research community, as demonstrated by the many journal and book publications produced.

In this section we will describe similarities identified when comparing the two perspectives discussed in this study (Section VII-A), and propose an agenda for future directions of research in the area (Section VII-B).

A. Identified Similarities

The authors of this study are convinced that software architecture and fault tolerance rely on very similar sets of fundamental principles which are crucial for both domains. Moreover, we believe that the work in the two areas, perceived by many as quite distinct, is in fact driven by similar forces. These include:

- Managing system complexity. Without a clear understanding of system structuring, systems cannot be reasoned about, not can fault tolerance be developed systematically.
- Effective system structuring. Fault tolerance cannot be achieved without it (for example, to ensure error confinement, to understand error propagation and to conduct multilevel recovery), while the essence of software architecture is system structuring at the earlier development phases.
- The importance of component level reasoning. Components are clearly the basic blocks of any architecture and the main elements of any architectural level representation. At the same time, ensuring that errors are contained and dealt with at the level of components (e.g. nodes, modules, classes, agents, objects, and servers) is the most efficient way of achieving fault tolerance. Various wrapping techniques and styles are used at the architectural level to ensure these properties of components.
- Understanding component dependencies. In the SA community, knowing component dependencies is essential for architecture evolution and configuration [56]. In fault tolerance, the choice of the appropriate recovery strategy heavily depends on a clear understanding of error propagation between components. Both domains strive to reduce the number of dependencies.
- A clear definition of component interfaces. In system integration, architectural configurations can be built only when the component interfaces are known. For error detection, either component interfaces are extended with interface exceptions, or the information visible at the interfaces of several components is checked to detect complex erroneous conditions at the system level.
- Similar modularization criteria. These are applied in constructing complex systems to ensure their fault tolerance and to improve their architectures [57], [29], [58]. Both domains aim at developing systems with fewer inter-component links to make it easy for the developers to understand, maintain, upgrade and verify them.

The essence of system architecting is defining system components and their interconnections, thus keeping information links between components in check contributes to better system structuring. There are several reasons why modularization is particularly important for achieving fault tolerance. For one thing, system modularization allows representing its functioning in both normal and fault tolerance modes. Also, generally speaking, decreasing the number of inter- component links facilitates system recovery as it allows errors to be contained in smaller recovery domains (i.e., smaller parts of system architectures).

- Addressing earlier development phases. While software architecture is always determined at the earlier development phases, it is becoming clearer to many researchers working in fault tolerance that in order to be successful fault tolerance should be systematically applied from the same stage [59], [60].

It is interesting that the fault-error-failure chain, which is in the core of dependability concepts, is effectively defined in terms of architectural elements: a component failure can cause an error which can propagate to the architectural level boundary and result in system failure. Finally, it should be pointed out that there are concepts which are understood by the two communities in a very similar way, the most important ones being the concepts of components, interfaces and configuration, as well as the idea of using specialised styles and patterns to help system developers.

B. Agenda for Future Research

This section will describe our view of the current and future directions of research in fault tolerant software architectures.

Service-oriented and Dynamic Architecture. Service-Oriented Computing is considered to be a challenging computing paradigm that utilizes services as fundamental elements for developing applications. Service-oriented architecture (SOA) is a component model that links different functional units of an application, called services, through well-defined interfaces and contracts between them.

One of the defining characteristics of a SOA is its dynamicity. In the dynamic SA paradigm (usually referred to as DSA), architectures evolve dynamically, either because the existing components/services are modified, replaced or removed, or because new components are plugged in (as in, for instance, Service Oriented Architectures, publish/subscribe architectures, peer to peer distributed architectures). These run-time modifications may arise in response to the need to recover from functional or performance malfunctions, or may also be the natural consequence of the system evolution cycle.

In a service oriented architecture, service composition requires dynamic reconfiguration of services, and the SA, providing the reference model, should allow the system to evolve with new components, defining the way in which these can interact with the existing structure.

There is a growing amount of research dealing with fault tolerance in SOA, but only one of the surveyed approaches explicitly deals with SOA [61]. The recent work has, however, mainly focused on the design and

implementation phases, or middleware construction, rather than investigating how to architect a fault tolerant SOA system. This area can clearly benefit from introducing reasoning at higher levels of abstraction. For example, architectural patterns and styles for designing fault tolerant architectures (see Figure 5, the “FT style” column) could be revised so as to fit SOA; dynamically reconfigurable fault tolerant SA approaches (see Figure 5) and approaches addressing the “Run-time FT” topic could be adjusted for dealing with SOA reconfiguration needs.

Software Product Line Architectures. A software product line architecture (SPLA) [62] precisely captures, in a single specification, the overall architecture of a suite of closely-related products (rather than specifying the architecture of a single software system). The techniques for doing so are rooted in the disciplines of SA and configuration management, and focus on capturing the mandatory elements (which are present in the architecture of each and every product) and the variation points (which define the dimensions along which the architectures of the individual products can differ). A single product line architecture may have many variation points that are often orthogonal to each other: as a result, a considerable number of the product architectures can be formed by a single product line architecture.

This survey has identified only one approach which explicitly refers to “*architectures of architectures*”, presented in [63]. Being able to specify and then analyse a FT product line architecture will make it possible to improve the dependability of the entire family of architectures.

Model Driven Architecture and Explicit Resilience. Shifting the focus of software development from coding to modelling is one of the main achievements of Model-Driven Architecture [64] (MDA), which separates the application logic from the underlying platform technology and gives them precise semantic models. Consequently, models are the primary artefacts retained as the first class entities that can be manipulated using automated model transformations, in order to move from models to implementation. Software Architecture plays a fundamental role in MDA, and the current research in the SA community has put considerable effort in understanding how to bridge the gap between requirements and software architecture [65], [66], and between software architecture and coding [23], [52], [24].

In the dependability community, the need for explicitly dealing with fault tolerance (i.e., resilience) concerns during the entire life cycle has been recently recognised as one of the main approaches to ensuring the overall system dependability [12], [59], [67]. Fault tolerance ontologies are being proposed as part of the ReSIST Network of Excellence [67] to help capture the fault tolerance concerns at each development step and bridge the gap between these steps.

Only few of the surveyed approaches deal with model-based specifications of FT SA and code generation from models (see Figure 5, the “Specification” and “Specification and Coding” columns), and only [63] makes an explicit reference to MDA. Many approaches, instead, cover the topic of fault tolerance during the software development process (see Figure 5, the “FT Software Process” column).

Introducing new FT SA approaches which explicitly take the MDA principles into account will simplify the

specification and analysis of architectures and make the overall development more effective by avoiding the typical contamination of the architectural specification process by implementation considerations.

Self-healing Systems. In the last few years there has been a growing interest in the areas directly related and partially overlapping with fault tolerance, such as system self-healing, self-adaptation and self- management. Software-intensive self-healing systems are capable of self-managing through run-time adaptation to changes which could put the system at risk, including alterations of the available resources, dynamically changing user's needs, system intrusions or faults, and changes in the system environment. A self-healing system must configure and reconfigure itself, continually tuning and optimising its own behaviour, protecting and recovering itself from emerging threats, while keeping its complexity hidden from the user [68]. The promise of self-healing for our area is that it could allow us to automatically reconfigure the FT architecture when unforeseen and abnormal events happen at run-time. Even so, while this research is clearly relevant to fault tolerance, the relationship between fault tolerance and self- healing needs to be better understood.

Innovative Technologies and Tools. The most effective way of applying academic research is developing advanced technologies and tools. However, while integrating fault tolerance at the architectural level has been the focus of extensive theoretical exploration, not much has been produced in terms of mature technologies and tools (see Figure 5, the "Tool Support" column). In fact, most of the existing tools are at the prototype stage. New industrial-strength technologies and tools are needed to allow practitioners to effectively and efficiently deal with fault tolerance at the various phases of the development process. These should include fully-specified architectural languages which explicitly support error detection, error recovery and fault handling, as well as tools that enable the verification and validation of fault tolerance-specific properties. The focus should be on assessing the achieved level of fault tolerance and evaluating the dependability properties of the fault tolerant system. A very promising example of this is the ongoing work on the Architecture Analysis & Design Language - AADL. Defined by the Society of Automotive Engineers as the aerospace standard AS5506 [69], this language is supported by an Eclipse-based development environment. It has been extended by an Error ModelAnnex [70], to be used for specifying various error models of components and connections, as well as the appropriate redundancy management and risk mitigation methods chosen during system architecting. These features enable qualitative and quantitative assessments of such system dependability properties as reliability and availability.

Dealing with Dependability Means. Dependability is an integrated concept encompassing a variety of attributes, including availability, reliability, safety, integrity and maintainability. Generally, there are four means to be employed to attain dependability: fault prevention, fault tolerance, fault removal and fault forecasting. Clearly, in practice a combination of these needs to be applied to ensure the required dependability level. It is important to remember that all these activities are built around the concept of faults: where possible, faults are prevented or eliminated by using appropriate development and verification techniques, while the remaining ones are tolerated

at runtime to avoid system failures and estimated to help predict their consequences. As shown in this survey, in some cases fault removal (by testing or model-checking) and fault tolerance have been applied together at the architectural level. Certain types of analysis (such as performance and simulation) have also been applied to verify the quality of fault tolerant architectures. It appears that there is a need for deeper integration of these complementary techniques, in order to meet the increasing dependability needs of increasingly complex systems.

VIII. CONCLUSIONS

The primary object of this survey is to study the ways in which current research on architecting fault tolerant systems contributes to cross-fertilisation between the software architecture and fault tolerance communities. Yet since the two communities continue, for many technical and non-technical reasons, to address specific issues from different perspectives, the survey starts with analysing the existing approaches using two separate sets of parameters, one to focus on the architecture- and the other on the dependability-specific concepts. This should allow researchers in either domain to understand more clearly how the approaches used in the other fit in with their own investigation. The survey then goes on to draw some conclusions based on an integrated view, which will contribute to developing a common language that will enable software architects and fault tolerance experts to work together. The authors then provide their personal view, based on the data gathered, of what can be expected to happen in this area in the near future.

While there have been several workshops, tutorials and other events held which address related issues (e.g., [1], [2], [13], [71], [22]), this survey goes much further by summarising the existing body of work and analysing the major trends in the area. We hope our work will help to bring the researchers from the two communities together so that they can come up with even more integrated and practical solutions.

SHORT BIOS

Henry Muccini: I am currently an Assistant Professor at the University of L'Aquila (since 2002). I received my PhD degree in Computer Science from the University of Rome - La Sapienza - in 2002, and I have been visiting professor at Information & Computer Science, University of California, Irvine, in 2002.

My main research interests are on **using software architectures for producing quality software**. In this direction, I have been investigating how software architectures can be used for the verification and validation of complex (and dynamically evolving) software systems. More specifically, I have been researching methods for **testing, analysing, and monitoring** software systems based on their software architecture, as well as approaches and languages to *describe* architectures to improve their testability and ability to be validated. In these topics, I have been organizing various workshops and international meetings, and co-edited two books. I am currently chairing the PC of QSIC 2012 and Euromicro SEAA 2012.

Alexander (Sascha) Romanovsky: I am a Professor in the Centre for Software Reliability. I received a PhD degree in Computer Science from St. Petersburg State Technical University. I have been with this University from 1984 until 1996, doing research and teaching. In 1991 I worked as a visiting researcher at ABB Ltd Computer Architecture Lab Research Center, Switzerland, then I was a visiting fellow at Istituto di Elaborazione della Informazione, CNR, Pisa, Italy, a post-doctoral fellow with the Department of Computing Science, University of Newcastle upon Tyne.

My main research interests are **system dependability, fault tolerance, software architectures, exception handling, error recovery, system structuring** and **verification of fault tolerance**. I am currently the coordinator of the major FP7 Integrated Project on Industrial Deployment of System Engineering Methods Providing High Dependability and Productivity (DEPLOY, 2008-2012). I am a co-investigator of the TrAmS EPSRC/UK platform grant on Trustworthy Ambient Systems (2007-2011) and the Principle investigator of the new EPSRC/RSSB research project SafeCap on Overcoming the Railway Capacity Challenges without Undermining Rail Network Safety (2011-2014).

ACKNOWLEDGMENT

Alexander Romanovsky is supported by the ICT DEPLOY Integrated Project and the EPSRC/UK TrAmS-2 platform grant. Henry Muccini acknowledges the National Research Fund Luxembourg under the MA6 initiative and the University of Luxembourg for their support of a preliminary study contributing to this work.

The authors are indebted to Patricia Lago and Andrea Florio for conducting the initial systematic literature review on the topic of architecting fault tolerant systems. We also acknowledge the help of Patricia Lago in the definition of the systematic literature review protocol.

REFERENCES

- [1] WADS workshops, "WADS: ICSE and DSN Workshops on Architecting Dependable Systems, years 2002-2009," <http://www.cs.kent.ac.uk/events/conf/2009/wads/>, 2009.
- [2] EFTS workshops, "EFTS: Int. Workshop on Engineering of Fault Tolerant Systems, years 2006-2007," <http://efts2007.uni.lu>, 2007.
- [3] SERENE Workshops, "SERENE: International Workshop on Software Engineering for Resilient Systems, years 2008-2011," <http://serene2011.uni.lu/>, 2011.
- [4] F. Cristian, *Dependability of Resilient Computers*. Blackwell Scientific Publications, Oxford, 1989, ch. Exception handling, pp. 68–97.
- [5] J.-L. Lions, *Ariane 5 flight 501 failure. Technical report*. ESA/CNES, 1996.
- [6] Natural Resources, Canada, *Interim Report: Causes of the August 14th Blackout in the United States and Canada. Canada-U.S. Power System Outage Task Force*, November 2003.
- [7] D. Reimer and H. Srinivasan, "Analyzing Exception Usage in Large Java Applications," in *In Proceedings of the ECOOP 2003 Workshop on Exception Handling in Object-Oriented Systems: Towards Emerging Application Areas and New Programming Paradigms*, A. Romanovsky, C. Domy, J. L. Knudsen, and A. Tripathi, Eds. TR 03-028, Department of Computer Science, University of Minnesota, Minneapolis, USA, 2003.
- [8] M. Bruntink, A. van Deursen, and T. Tourwé, "Discovering faults in idiom-based exception handling," in *Proceedings of the 28th international conference on Software engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 242–251. [Online]. Available: <http://doi.acm.org/10.1145/1134285.1134320>
- [9] P. Sacramento, B. Cabral, and P. Marques, "Unchecked Exceptions: Can the Programmer be Trusted to Document Exceptions?" in *International Conference on Innovative Views of .NET Technologies (IVNET'06)*, Springer-Verlag, Ed., October 2006.

- [10] B. Cabral and P. Marques, "Exception handling: A field study in java and .net." in *ECOOP'07*, 2007, pp. 151–175.
- [11] R. de Lemos and A. Romanovsky, "Exception Handling in the Software Lifecycle," vol. vol. 16, numb. 2, no. 2. CRL Publishing, March 2001, pp. 167–181.
- [12] C. Rubira, R. de Lemos, G. Ferreira, and F. C. Filho, "Exception handling in the development of dependable component-based systems," *Softw. Pract. Exper.*, vol. 35, no. 3, pp. 195–236, 2005.
- [13] P. Pelliccione, H. Muccini, N. Guelfi, and A. Romanovsky, Eds., *Software Engineering of Fault Tolerant Systems*. World Scientific Publishing, Series on Software Engineering & Knowledge Engineering (Hardcover), June 2007.
- [14] C. Gacek and R. de Lemos, *Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective*. Springer-Verlag, 2006, ch. Architectural Description of Dependable Software Systems, pp. 127–142.
- [15] D. Garlan, "Software Architecture," in *Encyclopedia of Software Engineering*, John Wiley & Sons, 2001.
- [16] M. Bernardo and P. Inverardi, Eds., *Formal Methods for Software Architectures. Tutorial book on Software Architectures and Formal Methods*. LNCS 2804, 2003, vol. SFM-03:SA Lectures.
- [17] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, 2009.
- [18]
- [19] D. Srivastava and P. Narasimhan, "Architectural support for mode-driven fault tolerance in distributed applications," in *Proceedings of the 2005 workshop on Architecting dependable systems*, ser. WADS '05. New York, NY, USA: ACM, 2005, pp. 1–7. [Online]. Available: <http://doi.acm.org/10.1145/1082983.1083226>
- [20] QOSA Conferences, "Seventh International ACM Sigsoft Conference on the Quality of Software Architectures (QOSA 2011)," Boulder, Colorado, USA, June 20-24 2011.
- [21] ROSATEA workshops, "ROSATEA 2007: The Role Of Software Architecture in Testing and Analysis," <http://www.di.univaq.it/muccini/Rosatea2007/>, 2007.
- [22] ISARCS Symposium, "ISARCS 2011: Int. Symposium on Architecting Critical Systems," <http://www.isarcs.org/isarcs2011/>, 2011.
- [23] ArchJava, "ArchJava Project," <http://archjava.org/>, 2005.
- [24] Fujaba Project, "Fujaba Project," <http://www.fujaba.de/>, 2011, University of Paderborn, Software Engineering Group.
- [25] C. Hofmeister, R. Nord, and D. Soni, *Applied Software Architecture*. Addison-Wesley, 1998.
- [26] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice, second edition*, ser. SEI Series in Software Eng. Addison-Wesley Professional, 2003.
- [27] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. Dependable Sec. Comput.*, vol. 1, no. 1, pp. 11–33, 2004.
- [28] D. E. Perry, A. Romanovsky, and A. Tripathi, "Current trends in exception handling," *IEEE Trans. Softw. Eng.*, vol. 26, no. 10, pp. 921–922, 2000.
- [29] B. Randell, "System structure for software fault tolerance," *SIGPLAN Notes*, vol. 10, pp. 437–449, 1975.
- [30] A. Avizienis, "The N-Version Approach to Fault-Tolerant Software," *IEEE Transaction on Software Engineering*, vol. 11, no. 12, pp. 1491–1501, December 1985.
- [31] R. H. Campbell and B. Randell, "Error recovery in asynchronous systems," *IEEE Trans. Softw. Eng.*, vol. 12, pp. 811–826, August 1986. [Online]. Available: <http://portal.acm.org/citation.cfm?id=6223.6276>
- [32] J. Xu, B. Randell, A. B. Romanovsky, C. M. F. Rubira, R. J. Stroud, and Z. Wu, "Fault tolerance in concurrent object-oriented software through coordinated error recovery," in *Symp. on Fault-Tolerant Computing*, 1995, pp. 499–508.
- [33] N. Lynch, M. Merritt, W. Weihl, and A. Fekete, *Atomic Transactions*. Morgan Kaufmann Publishers, 1994.
- [34] T. Anderson and P. Lee, "Fault Tolerance: Principles and Practice, First Edition," *Prentice-Hall*, 1981.
- [35] H. Zhang, M. A. Babar, and P. Tell, "Identifying relevant studies in software engineering," *Inf. Softw. Technol.*, vol. 53, no. 6, pp. 625–637, Jun. 2011. [Online]. Available: <http://dx.doi.org/10.1016/j.infsof.2010.12.010>
- [36] A. Coker, V. Taylor, D. Bhaduri, S. Shukla, A. Raychowdhury, and K. Roy, "Multijunction fault-tolerance architecture for nanoscale crossbar memories," *Nanotechnology, IEEE Transactions on*, vol. 7, no. 2, pp. 202–208, march 2008.
- [37] Y. Brun, "Self-assembly for discreet, fault-tolerant, and scalable computation on internet-sized distributed networks," Ph.D. dissertation, University of Southern California, 2008.

- [38] B. Kitchenham, "Guidelines for performing systematic literature reviews in software engineering," Software Engineering Group, School of Computer Science and Mathematics, Keele University, and Department of Computer Science University of Durham Durham, UK, Tech. Rep. EBSE Technical Report EBSE-2007-01, July 2007.
- [39] R. Farenhorst and R. C. De Boer, *Knowledge Management in Software Architecture: State of the Art*. Springer Berlin Heidelberg, 2009, pp. 21–38. [Online]. Available: <http://archix1.nl/files/2009-Springer-AKStateOfTheArt.pdf>
- [40] D. E. Perry, A. A. Porter, and L. G. Votta, "Empirical studies of software engineering: a roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 345–355. [Online]. Available: <http://doi.acm.org/10.1145/336512.336586>
- [41] H. K. Wright, M. Kim, and D. E. Perry, "Validity concerns in software engineering research," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, ser. FoSER '10. New York, NY, USA: ACM, 2010, pp. 411–414. [Online]. Available: <http://doi.acm.org/10.1145/1882362.1882446>
- [42] A. Florio, "A systematic literature review on fault tolerant systems," Master's thesis, European Double Degree Master Programme in Global Software Engineering (GSEEM), VU University Amsterdam and University of LAquila, December 2010.
- [43] D. Garlan, "Formal Modeling and Analysis of Software Architecture: Components, Connectors, and Events," in *Formal Methods for Software Architectures*. Lecture Note in Computer Science, 2804, 2003, pp. 1–24.
- [44] H. Giese, Ed., *Architecting Critical Systems, First International Symposium, ISARCS 2010, Prague, Czech Republic, June 23-25, 2010, Proceedings*, ser. Lecture Notes in Computer Science, vol. 6150. Springer, 2010.
- [45] M. Shaw and P. Clements, "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems," in *COMPSAC97, 21st Int. Computer Software and Applications Conference*, 1997.
- [46] A. F. Garcia, C. M. F. Rubira, A. B. Romanovsky, and J. Xu, "A comparative study of exception handling mechanisms for building dependable object-oriented software." *Journal of Systems and Software*, vol. 59, no. 2, pp. 197–222, 2001.
- [47] C2 Style, "The C2 Architectural Style," On-line at: <http://www.ics.uci.edu/pub/arch/c2.html>, 2000.
- [48] L. Baresi, R. Heckel, S. Thöne, and D. Varró, "Style-based modeling and refinement of service-oriented architectures," *Software and System Modeling*, vol. 5, no. 2, pp. 187–207, 2006.
- [49] J. Magee and J. Kramer, *Concurrency: State models and Java Programs*, 2nd ed. Wiley publisher, July 2006.
- [50] D. Jackson, "The Alloy Community portal," <http://alloy.mit.edu/>, 2011.
- [51] A. Capozucca, N. Guelfi, P. Pelliccione, A. Romanovsky, and A. Zorzo, "Caa-drip: a framework for implementing coordinated atomic actions," in *Software Reliability Engineering, 2006. ISSRE '06. 17th International Symposium on*, nov. 2006, pp. 385–394.
- [52] H. Baumeister, F. Hacklinger, R. Hennicker, A. Knapp, and M. Wirsing, "A component model for architectural programming." *Electr. Notes Theor. Comput. Sci.*, vol. 160, pp. 75–96, 2006.
- [53] K. H. Kim and A. Kavianpour, "A distributed recovery block approach to fault-tolerant execution of application tasks in hypercubes," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 1, pp. 104–111, 1993.
- [54] J.-C. Laprie, J. Arlat, C. Beounes, and K. Kanoun, "Definition and analysis of hardware- and software-fault-tolerant architecture," *IEEE Computer*. *IEEE Press*, vol. 23, no. 7, pp. pp. 39–51, 1990.
- [55] J. B. Goodenough, "Exception handling: issues and a proposed notation," *Commun. ACM*, vol. 18, no. 12, pp. 683–696, 1975.
- [56] J. Stafford and A. Wolf, "Architecture-Level Dependence Analysis for Software Systems," *International Journal of Software Engineering and Knowledge Engineering*, vol. 11, no. 4, pp. 431–453, August 2001.
- [57] D. L. Parnas, "On the criteria to be used in decomposing systems into modules." *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.
- [58] A. Avizienis, "Toward systematic design of fault-tolerant systems," *Computer*, vol. 30, no. 4, pp. 51–58, 1997.
- [59] M. Kaaniche, J.-C. Laprie, and J.-P. Blanquart, "A framework for dependability engineering of critical computing systems," *Safety Science*, vol. 40, no. 9, pp. 731–752, December 2002.
- [60] A. Romanovsky, "A looming fault tolerance software crisis?" *ACM SIGSOFT SE Notes*, vol. 32, no. 2, 2007.
- [61] R. de Lemos, *Architecting Dependable Systems: v. 3*. Springer-Verlag Berlin and Heidelberg GmbH, LNCS 3549, 2005, ch. Architecting Web Services Applications fo Improving Availability, pp. 69–91.
- [62] J. Bosch, *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. ACM Press/Addison-Wesley Publishing Co., 2000.

- [63] N. Guelfi, R. Razavi, A. Romanovsky, and S. Vandenbergh, "DRIP Catalyst: An MDE/MDA Method for Fault-tolerant Distributed Software Families Development," in *OOPSLA and GPCE Workshop on Best Practices for Model Driven Software Development*, 2004.
- [64] Object Management Group (OMG), "OMG/Model Driven Architecture - A Technical Perspective," 2001, oMG Document: ormsc/01-07-01.
- [65] B. Nuseibeh, "Weaving Together Requirements and Architectures," *IEEE Computer*, vol. 34, no. 3, pp. 115–117, March 2001.
- [66] STRAW Workshop, "STRAW '03: Second Int. Workshop From Software Requirements to Architectures, May 09, 2003, Portland, Oregon, USA," 2003.
- [67] ReSIST Project, "ReSIST Network of Excellence project: Relisience for Survivability in IST," <http://www.resist-noe.org/>, 2009.
- [68] SEAMS Symposium, "The 6th int. symposium on software engineering for adaptive and self-managing systems (seams 2011)," May 2011.
- [69] AADL, "SAE Architecture Analysis & Design Language (AADL), Society of Automotive Engineers, SAE Aerospace. SAE-AS5506, 2004," 2004.
- [70] AADL Annex, "SAE Architecture Analysis & Design Language (AADL), Annex E: Error Model Annex. Society of Automotive Engineers, SAE Aerospace. SAE-AS5506/1. 2006," 2006.
- [71] H. Muccini, P. Pelliccione, and A. Romanovsky, "Architecting Fault Tolerant Systems (tutorial)," in *Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture, WICSA 2007*. Mumbai, India: IEEE CS, January 2007.

APPENDIX: LIST OF SELECTED PAPERS

#	Authors	Title	In	Date
P01	Gagliardi, Rajkumar, and Sha	Designing for evolvability: building blocks for evolvable real-time systems	In Proc. RTAS '96: 2nd IEEE Real-Time Technology and Applications Symposium, pages 100–109.	1996
P02	Cook and Dage	Highly reliable upgrading of components	In Proc. ICSE '99: Proc. of the 21st Int. Conference on Software Engineering. IEEE Computer Society, Los Alamitos, CA, USA, 203–212.	1999
P03	Issamy and Banatre	Architecture-based Exception Handling	In Proc. HICSS '01: 34th Annual Hawaii Int. Conference on System Sciences (HICSS-34)-Volume 9. IEEE Computer Society, Washington, DC, USA, 182–193.	2001
P04	Beder, Romanovsky, Randell, and Rubira	On Applying Coordinated Atomic Actions and Dependable Software Architectures in Developing Complex Systems	In Proc. ISORC'01: 4th IEEE Int. Symposium on Object-Oriented Real-time Distributed Computing. Magdeburg, Germany, 102–111.	2001
P05	de Lemos	Describing Evolving Dependable Systems using Co-operative software Architectures	In Proc. ICSE '01: IEEE Int. Conference on Software Maintenance. IEEE Computer Society, Washington, DC, USA, 320–329.	2001
P06	Garcia and Rubira	An Architectural-based Reflective Approach to Incorporating Exception Handling into Dependable Software	In Advances in Exception Handling Techniques, Alexander B. Romanovsky, C. Dony, J. Lindskov Knudsen, and A. Tripathi (Eds.), Springer-Verlag, London, UK, 189–206.	2001
P07	Ferreira, Rubira, and de Lemos	Explicit Representation of Exception Handling in the Development of Dependable Component-based Systems	In Proc. HASE'01: the 6th IEEE Int. Symposium on High-Assurance Systems Engineering. Special Topic: Impact of Networking (HASE '01). IEEE Computer Society, Washington, DC, USA, 182–193.	2001
P08	Romanovsky	Exception handling in component-based system development	In Proc. COMPSAC '01: 25th Int. Computer Software and Applications Conference on Invigorating Software Development. IEEE Computer Society, Washington, DC, USA, 580–589.	2001
P09	Rakic and Medvidovic	Increasing the confidence in off-the-shelf components: a software connector-based approach	In Proc. SSR '01: the 2001 Symposium on Software Reusability: putting software reuse in context. ACM, New York, NY, USA, 11–18.	2001
P10	Sha	Using simplicity to control complexity	In <i>IEEE Software</i> 18, 4 (July 2001), 20–28.	2001
P11	de C. Guerra, Rubira, and de Lemos	An Idealized Fault-Tolerant Architectural Component	In Proc. WADS '02: ICSE 2002 Workshop on Architecting Dependable Systems.	2002
P12	de Lemos, Gacek, and Romanovsky	Tolerating Architectural Mismatches	In Proc. WADS '02: ICSE 2002 Workshop on Architecting Dependable Systems, 175–194	2002
P13	Liu and Richardson	RAIC: Architecting Dependable Systems through Redundancy and Just-in-Time Testing	In Proc. WADS '02: ICSE 2002 Workshop on Architecting Dependable Systems.	2002
P14	de Lima Filho, de C. Guerra, and Rubira	FaTC2: An Object-Oriented Framework for Developing Fault-Tolerant Component-Based Systems	In Proc. WADS '03: ICSE 2003 Workshop on Architecting Dependable Systems, 13–18.	2003
P15	de Lima Filho, de C. Guerra, and Rubira	An Architectural-Level Exception-Handling System for Component-Based Applications	In Dependable Computing, R. de Lemos, T. Weber and J. Camargo (Eds.). Lecture Notes in Computer Science, 2003, Volume 2847/2003, 321–340.	2003
P16	de Lemos, Gacek, and Romanovsky	Architectural Mismatch Tolerance	In Architecting Dependable Systems, R. de Lemos, C. Gacek, and A. Romanovsky (Eds.), LNCS vol. 2677. Springer, 175–194.	2003
P17	de C. Guerra, Rubira, Romanovsky, and de Lemos	A Fault-Tolerant Software Architecture for COTS-Based Software Systems	In Proc. ESEC/FSE '11: in Proc. of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT Int. Symposium on Foundations of Software Engineering. ACM, New York, 97–115.	2004
P18	de Lemos	Analysing failure behaviours in component interaction.	In <i>Journal of System and Software</i> , 71, 1–2, 97–115.	2004
P19	Guelfi, Razavi, Romanovsky, and Vandenbergh	DRIP Catalyst: An MDE/MDA Method for Fault-tolerant Distributed Software Families Development	In Proc. OOPSLA and GPCE Workshop on Best Practices for Model Driven Software Development, October, Vancouver, Canada.	2004
P20	Parchas and de Lemos	An Architectural Approach for Improving Availability in Web Services	In Proc. WADS '04: ICSE 2004 Workshop on Architecting Dependable Systems, 37–41.	2004
P21	Alda and Creemers	Strategies for Component-based Self-Adaptability Model in Peer-to-Peer Architectures	In Proc. CBSE'7: 4th Int. Symposium on Component-based Software Engineering, LNCS 3054. Springer, Edinburgh, Scotland, 59–67.	2004

Fig. 8. Selected Papers (Page 1/4)

#	Authors	Title	In	Date
P22	de C. Guerra, Rubira, Romanovsky, and de Lemos	A Dependable Architecture for COTS-Based Software Systems using Protective Wrappers	In Architecting Dependable Systems II. R. de Lemos, C. Gacek, and A. Romanovsky. LNCS vol. 3069. Springer, 147--170.	2004
P23	Porcarelli, Castaldi, Di Giandomenico, Bondavalli, and Inverardi	A framework for reconfiguration-based fault tolerance in distributed systems	In Architecting Dependable Systems II. R. de Lemos, C. Gacek, and A. Romanovsky. LNCS vol. 3069. Springer, 343--358.	2004
P24	Bondavalli, Chiaradonna, Cotroneo, and Romano	Effective fault treatment for improving the dependability of COTS- and legacy-based applications	In <i>IEEE Transactions on Dependable and Secure Computing</i> , 1, 4, 223--237.	2004
P25	Das and Woodside	Dependability Modeling of Self-Healing Client-Server Applications	In Architecting Dependable Systems II. R. de Lemos, C. Gacek, and A. Romanovsky. LNCS vol. 3069. Springer, 128--165.	2004
P26	Laibinis and Troubitsyna	Fault Tolerance in a Layered Architecture: A General Specification Pattern in B	In Proc. SEFM'04: Second Int. Conference on Software Engineering and Formal Methods, 346--355.	2004
P27	Rubira, de Lemos, Ferreira, and de Lima Filho	Exception handling in the development of dependable component-based systems	In <i>Softw. Pract. Experience</i> 35, 3, 195--236.	2005
P28	de Lima Filho, da S. Brito, and Rubira	A framework for analyzing exception flow in software architectures	In Proc. WADS '05: ICSE 2005 Workshop on Architecting Dependable Systems, 37-41.	2005
P29	Feng, Huang, Zhu, and Mei	Exception Handling in Component Composition with the Support of Middleware	In Proc. SEM '05: Fifth International Workshop on Software Engineering and Middleware. ACM, New York, NY, USA, 90-97.	2005
P30	da S. Brito, Rocha, de Lima Filho, Martins, and Rubira	A method for modeling and testing exceptions in component-based software development	In Dependable Computing, C. Maziero, J. G. Silva, A. Andrade, and F. A. Silva (Eds.), Lecture Notes in Computer Science, 2005, Volume 3747/2005, 61-79.	2005
P31	de Lima Filho, da S. Brito, and Rubira	Modeling and Analysis of Architectural Exceptions	In Proc. FM'05: Workshop on Rigorous Engineering of Fault Tolerant Systems. 18-22 July 2005, University of Newcastle upon Tyne, UK, pages 112-121.	2005
P32	de Lemos	Architecting Web Services Applications to Improving Availability	In Architecting Dependable Systems III. R. de Lemos, C. Gacek, and A. Romanovsky (Eds.), Springer-Verlag Berlin and Heidelberg GmbH, LNCS 3549, pages 69--91.	2005
P33	de Lemos, de C. Guerra, and de Lemos	A Fault-Tolerant Architectural Approach for Dependable Systems	In <i>IEEE Software</i> , 23, 2, Special Issue on Software Architectures, pages 80--87.	2006
P34	de Lemos	Idealised Fault Tolerant Architectural Element	In Proc. DSN '06: Workshop on Architecting Dependable Systems (WADS06), Supplemental Proceedings of the 36th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pages 76-81.	2006
P35	de Lemos	Architectural Reconfiguration using Coordinated Atomic Actions	In Proc. SEAMS '06: ICSE 2006 Int. Workshop on Self-adaptation and self-managing Systems (SEAMS). ACM, New York, NY, USA, 44-50.	2006
P36	de Lima Filho, da S. Brito, and Rubira	Specification of Exception Flow in Software Architectures	In <i>Journal of Systems and Software</i> - Special Issue on Architecting Dependable Systems. Volume 79, Issue 10, October 2006, Pages 1397-1418	2006
P37	Magee and Maibaum	Towards specification, modelling and analysis of fault tolerance in self managed systems	In SEAMS '06: Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems. ACM Press, pages 30--36.	2006
P38	Bucchiarone, Muccini, and Pelliccione	Architecting Fault-tolerant Component-based Systems: from requirements to testing	In Proc. VODCA '06: second workshop on Views On Designing Complex Architectures. In Electronic Notes in Theoretical Computer Science 168 (ENTCS), Bertinoro, Italy. Pages 77-	2006
P39	de Lima Filho, da S. Brito, and Rubira	Reasoning about Exception Flow at the Architectural Level	In Rigorous development of Complex Fault Tolerant Systems. Lecture Notes in Computer Science, Vol. 4157. M. Butler, C. Jones, A. Romanovsky, and E. Troubitsyna (Eds.) 403 p.	2006
P40	da S. Brito, Lemos, and Rubira	Verification of exception control flows and handlers based on architectural scenarios	In Proc. HASE '08: the 11th IEEE High Assurance Systems Engineering Symposium. IEEE Computer Society, Washington, DC, USA, 177--186.	2008
P41	Wellings, Beus-Dukic, and Powell	Real-time scheduling in a generic fault-tolerant architecture	In Proc. RTSS '98: the IEEE Real-Time Systems Symposium. IEEE Computer Society, Washington, DC, USA, 390-.	1998
P42	de Lemos	On architecting software fault tolerance using abstractions	In <i>Electron. Notes Theor. Comput. Sci.</i> , 236, 21--32.	2009
P43	Mahdian, Rafeh, and Rahmani	Modeling fault tolerant services in service-oriented architecture	In Proc. TASE '09: Third IEEE Int. Symposium on Theoretical Aspects of Software Engineering. IEEE Computer Society, Washington, DC, USA, 319--320.	2009

Fig. 9. Selected Papers (Page 2/4)

#	Authors	Title	In	Date
P44	Michotte, France, and Fleurey	Modeling and integrating aspects into component architectures	In Proc. EDOC '07: Proc. of the 11th IEEE Int. Enterprise Distributed Object Computing Conference. IEEE Computer Society, Washington, DC, USA, 181.	2007
P45	Yuan, Dong, and Sun	Modeling and customization of fault tolerant architecture using object-z/xvcl architectures	In Proc. APSEC '06: the XIII Asia Pacific Software Engineering Conference. IEEE Computer Society, Washington, DC, USA, 209--216.	2006
P46	de C. Guerra, Romanovsky, and de Lemos	Integrating cots software components into dependable software architectures	In Proc. ISORC '03: the Sixth IEEE Int. Symposium on Object-Oriented Real-Time Distributed Computing. IEEE Computer Society, Washington, DC, USA, 139--.	2003
P47	Pereira and de Melo	A formal architectural model for exception handling	In Proc. SAC '08: the 2008 ACM symposium on Applied computing. ACM, pages 114--	2008
P48	Xu, Randell, and Romanovsky	A generic approach to structuring and implementing complex fault-tolerant software	In Proc. ISORC '02: Fifth IEEE Int. Symposium on Object-Oriented Real-Time Distributed Computing, 2002, pages 207 --214.	2002
P49	da S. Brito, de Lemos, Martins, Moraes, and Rubira	Architectural-based validation of fault-tolerant software	In Proc. LADC '09: the Fourth Latin-American Symposium on Dependable Computing. IEEE Computer Society, Washington, DC, USA, 103--110. Pages 103-110.	2009
P50	Domokos and Majzik	Design and analysis of fault tolerant architectures by model weaving	In Proc. HASE '05: the Ninth IEEE Int. Symposium on High-Assurance Systems Engineering. IEEE Computer Society, Washington, DC, USA, 15--24.	2005
P51	Sotirovski	Towards fault-tolerant software architectures	In Proc. WICSA: Working IEEE/IFIP Conference on Software Architecture. IEEE Computer Society, Washington, DC, USA, 7.	2001
P52	Chan and Wu	Architectural level support for dynamic reconfiguration and fault tolerance in component-based distributed software	In Proc. ICPADS '02: 9th Int. Conference on Parallel and Distributed Systems. IEEE Computer Society, Washington, DC, USA, 251.	2002
P53	Brun and Medvidovic	Fault and adversary tolerance as an emergent property of distributed systems' software architectures	In Proc. EFTS '07: the 2007 workshop on Engineering fault tolerant systems. ACM, New York, NY, USA	2007
P54	Harrison and Ageriou	Incorporating fault tolerance tactics in software architecture patterns	In Proc. SERENE '08: the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems. ACM, New York, NY, USA, 9--18.	2008
P55	Liu, Wang, Gopalakrishnan, He, Sha, Ding, and Lee	Ortega: An efficient and flexible online fault tolerance architecture for real-time control systems	In <i>IEEE Transactions on Industrial Informatics</i> , vol. 4, num. 4 (NOV), 213--224.	2008
P56	Issarny and Zaras	Software architecture and dependability	In Formal Methods for Software Architectures, M. Bernardo and P. Inverardi (Eds.) Lecture Notes in Computer Science, vol. 2804, 259--285.	2003
P57	Xie, Manimaran, Vittal, Phadke, and Centeno	Information architecture for future power systems and its reliability analysis	In <i>IEEE Transactions on Power Systems</i> , vol. 17, num. 3 (AUG), 857--863.	2002
P58	da S. Brito, de Lemos, Rubira, and Martins	Architecting fault tolerance with exception handling: Verification and validation	In <i>Journal of Computer Science and Technology</i> , vol. 24, num. 2 (march), 212--237.	2009
P59	Ren, Bakken, Courtney, Cukier, Karr, Rubel, Sabnis, Sanders, Schantz, and Seri	Aqua: An adaptive architecture that provides dependable distributed objects	In <i>IEEE Trans. Comput.</i> , vol. 52, num. 1 (Jan. 2003), pages 31--50.	2003
P60	Das and Woodside	Analyzing the effectiveness of fault-management architectures in layered distributed systems	In <i>Performance Evaluation</i> , vol. 56, num 1-4 (March 2004), pages 93--120.	2004
P61	Garcia, Beder, and Rubira	A unified meta-level software architecture for sequential and concurrent exception handling	In <i>The Computer Journal</i> , vol. 44, num. 6, pages 569--587.	2001
P62	Fahad, Nadeem, and Lyu	A survey of fault tolerant CORBA systems	In Proc. OTM '07: the 2007 OTM Confederated int. conference on On the move to meaningful internet systems: CoopIS, DOA, ODBASE, GADA, and IS - Volume Part I, Robert Meersman and Zahir Tari (Eds.), Vol. Part I, Springer-Verlag, Berlin.	2009
P63	da S. Brito, de Lemos, and Rubira	Development of fault-tolerant software systems based on architectural abstractions	In Proc. ECSA '08: the 2nd European conference on Software Architecture, Ron Morrison, Dharini Balasubramaniam, and Katrina Falkner (Eds.), Springer-Verlag, Berlin, Heidelberg, 131-147.	2008
P64	de Lemos	Architectural fault tolerance using exception handling	In Architecting Dependable Systems IV, R. de Lemos, C. Gacek, and A. Romanovsky (Eds.), Lecture Notes in Computer Science, Vol. 4615. Springer-Verlag, Berlin,	2007

Fig. 10. Selected Papers (Page 3/4)

#	Authors	Title	In	Date
P66	Goldberg and Stroud	Adaptive fault-tolerant systems and reflective architectures	In Proc. of the Workshops on Object-Oriented Technology (ECOOP '97), Jan Bosch and Stuart Mitchell (Eds.), Springer-Verlag, London, UK, 80-88.	1998
P67	da S. Brito, de Lemos, Martins, and Rubira	Architecture-centric fault tolerance with exception handling	In Dependable Computing, A. Bondavalli, F. Brasileiro, and S. Francisco (Eds.), Springer LNCS 4746, pages 75-94	2007
P68	Dragoni and Gaspari	An object based algebra for specifying a fault tolerant software architecture	In <i>Journal of Logic and Algebraic Programming</i> , Special Issue on Process Algebra and System Architecture, vol. 63, num. 2 (MAY-JUN), 271--297.	2005
P69	Fabre and Perennou	A metaobject architecture for fault-tolerant distributed systems: The friends approach	In <i>IEEE Trans. Comput.</i> , vol. 47, num. 1, 78--95.	1998
P70	Harrison, Avgeriou, and Zdun	On the impact of fault tolerance tactics on architecture patterns	In Proc. SERENE '10: 2nd Int. Workshop on Software Engineering for Resilient	2010
P71	Das and Woodside	Analyzing the effectiveness of fault-management architectures in layered distributed systems	In <i>Performance Evaluation</i> , vol. 56, num. 1-4 (March), pages 93--120.	2004
P72	Li, Chen, Huang, Mei, and Chauvel	Selecting fault tolerant styles for third-party components with model checking support	In Proc. CBSE '09: the 12th International Symposium on Component-Based Software Engineering, Springer-Verlag, Berlin, Heidelberg, 69--	2009
P73	Salatge and Fabre	Fault tolerance connectors for unreliable web services	In Proc. DSN'07: the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. IEEE Computer Society, Washington, DC, USA, 51--60.	2007
P74	Yuan, Dong, Sun, and Basit	Generic fault tolerant software architecture reasoning and customization	In <i>IEEE Transactions on Reliability</i> , vol. 55, num. 3 (sept.), pages 421--435.	2006
P75	Seo, Malek, Edwards, Popescu, Medvidovic, Petrus, and Ravula	Exploring the role of software architecture in dynamic and fault tolerant pervasive systems	In Proc. SEPCASE '07: 1st Int. Workshop on Software Engineering for Pervasive Computing, Applications, Systems, and Environments. IEEE Computer Society, 9-	2007

Fig. 11. Selected Papers (Page 4/4)