Architecting resilient computing systems: A component-based approach for adaptive fault tolerance

Miruna Stoicescu, Jean-Charles Fabre, Matthieu Roy

Architecting Resilient Computing Systems: A Component-Based Approach for Adaptive Fault Tolerance

Miruna Stoicescu[1] Jean-Charles Fabre Matthieu Roy

LAAS-CNRS, Université de Toulouse, CNRS, INP, Toulouse, France

[1] EUMETSAT (European Organisation for the Exploitation of Meteorological Satellites)

ABSTRACT

Evolution of systems during their operational life is mandatory and both updates and upgrades should not impair their dependability properties. Dependable systems must evolve to accommodate changes, such as new threats and undesirable events, application updates or variations in available resources. A system that remains dependable when facing changes is called resilient. In this paper, we present an innovative approach taking advantage of component-based software engineering technologies for tackling the on-line adaptation of fault tolerance mechanisms. We propose a development process that relies on two key factors: designing fault tolerance mechanisms for adaptation and leveraging a reflective component-based middleware enabling fine-grained control and modification of the software architecture at runtime. We thoroughly describe the methodology, the development of adaptive fault tolerance mechanisms and evaluate the approach in terms of performance and agility.

1. INTRODUCTION

Dependable systems are becoming increasingly complex and their capacity to evolve in order to efficiently accommodate changes is a requirement of utmost importance. Changes have different origins, such as fluctuations in available resources, additional features requested by users, environmental perturbations... All changes that may occur during service life are rarely foreseeable when designing the system. Dependable systems must cope with changes while maintaining their ability to deliver trustworthy services and their required attributes, e.g., availability, reliability, integrity [1].

A lot of effort has been put into building applications that can adapt themselves to changing conditions. A rich body of research exists in the field of software engineering consisting of concepts, tools, methodologies and best practices for designing and developing adaptive software [2]. For instance, *agile software development* approaches [3] emphasize the importance of accommodating change during the lifecycle of an application at a reasonable cost, rather than striving to anticipate an exhaustive set of requirements. Component-based approaches [4] separate service interfaces from their actual implementation in order to increase flexibility, evolvability and reuse. Although dependable systems could benefit from these advancements in order to become more flexible and adaptive, very little has been done in this direction for now. In this paper, we describe an innovative approach for tackling on-line adaptation of dependability mechanisms that leverages component-based software engineering technologies. Component-based fault tolerance mechanisms can be easily updated through transition packages.

The paper is organized as follows. Section 2 presents the context and motivation of our work. Section 3.1 briefly describes the resilient system architecture we consider; next, we describe a set of Fault Tolerance Mechanisms (FTMs) and analyze transitions between them in Section 3.2. Section 4 presents the process of designing FTMs for subsequent adaptation. In Section 4.4, we detail the practical implementation of FTMs on top of a reflective component-based support and in Section 5, we describe the implementation of on-line transitions between FTMs. Next, we evaluate our approach in Section 6. In Section 7, we discuss related work and in Section 8 we present the lessons learned and conclude.

2. PROBLEM STATEMENT

Resilient computing refers to dependability in the presence of changes due to system evolution [5]. Our work focuses on *Fault Tolerance Mechanisms* (FTMs) that can be influenced by changes occurring in the system or in its environment. Ideally, fault-tolerant applications consist of two interconnected abstraction layers. The first one (the *base level*) contains the business logic that implements the functional requirements . The second one (the *meta level*) contains fault tolerant mechanisms and is attached to the first one through clearly identified hooks. As the development of a fault-tolerant application implies different roles/stakeholders (application developer(s), safety expert, integrator), separation of concerns is a key concept. Separation of concerns has some limit since fault tolerance strategies often depend on the semantics of the business logic. However, hooks can be defined to externalize some "application defined assertions" used to parameterize fault tolerance mechanisms. In any case, the concept of separation of concerns is essential to implement adaptive fault tolerance without impacting deeply the business logic.

Among the well-established and documented FTMs, e.g., [6, 7], the choice of an appropriate FTM for a given application depends on the values of several parameters. We identified three classes of parameters: fault tolerance requirements, application characteristics and available resources.

• **Fault tolerance requirements (FT).** This class of parameters contains the considered fault model. A fault model determines the category of FTM to be used (e.g., simple replication or diversification). Our fault model classification is well-known [1], dealing with crash faults, value faults and development faults. We focus on hardware faults (permanent and transient physical faults) but the approach can be extended to other FTMs.

• **Application characteristics (A).** The characteristics that have an impact on the choice of an FTM are application statefulness, state accessibility and determinism. State accessibility is essential for checkpointing-based fault tolerance strategies. Determinism refers here to behavioural determinism, i.e., the same inputs produce the same outputs in the absence of faults, mandatory for active replication.

• **Resources (R).** FTMs require resources in terms of CPU, battery life/energy, bandwidth, etc. A cost function can be associated to each FTM based on the values of such parameters. For a given set of resources, several mechanisms can be used with different trade-offs (e.g., more CPU, less bandwidth).

The first two parameters FT and A correspond to assumptions to be considered for the selection of an appropriate mechanism and to determine its validity. The resource dimension R states the amount of resources needed to accept a given solution according to system resources availability.

In practice, based on the values of (FT,A,R) set at design time, an FTM is attached to an application when the system is installed for the first time. As far as resilient computing [5] is concerned, the challenge lies in maintaining consistency between the FTM(s) and the non-functional requirements despite variations of the parameters at runtime, e.g.:

• new threats/faults and physical perturbations such as electromagnetic interferences trigger variations of FT;

• the introduction of new versions of applications or modules may trigger variations of A;

• resource loss or addition of hardware components imply variations of R.

If the FTM is inconsistent with the current (FT,A,R) values, it will most likely fail to tolerate the faults the system is confronted with. Therefore, a transition towards a new FTM is required before the current FTM becomes invalid, which is possible in Adaptive Fault Tolerance (AFT).

On-line adaptation of FTMs has attracted research efforts for some time now because dependable systems cannot be fully stopped for performing off-line modifications. However, most of the proposed solutions [8, 9, 10] tackle adaptation in a preprogrammed manner: all FTMs necessary during the service life of the system must be known and deployed from the beginning and adaptation consists in choosing the appropriate execution branch or tuning some parameters.

Nevertheless, predicting all events and threats that a system may encounter throughout its service life and making provisions for them is obviously impossible.

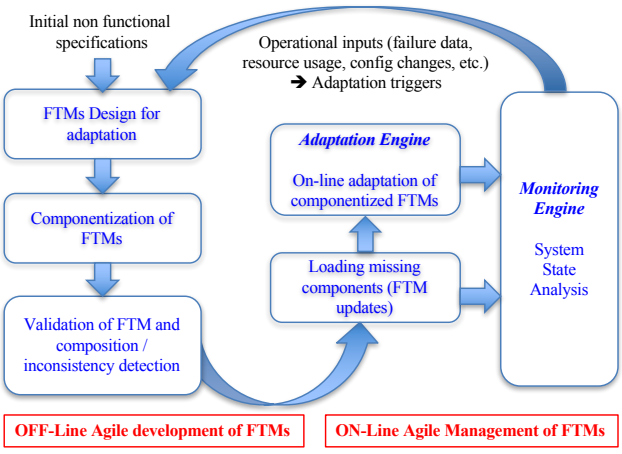


Figure 1: Adaptation process overview

This approach is of interest for long-lived space systems (satellites and deep-space probes) and for automotive applications regarding *over-the-air software updates*, a very important trend in the automotive industry today.

In this context, we propose an alternative to preprogrammed adaptation that we denote *agile adaptation of FTMs* (the term "agile" is inspired from agile software development). Agile adaptation of FTMs enables their systematic evolution: new FTMs can be designed off-line at any point during service life and integrated on-line in a flexible manner, with limited impact on the existing software architecture. Our approach for the agile adaptation of FTMs leverages advancements in the field of software engineering such as Component-Based Software Engineering (CBSE) technologies [4], Service Component Architecture [11] and Aspect-Oriented Programming [12]. Using such concepts and technologies, we design FTMs as brick-based assemblies (similar to "Lego" constructions) that can be methodically modified at runtime through *fine-grained* modifications affecting a limited number of bricks. This approach maximizes reuse and flexibility, contrary to monolithic replacements of FTMs found in related work, e.g., [8, 9, 10]. It is worth noting that, whatever the approach is (pre-programmed or agile), when the FTM evolution goes outside the foreseen boundaries of the FTM loaded into the system, the system may fail. The benefits of the proposed agile approach is to provide means to react more quickly and to simplify updates of loaded FTMs.

Summary of contributions. In this paper, after a short description of the resilient system architecture, we describe our methodology for agile adaptation of FTMs and its results, focussing on the following three key contributions:

• A "design for adaptation" approach of a set of FTMs, based on a generic execution scheme.

• A component-based architecture of the considered FTMs and fine-grained on-line transitions between them.

• Illustration of on-line FTM adaptation through several transition scenarios.

3. RESILIENT SYSTEM DESIGN

3.1 Architecture

The core objective of a resilient system is to guarantee the consistency of FTMs attached to applications according to major assumptions regarding the fault model and the application characteristics. First, this means identifying the link between applications and FTMs, and, more importantly, dynamically adapting FTMs according to operational conditions at runtime. As soon as FTMs are developed as assemblies of Lego bricks, it becomes easier to adjust their configuration by removing, adding, modifying individual bricks. This means that *bindings* between application and FTMs, but also inside FTMs can be managed dynamically. At runtime, any FTM implemented as a graph of Lego bricks is modified on-line when the FTM configuration is updated. Some Lego bricks can be uploaded when they are missing, after an off-line development when the FTM solution does not exist yet. The adaptation is carried out on-line by a specific service, called *Adaptation Engine* (see Figure 1).

A second important feature of a resilient system, also shown on Figure 1, is the runtime monitoring of the state

of the system according to several view points. This core service of the architecture is called *Monitoring Engine*. The first conventional role of the monitoring is to measure resource usage R. The second important role of the monitoring is to analyze the non-functional behavior of the system. This task is more complex and requires specific observers to capture rare error events at the hardware level, the rate of exceptions in the operating system and the applications, errors raised by operating system calls, and other incorrect behaviors reported in logs. From the collection of such inputs, triggers for FTM adaptation are computed. Monitoring issues are not addressed in this paper; we consider that triggers to signal a change in resource usage R or fault model FT are already available.

A third service, called *Resilience Management*, corresponds to the link with the off-line *System Manager* responsible for the updates of the application (i.e. versioning), the identification of their characteristics A, the development of a new FTM as a graph of software components, the update of components when needed, and the definition of transition scripts to modify the on-line configuration of an FTM.

The big picture of the resilient system architecture we consider is the following:

- a *Middleware* responsible for the execution of applications and FTMs as assemblies of software components;
- an *Adaptation Engine* responsible for the management of FTMs as graphs of software components;
- a *Monitoring Engine* responsible for the capture of the system state and the computation of triggers;
- a *Resilience Management Service* responsible for the interaction with the system manager.

This framework is consistent with the *Open Systems Dependability* concept and the principles of *Dependability Engineering for Ever-Changing Systems* [13].

3.2 Adaptation of FTMs

In this section, we describe a set of well-established FTMs in terms of their underlying characteristics and we present a transition scenario encompassing these FTMs. Our objective in this work is not to invent more sophisticated FTMs, nor to discuss complex variants of the proposed mechanisms, but to illustrate our approach for resilient computing with quite simple implementations of conventional techniques.

3.2.1 Illustrative set of FTMs

To illustrate our approach, we use two variants of a duplex protocol tolerating crash faults and two mechanisms for tolerating hardware value faults (e.g., bit flips).

- **Tolerance to crash faults:** Simple replication of the server on two distinct hosts (master and slave(s)) is a solution: a crash of the master is detected by a dedicated entity (e.g., heartbeat, watchdog) and triggers a recovery action by which the slave becomes master. There are two main types of duplex protocols: passive and active, and many variants. *Primary-Backup Replication (PBR)* is a passive strategy: only the master processes client requests and sends checkpoints containing its state to the slave(s). *Leader-Follower Replication (LFR)* is an active strategy: all replicas process input requests but only the master replies to the client.
- **Tolerance to value faults:** A solution can rely on repetition of request processing or duplex processing with acceptance tests/assertions. For instance, *Time Redundancy (TR)* tolerates transient value faults by processing repetition on a single host. A request is processed twice and results are compared. If results differ, due to a transient fault, the request is processed again and if two results out of three are identical, the reply is sent.

Another technique can be used to tolerate transient faults using assertions derived from safety properties of the application: the *Assertion&Duplex (A&Duplex)* strategy. When the first execution fails (the assertion is false), then the re-execution is done on a different node. The synchronization between replicas can be based on any duplex protocol variant mentioned above, tolerating thus crash faults as well. The assertion can be determined by a safety analysis of the system, e.g. a *Failure Mode, Effects, and Critical Analysis (FMECA)*. This technique is similar to distributed recovery blocks [14] that can tolerate both hardware and software faults when replicas are diversified.

- **Assumptions and performance analysis:** The underlying characteristics of the considered FTMs, in terms of (FT,A,R), are shown in Table 1. For instance, PBR and LFR tolerate the same fault model, but have different A and R. PBR allows non-determinism of applications because only the Primary computes client requests while LFR only works for deterministic applications as both replicas compute all requests. PBR requires state access for checkpoints and higher network bandwidth (in general), while LFR does not require state access but generally incurs higher CPU costs (and, consequently, higher energy consumption) as both replicas perform all computations. TR requires state access because application state must be captured before the first request processing and restored between two consecutive executions. As it runs on only one host, it cannot tolerate crash and it has no bandwidth requirements. A&Duplex can tolerate both crash faults and value faults as two CPUs are used to run the replicas. Both TR and A&Duplex require more computation power (i.e., more energy) than PBR because of multiple request processing.

Characteristics \ FTM		PBR	LFR	TR	A&Duplex
Fault Model (FT)	Crash	✓	✓		✓
	Transient value			✓	✓
	Permanent value				✓
Application Characteristics (A)	Deterministic	✓	✓	✓	✓
	Non-deterministic	✓			✓
	Requires state access	✓		✓	
Resources (R)	Bandwidth	high	low	n/a	low
	CPU	low	low	high	high

Table 1: (FT, A, R) parameters of considered FTMs

- **Dealing with more complex fault tolerance strategies:** We voluntarily use simple definitions of the mechanisms here for the sake of clarity. However, many variants of passive and active strategy exist. A more detailed discussion about replication techniques in distributed systems can be found in [15]. Variants of the LFR strategy where non-deterministic decisions are captured and forwarded from the *Leader* to the *Follower* do exist. We could also consider multiple *Backups* or *Followers* making thus the use of Atomic Broadcast protocols highly useful in the implementation. With the *Assertion&Duplex (A&Duplex)* strategy, one can decide to use a diversified implementation of the function to handle software faults. Clearly, *Fault Tolerance Design Patterns* can be implemented in many ways.

Considering several software fault tolerance techniques like

Recovery Blocks (RB) and Triple Modular Redundancy (TMR), the proposed approach, promoting dynamic updates using Lego bricks, is of interest without changing the execution logic of the mechanism — for RB, an update consists of changing the *acceptance test*; for TMR, an update consists of replacing the *decision algorithm*. Both updates help improving the fault tolerance coverage of such techniques.

The resource parameters, monitored on-line, obviously depend on the application, the physical architecture of the system (CPU, networks, HW configuration, etc.).

3.2.2 Possible transitions

During the service life of the system, the values of the parameters enumerated in Table 1 can change. An application may become non-deterministic because a new version is installed. The fault model may change from crash fault only to crash and value fault, due to hardware aging or physical perturbations. Available resources may also vary, e.g., bandwidth drop or constraints in energy consumption. Figure 2 shows a graph of possible transitions between the previously defined FTMs variants. The vertices represent the FTMs and the edges are labeled with the parameter (*FT*, *A*, *R*) whose variation triggers the transition. For instance, the PBR→LFR transition is triggered by a change in application characteristics or in resources, while the PBR→A&Duplex transition is triggered by a change in the considered fault model, i.e. the need to check a user-defined safety property with an assertion. Transitions can occur in both directions, according to parameter variation.

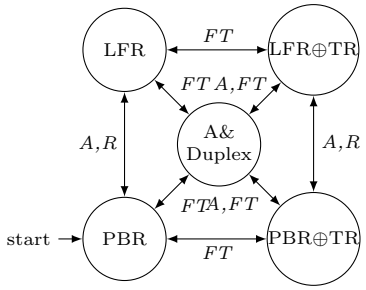


Figure 2: Transitions between FTMs

The variation of the fault model often requires a *composition of FTMs*, e.g., if we start by tolerating only crash faults with PBR and we want to add tolerance to transient value faults, we will compose PBR with TR and obtain a composed FTM combining the features specific to PBR with those specific to TR. In this paper, the ⊕ operator denotes composition (e.g., PBR⊕TR in Figure 2).

4. DESIGN FOR ADAPTATION OF FTMs

The second step of our development process is an analysis of the previously described FTMs in order to extract common parts and variable features between them. This leads to a generic execution scheme of fault tolerance strategies and a framework of *Fault Tolerance Design Patterns* (FTDPs). We follow a "design for adaptation" approach consisting of several design loops. Each loop represents a refinement step towards the optimal representation of the various concerns and protocols. The result is a pattern system [16] for fault tolerance because the considered FTMs are reusable and customizable solutions to specific problems and we highlight the structural and conceptual dependencies between them.

In its first stage, the framework contained only the design (and corresponding implementation) of a PBR strategy whose core was concentrated in a class encompassing general fault tolerance concerns, duplex concerns and specific PBR concerns. Our aim was to reach a clean separation between all these concerns in order to maximize reuse when developing new duplex variants and other FTMs.

4.1 First design loop

By analyzing the above described FTMs, we identified a generic execution scheme which captures their common parts and their variable features. Upon reception of a request from the client, a fault-tolerant server executes some actions *before* processing. Then it *proceeds* with request processing. *After* processing, it executes some actions and finally it sends the reply to the client. We call this the *Before-Proceed-After* generic execution scheme, inspired from aspect-oriented programming [12]. The *Before-Proceed-After* software bricks comply with the *Server coordination* phase (*synchronisation before*), the *Execution* phase (*proceed*) and the *Agreement coordination* phase (*synchronization after*) described in [15]. Table 2 describes the content of each *before-proceed-after* execution step for the FTMs we consider.

FTM	Before	Proceed	After
PBR (Primary)	Nothing	Compute	Checkpoint to Backup
PBR (Backup)	Nothing	Nothing	Process checkpoint
LFR (Leader)	Forward request	Compute	Notify Follower
LFR (Follower)	Receive request	Compute	Process notification
TR	Capture state	Compute	Restore state
A&Duplex	Nothing	Compute	Assert output

Table 2: Generic execution scheme of considered FTMs

When designing a duplex mechanism, this scheme can be translated in *Before-Proceed-After*, because an inter-replica synchronization takes place before request processing and another one after. This generic execution scheme enabled us to factorize in a class what is common to all duplex protocols, `DuplexProtocol` and then specialize, through inheritance, the concrete FTMs, `PBR` and `LFR` (see Figure 3). Other duplex variants can be added to the framework, either by inheriting from the abstract base class `DuplexProtocol` or by overloading concrete classes.

4.2 Second design loop

Another separation can be done between what is common to all FTMs and what is specific to duplex ones. Communication with the client, preservation of "at-most-once" semantics and request forwarding to the concrete functional service in the *processing* step are now encapsulated in a class, `FaultToleranceProtocol` in Figure 3. This second factorization enabled us to introduce in our framework non-duplex protocols targeting other fault models than crash, more precisely value faults (transient and permanent): `TimeRedundancy` and `Assertion`, which follow the same generic execution scheme. New protocols can be easily added to the framework either by extending the abstract base class `FaultToleranceProtocol` or one of the concrete classes.

Composing FTMs — As a direct consequence of the two design loops, the composition of FTMs is intuitive and almost immediate. By inheriting from a duplex protocol (tolerating crash faults) and from a value fault tolerance mechanism, we obtain four composed FTMs (Figure 3): `PBR_TR`

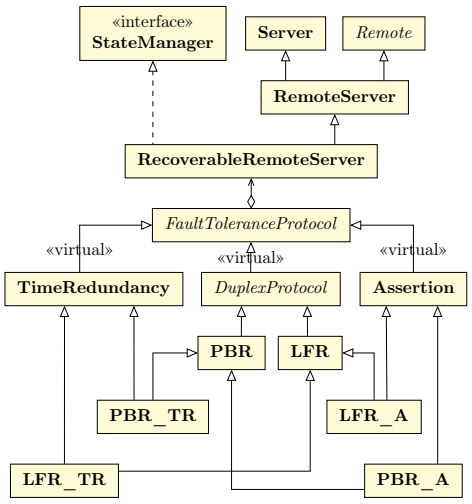


Figure 3: Excerpt of Fault Tolerance Design Patterns

and `LFR_TR`, corresponding to PBR⊕TR and LFR⊕TR respectively in Figure 2, and `PBR_A` and `LFR_A` which are two variants of A&Duplex. Figure 3 shows an excerpt of the final framework resulting from the two design loops.

Variable features — The elements of our generic execution scheme represent the variable features between FTMs. Comparing, for instance, the execution scheme of PBR with the one of LFR (see Table 2) gives us the intuition that by dividing the inter-replica protocol in isolated bricks/components which can be identified and manipulated on-line, we could execute a *differential* transition between PBR and LFR. This means replacing only the components containing the variable features between the two FTMs, without modifying the rest of the system (e.g., communication with the client, the actual processing to which *proceed* only forwards the requests). By identifying the variable features, we can easily develop FTM variants from existing ones (off-line) and, using a component-based middleware support, execute transitions between FTMs with few modifications (on-line).

4.3 Validation of the design

All the above design versions developed in UML with *IBM Rational Software Architect* have been implemented and validated in C++. As further explained, the benefits of the "design for adaptation" approach lie in reducing the development time and effort necessary for implementing new FTMs and in increasing readability and separation of concerns/-functionalities. The implementation of composed mechanisms is almost immediate thanks to the two design loops.

Figure 4 shows that the effort spent on factorization and design refinement during the two design loops is 4 to 5 times more significant than the effort to develop one FTM. However, once the right design is achieved, the development of both individual and composed FTMs is extremely easy. For instance, while the second design loop took 5 days, the development of Assertion and Time Redundancy each took half a day. The composition of FTMs, which is probably the most interesting result of this endeavor, only took half a day thanks to the two design loops.

During the development of this system of patterns, we observed that our design approach was very efficient, in terms

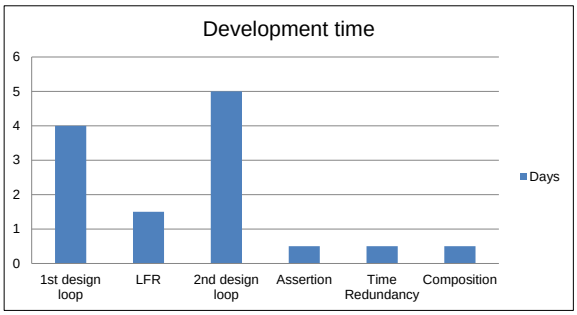


Figure 4: FT design patterns: development time

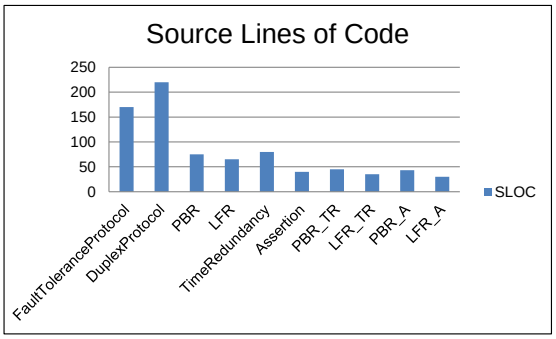


Figure 5: FT design patterns: source lines of code SLOC

of development time of new FTMs (see Figure 4), in terms of code reuse and lines of code to be produced (see Figure 5).

The development of any FTM is carried out off-line. Any update impacts the FTM that must be validated off-line before it can be used. The analysis of possible inconsistencies in FTMs composition is part of the off-line validation process and is thus performed before any on-line adaptation. Inconsistency analysis of FTMs composition has been addressed in our previous works. The interested reader is referred to [17, 18] for details.

4.4 Component-based FTMs

In this step of the development process, the fine-grained design of FTMs based on the generic *Before-Proceed-After* execution scheme was mapped on FraSCAti [19], an open-source reflective component-based middleware. FraSCAti provides runtime support for applications designed according to the Service Component Architecture (SCA) specifications [11]. FraSCAti enriches the basic SCA specification with support for on-line exploration and reconfiguration of component-based assemblies. To this aim, it integrates FScript [20], a script language for writing reconfiguration procedures. FraSCAti and FScript provide what we have identified as the *minimal API* for performing the fine-grained adaptation of FTMs, namely:

- control over component lifecycle at runtime (add, remove, start, stop);
- control over interactions between components for creating and removing reference-service connections;
- consistency of reconfigurations performed by scripts written in FScript.

It is worth noting that our approach is reproducible on any other platform that provides these features.

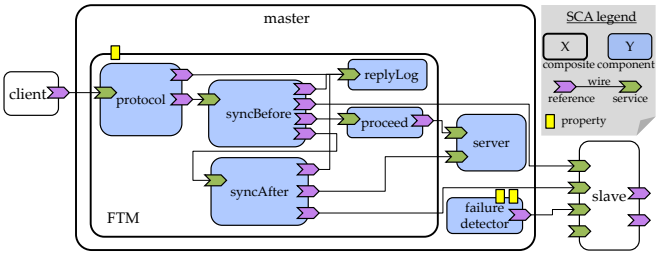


Figure 6: Component-based architecture of PBR

Figure 6 shows the resulting component-based architecture of PBR. The steps of the generic execution scheme are isolated in the components syncBefore, proceed, syncAfter. Thanks to the "design for adaptation" process, these variable features, that are subject to change during transitions, are mapped on small stateless components. The actual state of the FTMs (e.g., request id, computation result) and the common parts of FTMs that were captured in the two base classes in the object-oriented design (i.e., `FaultTolerance-Protocol` and `DuplexProtocol` in Figure 3) are now mapped on components that are not modified during transitions between FTMs, i.e., protocol, replyLog and server.

5. ADAPTATION OF FTMs AT RUNTIME

5.1 Adaptation process implementation

Figure 7 outlines the adaptation process implementation for executing fine-grained transitions between FTMs, under the supervision of the *System Manager*. The target *system* runs an application to which are attached appropriate FTM(s) (i.e., consistent with the current values of (FT,A,R)).

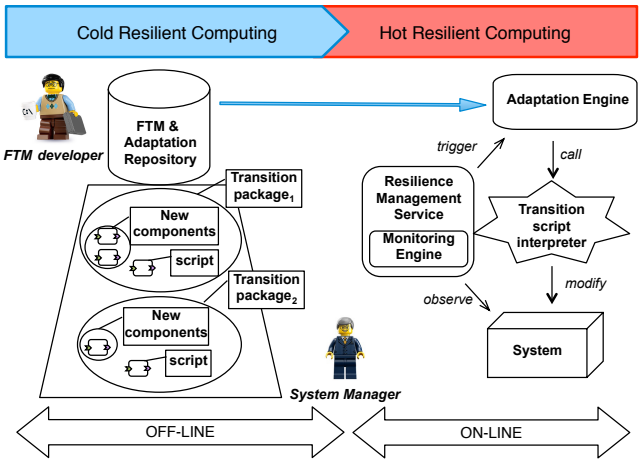


Figure 7: Adaptation process implementation

When there is an inconsistency between the values of (FT,A,R) and the current FTM, the *Resilience Management Service* triggers the *Adaptation Engine* giving it as input the new FTM towards which a transition must be executed (if such an FTM exists). The *Adaptation Engine* gets the required transition package from the *FTM&Adaptation Repository*. This package contains the new bricks that must be integrated into the existing software architecture in order to execute a differential transition from the current FTM to the new one and a script that operates the transition. Next, the transition package feeds the *Script interpreter* that modifies the current FTM.

In our view, the process encompasses two aspects: *cold resilient computing*, covering off-line activities (the development of the transition packages from the repository), and *hot resilient computing*, consisting of all the other entities and their interactions that occur on-line. Together, these two aspects form a loop: the cold one feeds transition packages to the hot one, and the hot one provides feedback for improving and enriching the cold one.

5.2 On-line fine-grained transitions

To illustrate the feasibility of the approach, we performed several fine-grained transitions between selected FTMs, corresponding to the transition scenario from Figure 2. The PBR→LFR transition is triggered by variations in A or R (based on Table 1) and requires changing the syncBefore and syncAfter components (based on the generic scheme in Table 2). All the other components corresponding to the massive common parts are left untouched.

Each transition implies the deployment of a transition package containing the new components that must be introduced in the existing architecture (in this example, syncBefore and syncAfter of LFR) and a script written in FScript that removes the components that are no longer necessary (here, syncBefore and syncAfter of PBR) and replaces them with the new ones. In short, the script written in FScript that performs the PBR→LFR transition does the following:

- disconnect the old *syncBefore* and *syncAfter* from all their services and references;
- delete old components and add the new ones;
- connect the new *syncBefore* and *syncAfter* to all the necessary services and references.

The LFR→LFR⊕TR transition (i.e., the composition of LFR with TR) is triggered by the evolution of the fault model, from crash fault to crash fault and transient value faults. Such an evolution of the fault model may occur due to hardware aging or physical perturbations. Table 2 outlines the specificities of TR, namely the state capture and restoration *before* and *after* processing, respectively. To simplify the mapping on the component-based architecture, the behavior of TR is implemented by a proceed component that repeats processing and compares results. The LFR→LFR⊕TR transition thus consists in replacing the elementary proceed component of LFR.

5.3 Consistency of distributed adaptation

As evolvability must not affect the reliability of fault-tolerant application, the consistency of transitions between FTMs must be ensured.

Local consistency— FraSCAti and its integrated FScript engine guarantee the consistency of local reconfigurations performed using scripts. FScript enforces an *all-or-nothing* semantics [20]. The reliability of the reconfiguration is achieved

using a model of component-based configurations and reconfigurations (i.e., transitions between two consistent configurations), verifying integrity constraints and performing reconfigurations in a transactional manner [20]. In case of constraint violation during the reconfiguration process, a `ScriptException` is thrown, the transaction is rolled back and the FTM remains in its initial configuration.

Consistency of request processing— Stopping a component that must be replaced implies waiting for it to reach a quiescent state where all its internal processing is finished, blocking its inputs and buffering them. This means that a client request received before adaptation is processed and the client receives the reply before the subsequent requests are blocked and buffered and the interaction between replicas is locked. When the lock is released, the buffered client requests are processed in the new configuration of the FTM.

Distributed consistency — The FTMs consist of two replicas and a specific inter-replica protocol, therefore transitions between FTMs is performed on two hosts. On each one, a script component performs the required reconfiguration, which can either terminate successfully or with a `ScriptException`, if integrity constraints are violated. The script component on each host is wrapped in a Java class which kills the local replica if an exception is thrown, to enforce the fail-silent assumption and to prevent the overall FTM from reaching an inconsistent configuration. Therefore, a local reconfiguration can either terminate successfully or crash. The failure detection mechanism incorporated in all duplex strategies detects the crash and informs the remaining replica. If this replica has successfully reconfigured, it becomes master-alone.

Recovery of adaptation— Replicated computing units may crash during transitions. Upon successful completion of the reconfiguration of one replica (either master or slave), the current configuration (i.e., the target FTM) is logged on a stable storage. Should the other replica crash (either because of a `ScriptException` or because of a physical fault) before completing its own reconfiguration, a new replica is restarted in the same configuration as its counterpart, which has successfully completed the transition. This information is recovered from the stable storage which keeps track of the currently active configuration.

5.4 Detailed analysis of transition scenarios

Figure 8 shows an excerpt of the graph of possible transitions (on the left) and an extended graph of transition scenarios between the FTMs in our subset resulting from it (on the right). We call it a graph of scenarios because there are several events which lead to a transition between FTMs.

For the sake of clarity, we present a simplified view of adaptation triggers. In general, they consist in more complex logical expressions, which are out of the scope of this paper. As previously explained, PBR requires state access and can be used both for deterministic and non-deterministic applications. LFR requires application determinism and can be used both for applications that provide state access and those that do not (see Table 1). This explains the two PBR states and the two LFR states in Figure 8. The "No generic solution" state corresponds to non-deterministic applications that do not provide state access.

Mandatory vs. possible transitions — Figure 8 shows three types of transitions: mandatory transitions (continuous red lines), possible transitions (dashed green lines) and intra-FTM transitions (dotted black lines).

- *Mandatory transitions*: Parameters whose variation invalidates the initial FTM or affects its performance and therefore requires a transition towards an appropriate one. These are mandatory transitions. When starting from PBR, there are two such cases: bandwidth drop (which introduces undesired overheads) and state inaccessibility (which makes checkpointing impossible).

- *Possible transitions*: Parameters whose variation only makes optional the use of another FTM, without invalidating the initial one, lead to possible transitions. When starting from PBR, there are two such cases: increase in available CPU (because LFR demands more processing than PBR) and application determinism (because both PBR and LFR work in this case).

While mandatory transitions can be executed automatically, possible transitions are executed only if the system manager decides to. The intra-FTM transitions are only represented in Figure 8 for the sake of completeness. If a possible transition towards a new FTM is not executed (e.g., to go from "PBR with non-determinism" to "LFR with state access" when the application becomes deterministic), the current FTM will still change its configuration, i.e., execute an intra-FTM transition (e.g., "PBR with non-determinism" goes to "PBR with determinism").

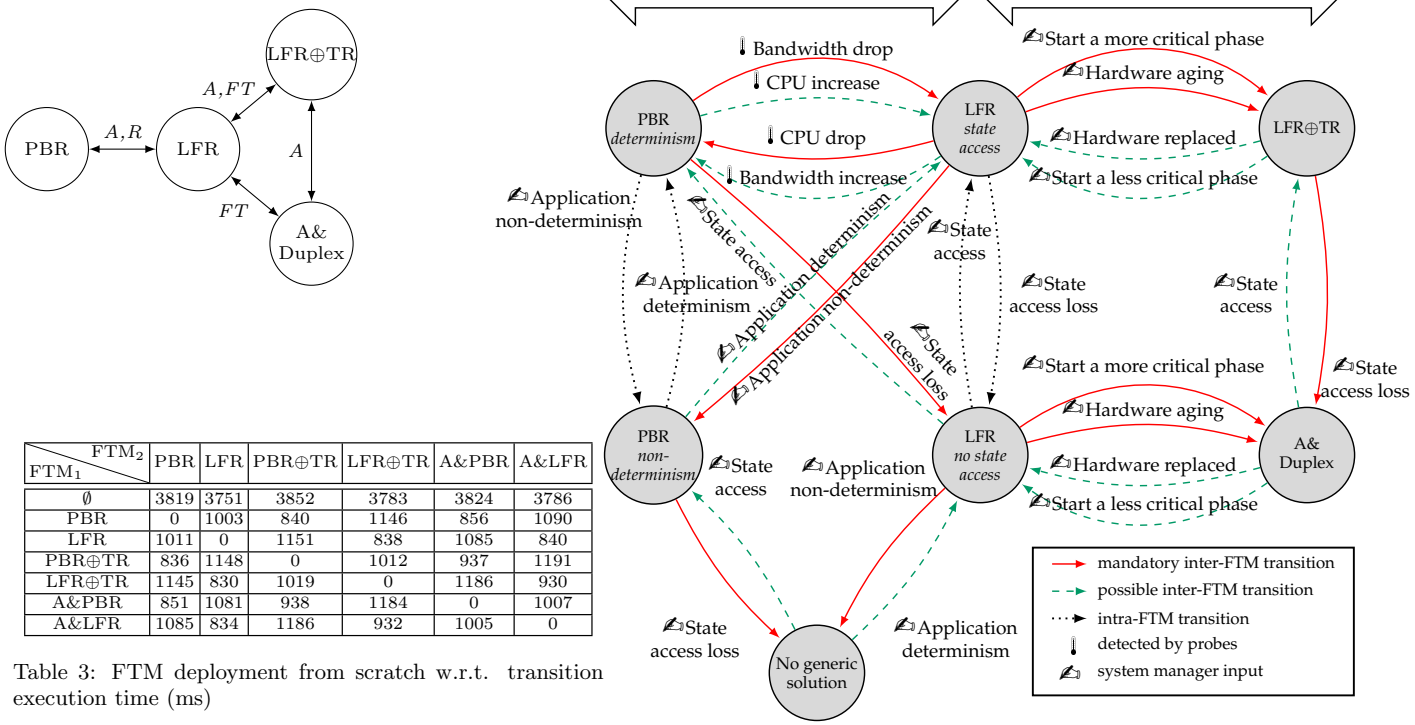
Stability of transitions and triggers— A problem which can be encountered in adaptive fault tolerance, perceived as a closed loop system, is the oscillation between FTMs: if a transition is triggered by the variation of a parameter that oscillates near the reconfiguration threshold, the system can become unstable and reconfigure itself too often, thus reducing its availability. By distinguishing between mandatory and possible transitions, this issue is solved for our FTM examples: as Figure 8 shows, the reverse of a mandatory transition is always a possible one. The risk of oscillation is eliminated because once the system executes a mandatory transition due to the variation of a parameter, it will not be able to revert to the previous FTM, unless the system manager decides to.

Figure 8 also shows that changes of some parameters can be detected automatically by using probes 🌡, while others likely require input/observations from the application developer or from the system manager ✍. The former encompasses R variations, the latter concerns A and FT variations.

Reactive vs. proactive — A fundamental difference in the nature of transitions concerns *when* they must be triggered, either as a *reaction* to an event which has occurred or *in advance*, before the foreseen occurrence of an event. Changes in available resources R usually impact the performance overhead entailed by a given FTM. Therefore, the system manager can search for a more appropriate FTM as a reaction to fluctuations in resources. In the case of application changes A due to versioning, the system manager can select/define a suitable FTM for the new version, taking input from application designers wrt the new characteristics. The transition encompasses changing the application version together with the FTM (if the new characteristics invalidate the previous FTM). As such, the FTM is changed as a reaction to application changes.

The case of fault model FT changes is the most complex. In the context of operational phases, one can understand

Figure 8: Excerpt of Figure 2 (left) and extended graph of transition scenarios between FTMs (right)



FTM$_1$ \ FTM$_2$	PBR	LFR	PBR⊕TR	LFR⊕TR	A&PBR	A&LFR
∅	3819	3751	3852	3783	3824	3786
PBR	0	1003	840	1146	856	1090
LFR	1011	0	1151	838	1085	840
PBR⊕TR	836	1148	0	1012	937	1191
LFR⊕TR	1145	830	1019	0	1186	930
A&PBR	851	1081	938	1184	0	1007
A&LFR	1085	834	1186	932	1005	0

Table 3: FTM deployment from scratch w.r.t. transition execution time (ms)

that the fault model for a given phase has been anticipated and, for critical phases, it is stronger than for non-critical ones. For unanticipated changes, the situation is more subtle because unpredicted faults may be out of the scope of the current FTM, which is thus unable to tolerate them. In this context, the evolution of the fault model in operation must be addressed in a proactive way that performs FTM updates in advance, either because the system is getting to a new operational phase or because of an early detection of fault model changes.

Figure 8 highlights the reactive/proactive nature of transitions: the transition between PBR and LFR, caused by variations in A or R, is reactive, while the transitions between LFR and LFR⊕TR and between LFR and A&Duplex, caused by variations in FT, have a proactive nature.

Again, any transition not present in the graph corresponds to an unanticipated evolution trajectory. As a matter of fact, the system cannot perform the corresponding FTM update. However, our approach enables this update to be developed and installed faster than with monolithic approaches.

6. EVALUATION

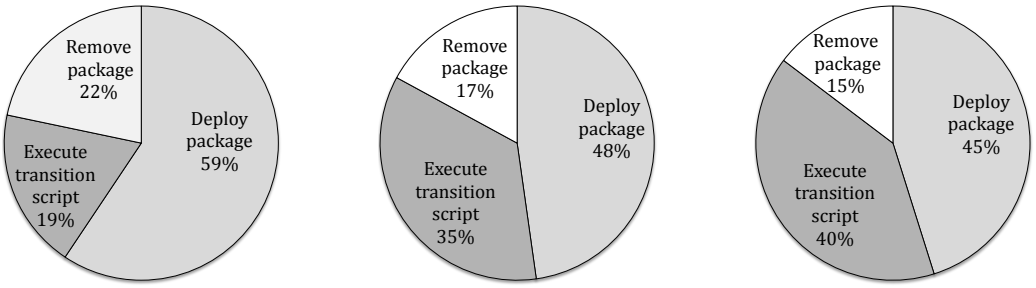
In this section, we analyze the performance and agility of our differential approach and associated on-line transitions between FTMs. Firstly, we assess the advantages of performing differential modifications through some measurements. Next, the agility of transitions is discussed.

6.1 Performance

In order to assess the performance of agile fine-grained transitions between FTMs, we mapped all the FTMs in the illustrative set on a component-based architecture. More precisely, we developed PBR, LFR, PBR⊕TR, LFR⊕TR, A&PBR and A&LFR as stand-alone FTMs that can be directly deployed and all the differential transitions between them, both direct and inverse.

In Table 3, we compare the time necessary for deploying full FTMs (first line) and the time necessary for executing differential transitions between them. These results represent averages over 100 test runs on a PC for each cell of the table. As deployment of FTMs and transitions are performed in parallel on two replicas, in this table we show the time corresponding to one replica. We can easily notice that our differential approach not only eliminates state transfer issues inherent to monolithic replacements of FTMs but also is more efficient in terms of execution time, making the system more reactive to changes. Clearly, the ratio between the transition time and the deployment time is more relevant than absolute values.

Transitions consist in three main steps: deployment of transition packages, execution of reconfiguration scripts and removal of residual components. This process is orchestrated by the *Adaptation Engine* (see Figure 7). Figure 9 shows the contribution of each step to the total duration for three transitions affecting different number of components (i.e., from one variable features to all three). The actual execution of the reconfiguration script, which can be considered the most complex step, takes 19% of the total time for the replacement of 1 component, 35% for 2 components and 40% for 3 components. This means that even in the most complex transitions, affecting all three variable features, the execution of the transition script (i.e., of the reconfiguration mechanics) takes less then half of the total transition time. These quantitative measurements also give us indications as to what could speed up the transition process, namely the optimization of the deployment step, which currently takes



(a) LFR→LFR⊕TR (1) (b) PBR→LFR (2) (c) PBR→LFR⊕TR (3)

Figure 9: Transition time distribution w.r.t. number of components replaced

approximately half of the total transition time.

6.2 Agility

The agile fine-grained approach aims to equip an application with the FTMs which are necessary only, and provide system developers and managers with means to adapt these FTMs when needed to accommodate change.

The current implementation of our framework proves the feasibility of agile adaptation of FTMs. As illustrated in Figure 7, transitions which were either unknown or considered unnecessary at design time can be performed during the lifetime of the system by developing off-line the necessary transition package and integrating it on-line. All the transitions between FTMs are performed by minimizing the impact on the overall component-based architecture, i.e., by replacing only the variable features. Furthermore, at no point in time is the system loaded with unnecessary FTMs or parts of FTMs (resulting in "dead code") as in the case of preprogrammed adaptations.

In our investigation of related work on AFT, we found examples of quantitative evaluations of transitions between FTMs. In [10], the switch from an active to a passive strategy takes 4.5ms. The stabilization between passive and active replication takes 360ms and the reverse takes 390ms in [9]. In both cases adaptation is preprogrammed, i.e., the supported FTMs are known at initial design time and hard-coded at system deployment. In [8], it takes 260ms to alternate between passive and active strategy. Although the authors leverage a component model, reconfiguration does not appear to be performed agilely at runtime. While in our case the transition from passive to active replication takes 1003ms in total , the substantial difference lies in the fact that this is an agile adaptation, not a preprogrammed one. As expected, agility comes with an additional cost in terms of deployment time. However, compared to [9, 8], this cost does not appear to be excessive, given that our approach brings flexibility and the ability to accommodate changes unforeseen at design time.

7. RELATED WORK

There is a substantial body of work on adaptive software [2]. For example, RAINBOW [21] builds on the use of architectural models for problem diagnosis and repair. An architecture manager is in charge of maintaining the architectural model at runtime and of detecting the violation of constraints on system elements. The project includes an ADL called ACME [22], a system in charge of verifying constraints, called ARMANI, a library of gauges, etc. The Plastik framework [23] results from the mapping of an enriched version of ACME to the OpenCOM component-based middleware and enables programmed and ad-hoc changes at runtime while maintaining certain constraints. Although interesting from a methodological point of view, these projects do not tackle the particularities of fine-grained adaptation of Fault Tolerance Mechanisms (FTMs).

The need for Adaptive Fault Tolerance (AFT) rising from the dynamically changing fault tolerance requirements and from the inefficiency of allocating a fixed amount of resources to FTMs throughout the service life of a system was stated in [24]. AFT is gaining more importance with the increasing concern for lowering the amount of energy consumed by cyber-physical systems and the amount of heat they generate [25]. Conceptual frameworks for adaptive fault tolerance (AFT) describing algorithms and target systems are presented in [26, 27]. Several CORBA-based middleware exist [28, 9, 8] but evolution is tackled differently: adaptation has a parametric form (e.g., number of replicas) or it is performed off-line or, if done on-line, has a coarse-grained nature. Zheng and Lyu present in [29] an interesting approach for adapting FTMs for Web services (i.e., mechanisms tolerating software faults) based on a user-collaborated QoS-aware middleware. In [30], the authors also leverage concepts from software engineering such as software product lines (SPL) to provide a representation of FTMs targeting software faults that captures their common parts and their variable features. However, they do not go as far as implementing agile transitions between them.

8. LESSONS LEARNED & CONCLUSION

Compared to existing solutions in which adaptation consists in switching among a set of statically predefined mechanisms, we propose a development process that enables systematic *agile* fine-grained adaptation of FTMs without burdening applications with inactive code. Our approach relies on several key factors.

First of all, the "design for adaptation" of FTMs is a process during which common parts and variable features between them are identified. By analyzing an illustrative set of FTMs, a generic execution scheme called *Before-Proceed-After* was identified. The steps of this scheme, which can be directly reused on other FTMs (e.g., N-Version Programming [7]) and non-functional mechanisms (e.g., encryption), capture the variable features between mechanisms that are subject to change during transitions.

Secondly, the features provided by the component-based middleware serving as a runtime support are essential. The runtime support must implement the minimal API for fine-grained adaptation we have identified (componentization and

dynamic binding at runtime) and it must guarantee the consistency of runtime modifications of the component-based architecture. Our approach can be implemented on any runtime support providing such capabilities.

An important element of our approach is the presence of a system manager in the adaptation loop. Although monitoring and adaptation triggers are out of the scope of this work, we discussed several important aspects regarding transitions and their nature. The most interesting outcomes of this analysis are that transitions caused by fault model changes must be triggered in a proactive manner (i.e., before the actual change) and that a man-in-the-loop can prevent oscillations due to variations of a single parameter.

The main benefit of our approach lies in the agility provided to fault-tolerant systems. In this context, agility — the capacity to add new FTMs on-the-fly during operational life and to tune existing ones — is more important than simple quantitative measurements of the time it takes to perform a transition. Clearly, the transition duration must be as short as possible in order to reduce service disruption.

This work demonstrates that component-based software engineering techniques enable adaptive fault tolerance mechanisms to be developed. Our future work is carried out with partners in the European automotive industry.

9. REFERENCES

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable Secur. Comput.*, vol. 1, pp. 11–33, January 2004.

[2] P. McKinley, S. Sadjadi, E. Kasten, and B. H. C. Cheng, "Composing Adaptive Software," *Computer*, vol. 37, no. 7, pp. 56–64, 2004.

[3] J. Highsmith and A. Cockburn, "Agile Software Development: The Business of Innovation," *Computer*, vol. 34, no. 9, pp. 120–127, 2001.

[4] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Boston, MA, USA: Addison-Wesley, 2002.

[5] J.-C. Laprie, "From dependability to resilience," in *DSN, Anchorage, AK, USA, pp. G8-G9*, vol. 8, 2008.

[6] P. Barret, A. Hilborne, P. Bond, D. Seaton, P. Verissimo, L. Rodrigues, and N. Speirs, "The Delta-4 extra performance architecture (XPA)," in *Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium*. IEEE, 1990, pp. 481–488.

[7] A. Avižienis, "The *N*-Version Approach to Fault-Tolerant Software," *IEEE Transactions on Software Engineering*, no. 12, pp. 1491–1501, 1985.

[8] J. Fraga, F. Siqueira, and F. Favarim, "An Adaptive Fault-Tolerant Component Model," in *9th Workshop on Object-Oriented Real-Time Dependable Systems*. IEEE, 2003, pp. 179–186.

[9] L. C. Lung, F. Favarim, G. T. Santos, and M. Correia, "An Infrastructure for Adaptive Fault Tolerance on FT-CORBA," in *9th International Symposium on Object and Component-Oriented Real-Time Distributed Computing*. IEEE, 2006.

[10] O. Marin, P. Sens, J.-P. Briot, and Z. Guessoum, "Towards Adaptive Fault-Tolerance for Distributed Multi-Agent Systems," in *4th European Research Seminar on Advances in Distributed Systems*, 2001, pp. 195–201.

[11] J. Marino and M. Rowley, *Understanding SCA (Service Component Architecture)*. Addison-Wesley, 2009.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," *ECOOP'97 Object-Oriented Programming*, pp. 220–242, 1997.

[13] Mario Tokoro (Eds), in *Open Systems Dependability: Dependability Engineering for Ever-Changing Systems*. CRC Press,USA, 2015, p. 288.

[14] K. Kim and H. O. Welch, "Distributed execution of recovery blocks: An approach for uniform treatment of hardware and software faults in real-time applications," *Computers, IEEE Transactions on*, vol. 38, no. 5, pp. 626–636, 1989.

[15] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso, "Understanding replication in databases and distributed systems," in *Distributed Computing Systems, 2000. Proceedings. 20th International Conference on*. IEEE, 2000, pp. 464–474.

[16] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-oriented Software Architecture: A System of Patterns, Volume 1*. Wiley Chichester, United Kingdom, 1996.

[17] J. Lauret, J.-C. Fabre, and H. Waeselynck, "Detection of interferences in aspect oriented programs using executable assertions," in *IWPD 2012*.

[18] ——, "Fine-grained implementation of fault tolerance mechanisms with AOP: to what extent?" in *SAFECOMP*, 2013.

[19] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani, "A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures," *Software: Practice and Experience*, 2011.

[20] M. Léger, T. Ledoux, and T. Coupaye, "Reliable Dynamic Reconfigurations in a Reflective Component Model," *13th International Conference on Component-Based Software Engineering*, 2010.

[21] D. Garlan and B. Schmerl, "Model-based adaptation for self-healing systems," in *Proceedings of the first workshop on Self-healing systems*, ser. WOSS '02. New York, NY, USA: ACM, 2002, pp. 27–32.

[22] D. Garlan, R. Monroe, and D. Wile, *Acme: Architectural description of component-based systems*. Cambridge University Press, 2000.

[23] T. Batista, A. Joolia, and G. Coulson, "Managing dynamic reconfiguration in component-based systems," *Software Architecture*, pp. 1–17, 2005.

[24] K. H. K. Kim and T. F. Lawrence, "Adaptive Fault Tolerance: Issues and Approaches," in *Proceedings of the Second IEEE Workshop on Future Trends of Distributed Computing Systems*. IEEE, 1990, pp. 38–46.

[25] C. Krishna and I. Koren, "Adaptive Fault-Tolerance for Cyber-Physical Systems," in *IEEE International Conference on Computing, Networking and Communications (ICNC)*, 2013, pp. 310–314.

[26] J. Goldberg, I. Greenberg, and T. F. Lawrence, "Adaptive Fault Tolerance," in *Advances in Parallel and Distributed Systems, 1993., Proceedings of the IEEE Workshop on*. IEEE, 1993, pp. 127–132.

[27] M. A. Hiltunen and R. D. Schlichting, "A Model for Adaptive Fault-Tolerant Systems," in *EDCC, Proceedings of the 1st European Dependable Computing Conference, Lecture Notes in Computer Science*. Springer-Verlag, 1994, pp. 3–20.

[28] P. Narasimhan, T. Dumitraş, A. M. Paulos, S. M. Pertet, C. F. Reverte, J. G. Slember, and D. Srivastava, "Mead: support for real-time fault-tolerant corba," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 12, 2005.

[29] Z. Zheng and M. R. Lyu, "An adaptive qos-aware fault tolerance strategy for web services," *Empirical Software Engineering*, vol. 15, no. 4, pp. 323–345, 2010.

[30] A. S. Nascimento, C. M. F. Rubira, and J. Lee, "An SPL Approach for Adaptive Fault Tolerance in SOA," in *Proceedings of the 15th International Software Product Line Conference*, vol. 2. ACM, 2011, p. 15.