Selecting Fault Tolerant Styles for Third-Party Components with Model Checking Support

Junguo Li, Xiangping Chen, Gang Huang, Mei Hong, Franck Chauvel

Selecting Fault Tolerant Styles for Third-Party Components with Model Checking Support

Junguo Li, Xiangping Chen, Gang Huang*, Hong Mei, and Franck Chauvel

Key Laboratory of High Confidence Software Technologies, Ministry of Education,
School of Electronics Engineering and Computer Science, Peking University,
Beijing, 100871, China
{lijg05,chenxp04,huanggang,franck.chauvel}@sei.pku.edu.cn,
meih@pku.edu.cn

Abstract. To build highly available or reliable applications out of unreliable third-party components, some software-implemented fault-tolerant mechanisms are introduced to gracefully deal with failures in the components. In this paper, we address an important issue in the approach: how to select the most suitable fault-tolerant mechanisms for a given application in a specific context. To alleviate the difficulty in the selection, these mechanisms are abstracted as Fault-tolerant styles (FTSs) at first, which helps to achieve required high availability or reliability correctly because the complex interactions among functional parts of software and fault-tolerant mechanism are explicitly modeled. Then the required fault-tolerant capabilities are specified as fault-tolerant properties, and the satisfactions of the required properties for candidate FTSs are verified by model checking. Specifically, we take application-specific constraints into consideration during verification. The satisfied properties and constraints are evidences for the selection. A case study shows the effectiveness of the approach.

Keywords: Fault tolerance, model checking, fault-tolerant style, software architecture.

1 Introduction

Third-party components, such as COTS (Commercial Off-The-Shelf) components and service components, are commonly used to implement large-scale (often business related) applications, which contributes to their ability to reduce costs and time. These components share some common characters: a) they are produced and consumed by different people. Application developers work as consumers to integrate third-party components into applications, according to interfaces provided by components providers. b) They are reused as black boxes or "gray" boxes. Consumers are unaware of technical details about how such a component is implemented as well as what's the difference between its updated version and its previous version.

* Corresponding author.

These characters raise a challenge to build highly available or reliable applications out of unreliable components that lack special Fault Tolerance (FT) design[1]. Most of the existing studies on the topic try to attach fault-tolerant mechanisms (for example, reboot, retry, or replication) to the external of COTS components [11] or services [24] as "wrappers" or "proxies". They specify such applications at Software Architecture (SA) level because SA is good at modeling interactions among multiple components. The approach works well except that a question is not well answered yet: which fault-tolerant mechanism is the most suitable one for a given third-party component in a specific application? The question stems from two facts. On the one hand, the effectiveness of a fault-tolerant mechanism depends on its fitness for an application context, including fault assumption, application domain, system characteristics, etc. [23]. None of the existing mechanisms, such as reboot, recovery blocks and N-Version Programming, are capable of tolerating all faults in all contexts. On the other hand, an anticipated fault characters in a third-party component may change due to its upgrade. So we need to select a suitable mechanism for third-party components during application development or maintenance, taking into consideration the components' fault assumption, the application's specific constraints, etc.

In the paper, we present a specification and verification-based solution to the problem. The contributions of the paper are two-fold. First, we offer solid evidences to selecting the most suitable fault-tolerant mechanism for a component running in a specific application. The evidences are obtained by model checking and they are necessary to resolve conflicts between continually evolving components and the specificity required by fault-tolerant mechanisms. Once a mechanism cannot successfully recover a fault in an upgraded component, the approach applies to the component again to select another. Second, we take application-specific dependency relationship into consideration in formal specification of an FTSA. This relationship affects the selection and usage of fault-tolerant mechanisms.

In the solution, we specify fault-tolerant mechanisms at first, which improve third-party components' availability or reliability, as a special software architectural style, i.e. fault-tolerant styles (FTSs). Then we define semantics of Fault-Tolerant Software

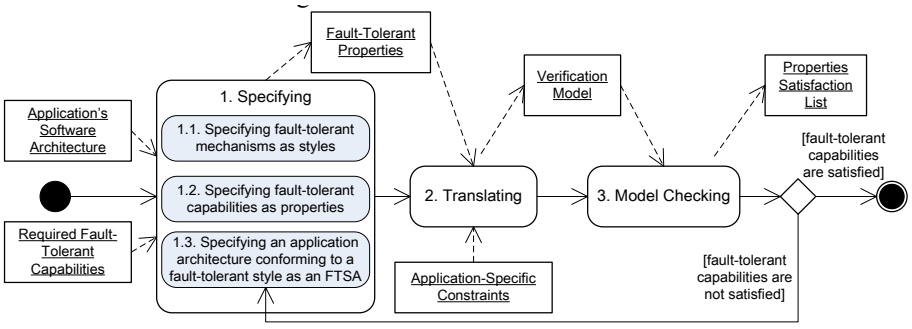


Fig. 1. A process for selecting suitable fault tolerance mechanisms for an component in an application

[1] We only discuss software-implemented fault tolerance (SIFT) design in the paper.

Architectures (FTSAs) conforming to an FTS. The definition supports an application-specific constraint (dependency relationships among components), which is seldom considered before. Based on formally specified fault-tolerant properties like fault assumptions, fault-tolerant capability, and application-specific constraints, we verify whether an FTS preserves required properties by model checking. The result determines the suitableness of a mechanism for a specific component in an application. An automatic translation from FTSAs' behavioral models into model checker's input (verification model) is also given. Fig. 1 presents an overview of the proposed approach.

The remainder of the paper is organized as follows: Section 2 gives an overview of FT and a motivating example of selecting an FTS for an EJB component. Section 3 describes the concept of FTS and FTSA, and how to model FTSs before Section 4 describes how to formalize FTSA, fault-tolerant properties and application-specific constraints. Section 5 focuses on translating the above formalized model into Spin model checker's input. Section 6 shows how to use the approach to solve the problem in the motivating example. Section 7 clarify some notable points in the approach and discuss the limits of the approach. At last, we give an insight into conclusions and future work in Section 8.

2 Background and a Motivating Example

In this section, we give an overview of software-implemented fault tolerance mechanism and a motivating example to clarify the problem solved in the paper.

2.1 Software-Implemented Fault Tolerance

Software-implemented fault tolerance is an effective way to achieve high availability and reliability. An activated fault in a component causes errors (i.e. abnormal states that may lead to failures), which is manifested as a failure to its clients. Failures prevent software from providing services or providing correct services for clients [3]. A fault-tolerant mechanism uses a set of software facilities to take two successive steps to tolerate faults: the error detection step aims to identify the presence of an error, while the recovery step aims to transit abnormal states into normal ones (some masking-based fault-tolerant mechanisms do not take the recovery step). The difference among various mechanisms is the way to detect errors and to recover states. Existing fault-tolerant mechanisms are classified as design diversity-based, data diversity-based or environment (temporal) diversity-based mechanisms. Design diversity-based mechanisms require different designs and implementations for one requirement. As a result, even if a failure happened in one version, correct results from other versions can mask it. Data diversity-based mechanisms use data re-expression algorithms to generate a set of similar input data, execute the same operation on those inputs, and use a decision algorithm to determine the resulting output. Environment (temporal) diversity-based mechanisms try to obtain correct results by re-executing failed operations, with different environment configurations. This kind of mechanism is efficient to deal with environment-dependent failure that only happened in a specific execution environment.

Fault-Tolerant Style (FTS) specifies the structural and behavioral characters of an fault-tolerant mechanism very well. Usually FTSs can be modeled in: a) Box-and-line diagrams and formal (or informal) behavioral descriptions [27]; b) Architectural Description Languages (ADLs) [10, 13, 25]; or c) UML or an UML profile [26, 14]. The latter two are widely used in current practices. We take an UML profile for both SA and FT as the modeling language in the study, but pure ADLs can also be used.

2.2 A Motivating Example

ECperf [9] is an EJB benchmark application, which simulates the process of manufacturing, supply chain management, and order/inventory in business problems. *Create-a-New-Order* is a typical scenario in ECperf, i.e. a customer lists all products, adds some to a shopping cart, and creates a new order. We use Software Architecture (SA) model to depict the interactions among EJBs in the scenario in Fig. 2. We assume these EJBs are black boxes and we have nothing known about their implementation details. The structural model of ECperf comes from runtime information analysis, with the monitoring support provided by a reflective JEE Application Server (AS) [16].

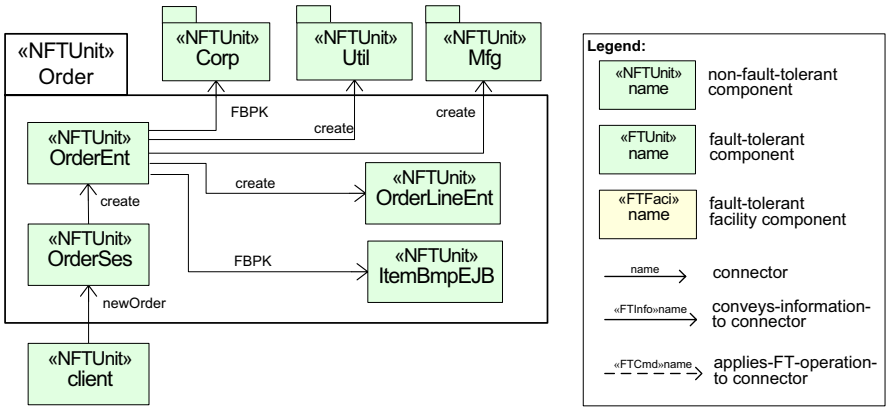


Fig. 2. The part of software architecture of ECperf in the Create-a-New-Order scenario

ECperf cannot tolerate any faults originally, but it needs to be fault tolerable, especially for *ItemBmpEJB*, which is a frequently used bean-managed persistent EJB in the Create-a-New-Order scenario. There are on average more than 400 invocations for ItemBmpEJB, compared with only one invocation for *OrderEnt* in the scenario. The availability of ItemBmpEJB may be imperiled by database faults or unreliable connections to databases. These faults are permanent – they do not disappear unless the database or the connections are recovered, unlike transient faults which may disappear in a nondeterministic manner even when no recovery operation is performed. In addition, these faults are activated only under certain circumstances like heavy-load or heavy communication traffic. So the first fault-tolerant requirement is to make ItemBmpEJB capable of tolerating environment-dependent and non-transient (EDNT) faults.

A common constraint in applications is dependency among components. A component in an application may use other components to fulfill desired functionalities. We call the former a *depending* component, and the latter a *depended-on* component and there is a dependency between them [17]. We classify dependencies as *weak* ones or *strong* ones: If every time a depending component C_1 uses a depended-on component C_2, and C_1 has to look up or acquire C_2 as the first step, there is a weak dependency. On the other hand, if C_1 can invoke C_2 directly by a reference R that is created during C_1's first initialization, it is a strong dependency. Strong dependency is common because the design improves application's performance if the invocation happens frequently. With the proliferation of Dependency Injection design in component-based development, the heavy dependency is more popular. Strong dependencies require a coordinated recovery capability. If C_2 failed at some time, the R in C_1 is invalid either. Any new invocation for C_2 by the R would definitely cause a failure. From the point of view of fault, it is a global error that needs to be recovered coordinately, which is stated in our previous work [17]. Either the R is updated or C_1 is recovered when recovering C_2. By analyzing runtime information, we find OrderEnt strongly depends on ItemBmpEJB because there is a *lookup* invocation in OrderEnt when it calls ItemBmpEJB at the first time, and there is not any lookup invocation later when calling ItemBmpEJB again. As a result, both of them should be configured with a fault-tolerant mechanism as a whole. Another application-specific constraint is that ItemBmpEJB's response time must be within 10 seconds because 10 seconds is long enough for a client waiting for a response.

When all fault-tolerant requirements and application-specific constraints are given, the problem is: which fault-tolerant mechanism is the most suitable one for ItemBmpEJB and OrderEnt, without any modifications on their source code? To solve the problem, we describe our approach step by step in the following sections.

3 Modeling Fault-Tolerant Mechanisms as Styles

In this section, we specify the structural and behavioral characters in fault-tolerant mechanisms as Fault-Tolerant Styles (FTSs) (Step 1.1 and 1.3 in Fig. 1). By surveying the literature on FT [2, 3], we derive some reusable parts that can be combined to form different FTSs. The well organized FTSs form the foundation of selecting a suitable mechanism for an application.

3.1 The Concept of Fault-Tolerant Styles

Although different fault-tolerant mechanisms are distinguished by their structure and interactions with functional components, their primary activities are similar. They control the messages passed in or received from a component, and monitor or control a component's states. An architectural style is a set of constraints on coordinated architectural elements and relationships among them. The constraints restrict the role and the feature of architectural elements and the allowed relationships among those elements in a SA that conforms to that style [21]. From the point of view of architectural style, entities in a fault-tolerant mechanism are modeled as components, interactions among the entities are be modeled as connectors, and constraints in a mechanisms are modeled as

Table 1. The stereotype definition for both fault-tolerant components and connectors in FTSA

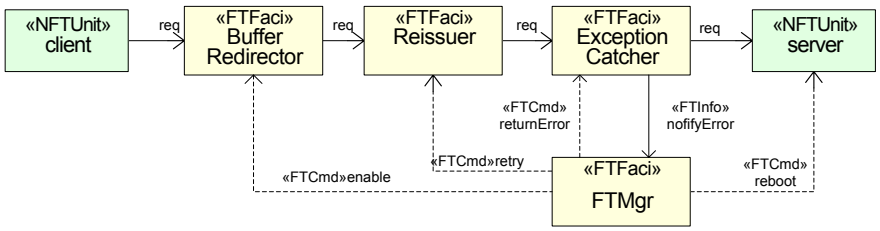
UML Stereotype	Type	Meaning
«FTUnit»	Component	Components that have fault-tolerant capabilities
«NFTUnit»	Component	Components that have not fault-tolerant capabilities
«FTFaci»	Component	Facility components that enable fault tolerance. By obeying specific structural constraints and coordination, «FTFaci» elements interact with «NFTUnit» elements, to make the latter fault-tolerable.
«FTCmd»	Connector	Changing states
«FTInfo»	Connector	Conveying information

an FTS [18, 26], which is a kind of architectural style. The architecture of a fault-tolerant application is a Fault-Tolerant Software Architecture (FTSA), which conforms to an FTS and tolerates a kind(s) of faults.

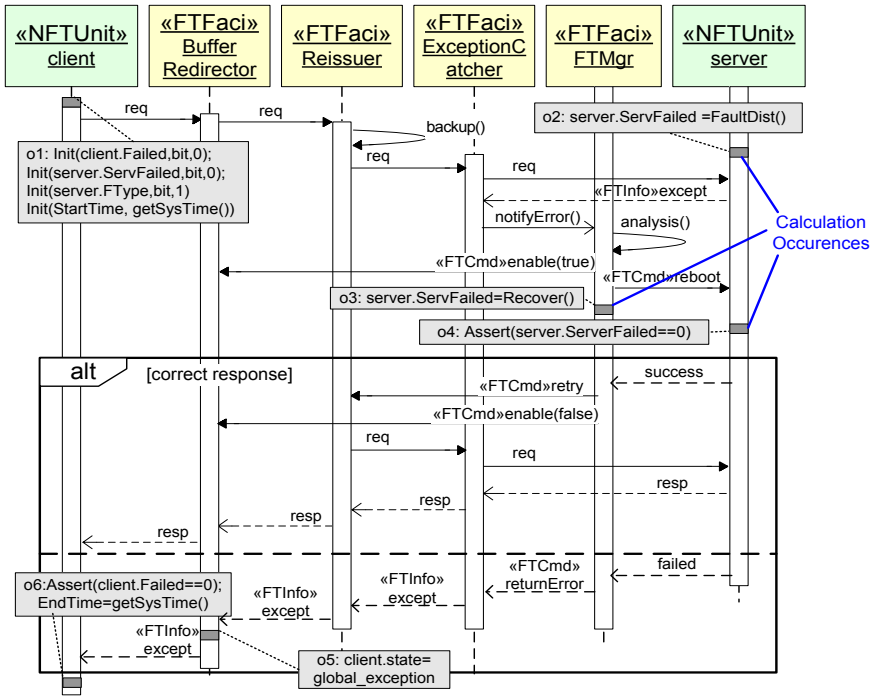
We use a UML profile for both SA and FT [20, 26] and made necessary extensions to specify FTSs and FTSAs (see Table 1), because UML is widely used and easy for communication. There are three kinds of components in the profile: *«NFTUnit»* components are business components without fault-tolerant capability; *«FTUnit»* components are business components with fault-tolerant capability either by its internal design or by applying a set of *«FTFaci»* components to an «NFTUnit» component. We define a stereotype «FTFaci» for well-designed and reliable components, which provide FT services for «NFTUnit»components. There are two kinds of connectors: «FTInfo» connectors are responsible for conveying a component's states to another; «FTCmd» connectors are responsible for changing an «NFTUnit» component's states. An «NFTUnit» component and its attached «FTFaci» components, which interact with each other in a specific manner, form a composite «FTUnit» component.

Based on the profile, we model fault-tolerant mechanisms as FTSs. Each mechanism's structure is modeled in UML2.0 component diagram (see Fig. 3 (a)). In order to model the mechanism's behavior in a similar manner, we use UML2.0 sequence diagram. UML2.0 sequence diagram provides a visual presentation for temporal relations among concerned entities, but its capability to explicitly specify internal states and states transition are limited. So we introduce a special *calculation occurrence* from the general *execution-occurrence* definition in UML2.0 sequence diagram. Execution-occurrences are rectangles drawn over lifelines in a sequence diagram, and represent the involvement of components in an interaction or scenario (see Fig. 3 (b)). A calculation occurrence is a special execution-occurrence to define, initialize or change variables in an interaction. The order of different calculation occurrences in a scenario stands for the temporal relations among interactions.

Micro-reboot mechanism [7] is an illustrative mechanism to be modeled as FTS. A Micro-reboot style consists of some «FTFaci» components (ExceptionCatcher, Reissuer, and BufferRedirector) and an «FTCmd» Reboot connector for an «NFTUnit» component (Fig. 3). The ExceptionCatcher catches all unexpected exceptions in the «NFTUnit» component. After the caught exceptions are analyzed and the failed component is identified, the failed component is rebooted. Meanwhile, the BufferRedirector blocks incoming requests for the component during recovery. When the failed component is successfully recovered, the BufferRedirector re-issues the blocked requests and the normal process is resumed.



(a) The structure of *Micro-reboot* style



(b) The behavioral of *Micro-reboot* style

Fig. 3. The structural (a) and behavior (b) specifications of Micro-reboot style

We also model other FTSs such as Simple Retry style and Retry Blocks style. Simple Retry style is similar to Micro-reboot style except it only re-invokes a failed component again, without rebooting it. Retry Blocks style is similar to Simple Retry style except it uses data re-expression component to mutate inputs before retrying. These FTSs are not shown in the paper due to the space limitations.

3.2 The Classification of Fault-Tolerant Styles

We identify some common and key «FTFaci» components and «FTCmd» connectors in Table 2, which can be used for third-party components. They are classified by their

functionalities: detecting errors, recovering error states, or smoothing the recovery procedures. A complete FTS consists of an error-detection part, a recovery part, and an auxiliary part (Some FTSs only have two parts or even one part, depending on different FT design principles). Combinations based on the above FT components or connectors form different FTSs. In Fig. 4, seven major FTSs are given by the combination of these entities. These FTSs stands for typical fault-tolerant mechanisms for third-party components. Recovery Blocks and N-Version Programming are design diversity-based mechanisms; N-Copy Programming and Retry Blocks are data diversity-based mechanisms; and Micro-reboot, Simple Retry, and Checkpoint-restart are environment (temporal) diversity-based mechanisms.

The identification of common «FTFaci» components and «FTCmd» connectors makes FTSs flexible. A FTS's error detection part or recovery part can be replaced by another if necessary. For example, Recovery Block style, which works on a primary component and a secondary component implementing same functions, requires an AcceptanceTest to determine whether the primary works correctly or not. But if the primary mainly thrown exceptions when a failure happened, AcceptanceTest is not good at dealing with such abnormal. It can be replaced by ExceptionCatcher, which

Table 2. Key modules in fault-tolerant mechanisms

Type	Component/ Connectors in FTSA	Explanation	Mechanism examples
Error Detection	«FTFaci» ExceptionCatcher	Catches thrown exceptions by a component	Micro-reboot.
	«FTCmd» Watchdog	Periodically sends a request to a component to testify its liveness.	Watchdog.
	«FTFaci» AcceptanceTest	Decides the correctness of a returned value	Recovery Blocks, Retry Blocks.
Recovery	«FTCmd» Reboot	Resets a failed component's states by reboot	Micro-reboot.
	«FTFaci» StateSetter	Set a component's states, according to given parameters.	Checkpoint-restart.
	«FTFaci» Switcher	Sends a request to a version/instance of a component.	Recovery Blocks.
	«FTFaci» DistributerCollector	Sends identical requests to multiple versions/instances of a component, and determines a result by comparing all the returned values.	N-Version Programming (NVP), N-Copy Programming (NCP)
Auxiliary	«FTFaci» DataReexpression	Slightly modifies an input	Recovery Blocks, NCP.
	«FTFaci» BufferRedirector	Buffers all requests to a failed component and redirects them when it is recovered	Micro-reboot.
	«FTCmd» Reissuer	Re-sends a request to the target component, after a varied waiting time	Checkpoint-restart.
	«FTFaci» FTMgr	Coordinates operations between error detector and recovery, or between several recovery operations	Used in All most all mechanisms.

is good at dealing with failures manifested as exceptions. The occasion of a replacement is usually decided by the time when a fault assumption is changed, and the verification of the correctness of the replacement is supported by our model checking approach described in Section 5 if only the correctness is also specified as a property.

The combined FTSs would be more plentiful if more «FTFaci» components or «FTCmd» connectors are included. But keep in mind that not all existing fault-tolerant mechanisms are meaningful for third-party components. Because implementation details of these components are hidden from application developers, and components' internal states are invisible except those accessed through a predefined interface. Most of the mechanisms in the classification can be externally attached to components. A notable example is checkpoint-restart mechanism. Almost all checkpoint-restart protocols require accessing a component's internal states. Considering an impractical assumption that all components provide an interface to get/set their internal states, checkpoint-restart can only apply to a considerably small number of third-party components that provide state manipulation operations. The similar situation exists in replication mechanism too.

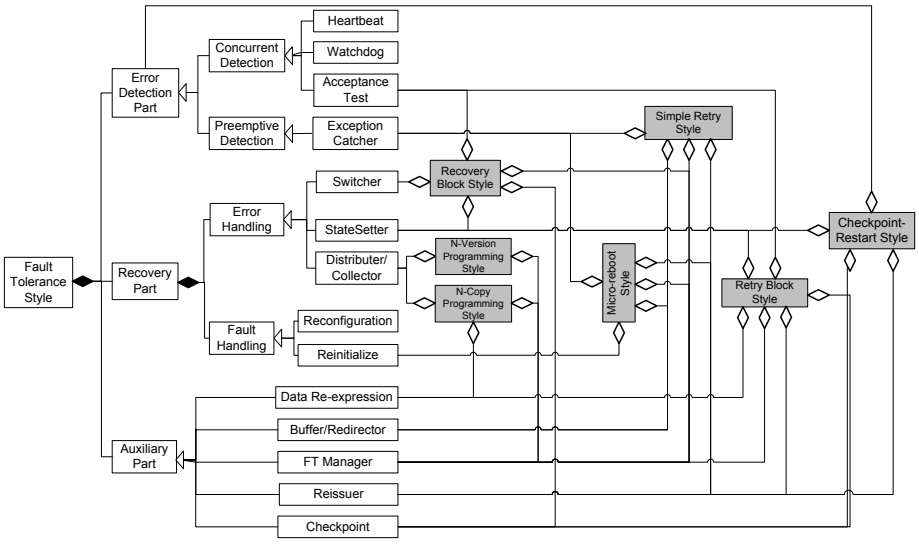


Fig. 4. The classification of FTSs according to their design principles. We assign the common «FTFaci» components and «FTCmd» connectors to a classification framework give by [3], and we show how to form a FTS by combining some of them.

4 Modeling Fault-Tolerant Properties and Application-Specific Constraints

4.1 Fault-Tolerant Properties and Application-Specific Constraints

Given a specific application, a set of requirements on fault-tolerant capabilities, and a set of candidate FTSs, it is critical to select the most suitable one for concerned

components in the application to meet the requirements. In this section, we abstract both fault-tolerant capability requirements and fault assumptions on components as fault-tolerant properties, and specify application-specific constraints (Step 1.2 in Fig. 1). In the next section, we translate an FTS's behavioral models in UML sequence diagram, the properties, and the constraints into verification models, and use model checking to verify the FTS's satisfaction of the properties and the constraints.

Table 3. Fault-tolerant properties and application-specific constraints

Type	Property Name & Description
Fault Assumptions	**Transient fault assumption (P_1)**: When a component is providing services and a transient fault is activated in it then, its states will be resumed if a fault-tolerant mechanism was applied. Transient faults are nondeterministic and are also called "Heisenbugs". **Environment-dependent and non-transient (EDNT) fault assumption (P_2)**: When a component is providing services and an EDNT fault is activated in it then, its states will be resumed if a fault-tolerant mechanism was applied. EDNT faults are deterministic and activated only on a specific environment. **Environment-independent and non-transient fault assumption (EINT) (P_3)**: When a component is providing services and an EINT fault is activated in it then, its states will be resumed if a fault-tolerant mechanism was applied. EINT faults are deterministic and are independent of specific environment.
Generic Fault-Tolerant Capabilities	**Fault containment (P_4)**: If an error is detected in a component, other components would not be aware the situation. **Fault isolation (P_5)**: When a failed component is being recovered, no new incoming requests can invoke the component. **Fault propagation (P_6)**: If an un-maskable fault is activated in a component and it cannot be recovered successfully, the client, who issues the request and activates the fault, would receive an error response. **Coordinated error recovery (P_7)**: If a global error, which affects more than one component, happened, the error can be recovered.
App-Specific Constraints	**Weak dependency (P_8)**: Component C_1 weakly depends on component C_2. **Strong dependency (P_9)**: Component C_3 strongly depends on component C_2. **Timely constraint (P_{10})**: A component always delivers correct response within 10 seconds.

Fault assumption is a kind of important properties. It assumes the characters of faults in a component or an application. Only when an FTS can deal with a certain kind of fault, it is meaningful to discuss the FTS's other capabilities. Properties P_1, P_2, and P_3 shown in Table 3 denote three fault assumptions. These three properties form a dimension of selecting FTSs. Then fault containment, fault isolation, fault propagation, and recovery coordination are four generic fault-tolerant capabilities. They are shown in Table 3 as P_4 to P_7 and form another dimension of the selection. P_4 stipulates that the source of a failure should be masked, for fear of error propagation. Because not all errors can be masked, property P_6 states that if a failed component cannot be recovered, the error should be allowed to propagate to others to trigger a global recovery processes. This is important for some faults that can be tolerated by coordinated recovery among several dependent components. P_5 stipulates that new

incoming requests cannot arrive at a failed component. P_7 is related to application-specific constraints P_8 and P_9, so it will be explained later. At last, application-specific constraints can also affect the selection of suitable FTSs. Properties P_8 to P_{10} describe some application-specific constraints. P_8 and P_9 stipulate weak and strong dependencies among components, respectively. Property P_{10} states a performance-related constraint: a component's response time must not be more than a certain time in all circumstances.

It should be noted that the above fault-tolerant properties only covers some important and typical ones, and they are distilled from a study of FT [3, 2]. Other properties, such as those presented in Yuan et al.'s study [13], can also be appended to the table.

4.2 Specifying Properties

Before formally specifying fault-tolerant properties, we define a base for FTSA. In an FTSA model, components $C=\{c_1, c_2, \ldots, c_n\}$, where each c_i $(1 \le i \le n)$ is a «NFTUnit» component. States of an component belong to a states set $S=\{normal, local\_error, global\_error\}$ and $\forall c \in C, c.state \in S$. Local errors are abnormal states that affect only one component and can be resumed by recovering this component individually, whereas global errors may affect more than one component and need to be recovered by coordination of all affected components. Each «NFTUnit» component has a variable *Failed* to indicate whether the component is failed or not, and *ErrorName* to indicate the name of an error.

$$\forall c \in C, c.Failed = \begin{cases} false, if\ c.state = normal \\ true, otherwise \end{cases}$$

The variable *ServFailed* is an alias of *Failed* in a component that provides services for others. Each «NFTUnit» component has an *Ftype* to indicate its fault assumption in a scenario. $\forall c \in C, c.Ftype \in \{transient, EDNT, EINT\}$. For each «NFTUnit» component, it is attached by a fault activation function *FaultDist*, which specifies faults' activation occasions and duration in the component decided by *Ftype* and the given failure-interval distribution. The function may transit a component's state from *normal* to *local_error* or *global_error*.

The above definition only covers FT-related issues. To represent application-specific constraints, we classify them as two types because application-specific constraints are various. Generally, if a constraint manifested as "*p implies q*", it is treated as fault-tolerant properties, like P_{10} in Table 3. If a constraint cannot be manifested as "*p implies q*", it would be treated as an extra modification of the state transition of FTSA. In our study, we take dependencies as an example of such constraints. It is one of important application-specific constraints in FT. For each c_1 and c_2 in C, we denote $c_1 \prec_w c_2$ if c_1 weak dependent on c_2, and $c_1 \prec_s c_2$ if c_1 strong dependent on c_2, otherwise, c_1 is independent from c_2 (i.e. there is no dependent relationships between them).

The lifecycle of a component is modeled as a process. The initial trigger of a components' state transition is a request for its service. A component's state transition function δ under the constraint of dependent relationship are: if $c_1 \prec_s c_2$,

$$\delta(c_1.state) = \begin{cases} FaultDist(c_1.Ftype), if\ c_1.state = normal\ and\ c_2.state = normal \\ c_1.state, \qquad if\ c_1.state\ != normal\ and\ c_2.state = normal \\ local\_error, \qquad if\ c_2.state\ != normal \end{cases}$$

Otherwise (i.e. $c_1 \prec_w c_2$ or c_1 is independent from c_2),

$$\delta(c_1.state) = \begin{cases} FaultDist(c_1.Ftype), if\ c_1.state = normal \\ c_1.state, \qquad otherwise \end{cases}$$

An FTS's recovery part has a *Recover* function to transit a component's error state to a normal one. «FTFaci» FTMgr holds a list *comp* containing components which is configured to be fault tolerable. The components in *comp* have an extra *recovering* state, which is visible only by the FTMgr, and a variable to indicate the name of an error.

Based on the above definitions, informally expressed fault-tolerant properties and application-specific constraints in Table 3 are formalized as:

***P*$_1$:** *client.*o_1*.Failed = false ^ server.*o_2*.ServFailed = true ^ server.*o_2*.Ftype = transient* $\Rightarrow$ *server.*o_4*.Failed = false*
***P*$_2$:** *client.*o_1*.Failed = false ^ server.*o_2*.ServFailed = true ^ server.*o_2*.Ftype = EDNT* $\Rightarrow$ *server.*o_4*.Failed = false*
***P*$_3$:** *client.*o_1*.Failed = false ^ server.*o_2*.ServFailed = true ^ server.*o_2*.Ftype = EINT* $\Rightarrow$ *server.*o_4*.Failed = false*
***P*$_4$:** *client.*o_1*.Failed = false ^ server.*o_2*.ServFailed = true* $\Rightarrow$ *client.*o_6*.Failed = false*
***P*$_5$:** *client.*o_1*.Failed = false ^ server.*o_2*.ServFailed = true ^ FTmgr.*o_3*.comp[server]. state = recovering* $\Rightarrow$ $\forall c \in C$, *c != server, c.ErrorName != FTmgr.*o_3*.comp [server].ErrorName*
***P*$_6$:** *client.*o_1*.Failed = false ^ server.*o_2*.ServFailed = true ^ server.*o_4*.ServFailed = true* $\Rightarrow$ *client.*o_6*.Failed = true*
***P*$_7$:** *client.*o_1*.Failed = false ^ server.*o_2*.ServFailed = true ^ server.*o_2*.state = global_error)* $\Rightarrow$ *client.*o_6*.Failed = false ^ server.*o_6*.Failed = false.*
***P*$_8$:** $c_1 \prec_w c_2$
P$_9$: $c_3 \prec_s c_2$
***P*$_{10}$:** *client.*o_1*.Failed = false ^ (server.*o_2*.ServFailed = false* $\vee$ *server.*o_2*.ServFailed = true)* $\Rightarrow$ *client.*o_6*.EndTime - client.*o_1*.StartTime <= 10.*

The o_1, o_2, o_3, o_4 and o_6 in the above formal specification are calculation occurrences in the extended sequence diagram (based on Fig. 3 (b)). The *StartTime* and *EndTime* variables in P_7 are user defined variables which record the time of starting a request and receiving a response.

5 Translating Behavioral Models and Properties into Verification Models

We verify FTSs' satisfaction of fault-tolerant properties and application-specific constraints by Spin model checker [12], because it is proven effective in many indus-

trial applications [12]. However a system to be verified in Spin must be modeled in Promela (Process Meta-Language). Programs in Promela cannot be visualized as UML diagrams, and are often called verification model.

We predefine a set of templates to automatically translate the extended UML2.0 sequence diagrams into Spin's verification model (Step 2 in Fig. 1). The automatic translation of standard elements in UML2.0 sequence diagram has been addressed in related literature [5]. Interaction elements in UML2.0 sequence diagram, such as timeline and message dispatch, are mapped to basic block or elements in Promela, such as process (*proctype*) and channel (*chan* element). Structured control operators in UML2.0 sequence diagrams, such as conditional execution and loop execution, are mapped to control-flow constructs in Promela, such as the selection statement (*if…fi*) and loop statement (*do…od*).

Calculation occurrences defined in our extension are mapped to code blocks in Promela processes, based on the position where the calculation occurrence is placed on. Variables defined in calculation occurrences are mapped to variables in Promela, and all of them are initialized in the Promela-defined init process. The variables may be re-assigned by *FaultDist* function that we defined to simulate faults, by *Recover* function that simulate a fault-tolerant mechanism, or by build-in functions such as obtaining system clock. The fault simulation function is mapped to a separate parameterized process that interacts with other processes.

Finally, conclusions in properties predicate are mapped to assertion statements in Promela. The positions of the assertions are decided by the quantifiers (universal or existential) of predicate and execution occurrences in which the conclusions covered. For example, a universal quantifier is prefixed to a component in the conclusion of property P_5 in Section 4.2, so the corresponding assertion should be placed in all processes corresponding to timelines of components, and its position in the processes must be after the position of the last execution occurrence (o_2) in P_5. For some application-specific properties that cannot be specified as predicates, such as P_8 and P_9, they are represented by affecting components' state transitions.

In model checking process (Step 3 in Fig. 1), Spin simulates an FTS's behavior and traverses all its states combinations. As we explained before, a component's states are defined and stored in variables defined in the calculation occurrences. These states are initialized at the beginning, and re-assigned by fault simulation function and state transit rules. When Spin control flow arrives at an assertion, it checks the truth or not of the assertion. It either confirms that the properties hold or reports that they are violated. A false assertion means the style does not preserve the property represented by the assertion and a counter-example is provided. Otherwise, the above verification process continues. When all assertions are true, it means the FTS satisfies all the concerned properties.

6 Utilization of the Approach for ECperf

In the previous sections, we explain how to specify fault-tolerant mechanisms that can be used for third-party components, how to specify fault-tolerant properties and application-specific constraints, and how to translate these specifications to verification model of Spin. In this section, we use the approach to selecting a suitable mechanism for ECperf.

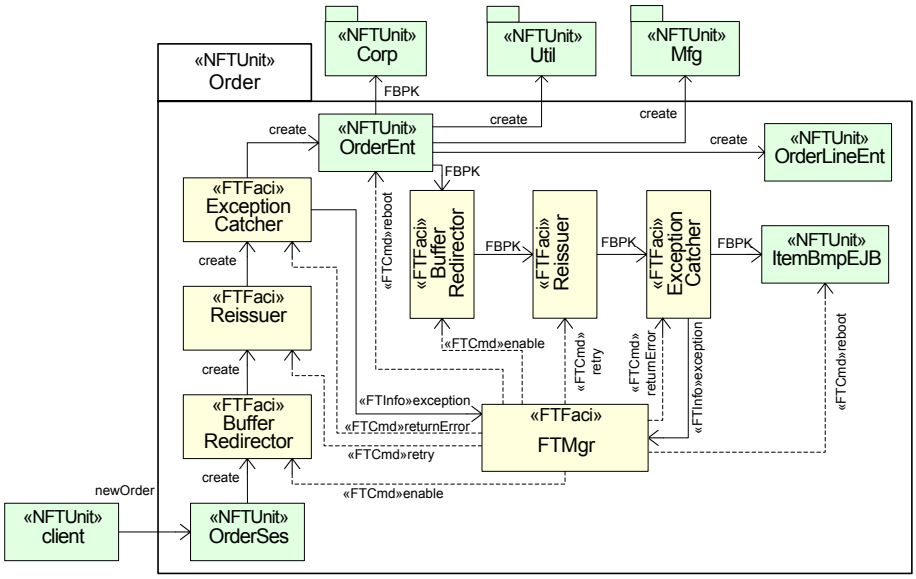


Fig. 5. The fault-tolerant architecture of ECperf that conforms to Micro-reboot style

Application-specific constraints can narrow the scope of candidate FTSs. ECperf runs on a sequential execution environment (JEE AS), so N-Version Programming style and N-Copy Programming style cannot be used because they require concurrent execution support. There exist no other variants of ECperf, so Recovery Blocks style cannot be used because it requires multiple alternatives for a same design. ItemBmpEJB does not provide interfaces to access its internal states, so Checkpoint and Restart style is excluded. Then the remaining candidates include Retry Blocks style, Simple Retry style, and Micro-reboot style. There are no more ECperf-specific characters help to select or exclude one of the above candidates. To select a suitable FTS from existing ones, we carry out a model checking process to verify if they satisfy fault-tolerant properties stated in Section 4.1 and ECperf-specific constraints.

We create three different versions of fault-tolerant ECperf by modifying its original SA. Each version conforms to one of the above three FTS (Fig. 5 shows the version conforming to Micro-reboot style). The behavioral models of the enhanced versions are translated into Promela programs, and we use Spin to verify their satisfactions on properties P_1 to P_7, and P_{10} listed in Table 3 in Create-a-New-Order scenario. To verify property P_{10}, we calculate the average response time of ItemBmpEJB with the help of runtime information, and estimate average response time of «FTFaci» components in Micro-reboot style. The result is shown in Table 4 and Micro-reboot style is the winner because it fits the EDNT fault assumption and supports coordinated recovery, but Retry Blocks style and Simple Retry style cannot.

We also perform a set of comparative experiments to validate the practical correctness of the selection. In the experiments, micro-reboot mechanisms and simple retry mechanism are attached to the components at the external, with the supports of the reflective JEE AS. We periodically inject Java exceptions into ItemBmpEJB to

simulate EDNT faults. As a result, the rates of successful submitted orders using Micro-reboot and Simple Retry are 87.3% and 50.7%, respectively, compare to 45.4% with no FT. It is clear that Micro-reboot style works better than Simple Retry style. The experimental result is coincident with the model checking result.

Table 4. The satisfaction of properties for Retry Blocks style, Simple Retry style, and Micro-reboot style. (●: preserve; ○: do not preserve). Fault assumptions form a dimension, other fault-tolerant properties and some application-specific constraints form another dimension. The results in the dash-line rectangle shows that Micro-reboot style satisfied all concerned properties and constraints under EDNT fault assumption, while Retry Blocks style and Simple Retry style cannot.

(a) Retry Blocks style

	P_4	P_5	P_6	P_7	P_{10}
P_1	●	●	●	●	●
P_2	○	○	○	○	○
P_3	○	○	○	○	○

(b) Simple Retry style

	P_4	P_5	P_6	P_7	P_{10}
P_1	●	●	●	●	●
P_2	○	○	○	○	○
P_3	○	○	○	○	○

(c) Micro-reboot style

	P_4	P_5	P_6	P_7	P_{10}
P_1	●	●	●	●	●
P_2	●	●	●	●	●
P_3	○	○	○	○	○

7 Discussion and Related Work

In the area of **Architecting Fault-Tolerant Systems** [1], components (computing entities), connectors (communication entities), and configuration (topology of components and connectors) have been used to model fault-tolerant software as FTSA. Previous work in the area mainly focus on how to model a specific fault-tolerant mechanism [10, 13, 14, 26, 27], for example, exception handling-based mechanism [14, 27]. A few studies consider the reasoning or analysis on an FTS. Yuan et al. [27] specify a Generic Fault-Tolerant Software Architecture (GFTSA), which obeys idealized Fault-Tolerant Component style, in formal language Object-Z, and perform manual formal proofs to demonstrate fault-tolerant properties the GFTSA preserves. The authors also present a template to automate the customization process when using the style. Sözer et al. [26] specify the structure of a local recovery style in an UML profile, and perform performance overhead and availability analysis. In contrast, we uniformly model and analysis various mechanisms that can be used for third-party components as fault-tolerant styles.

Verifying fault-tolerant software via model checking has been studied in previous work to prove the correctness of fault-tolerant design. Bernardeschi et al. [4] applies model checking to fault-tolerant software specified in Calculus of Communicating Systems (CCS)/Meije process algebra. Fault-tolerant properties are expressed in Action-based Computation Tree Logic (ACTL). Because a common prerequisite of model checking is that software model should be a formal one, this would be a barrier to its acceptance in practices. Some studies adopt another approach – after modeling software using popular modeling languages, translating the models into a formal one [8, 6]. Ebnenasir and Cheng [7] define a computation model and use UML state diagrams to specify fault, generic fault tolerant patterns, and fault-tolerant systems. The UML state diagram models are translated into Promela programs automatically. Thus the model is checked to verify its safety and liveness. The major difference between

their work and ours is that we concentrate on the specificity of fault-tolerant mechanisms but they focus on the generality of them. They define detector pattern and corrector pattern to covers all error detection mechanism and all recovery mechanisms. But a specific mechanism (Micro-reboot, checkpoint and restart, etc.) cannot be distinguished from others in such a generic definition.

We use UML2.0 diagram to model FTSs, and use Spin to verify both fault-tolerant properties and application-specific constraints because both are widely used and well studied. Other specification languages and model checkers can also be chosen to implement our approach, but it should be noted that the translation between FTS models and verification models is specific to the choice. For example, if we use a traditional ADL to specify FTSs' behaviors and a model checker's formal language to specify properties, the translation is not necessary. This alternative is not preferred because it not only requires application developers to learn a formal language, but also requires a manual modification of the verification model if the FTS is changed.

The scalability of model checkers is a major limit of the technique, because the explored states are restricted by the computational resources. But it is not a serious problem in our study. The verifications of fault-tolerant properties and application-specific constraints are carried out at SA level, and the number of states in FTSs is considerably small (*state*, *Failed*, *Ftype*, *comp*, etc.). Moreover, although the state may be large when taking application into consideration, the verification process is oriented to a scenario (for example, Create-a-New-Order scenario in the case study), in which the number of components is restricted.

We talk little about other general requirements (such as safeness and correctness of an FTSA). The reason is not in that these requirements are minor, but in that we focus on the selection of FTSs in the paper and these requirements do nothing helpful to the selection. In fact, we take them as prerequisites for all valid FTSs. This means these requirements must be met at first. Moreover, we touch the fringe of the interaction among FT and other software qualities like performance. Performance works as an application-specific constraint for the selection of FT styles. We believe similar solution is also helpful to make a trade-off between fault-tolerant styles and performance optimization patterns [14].

It is also abstractive to use more than one FTSs for a component in an application, thus the component is capable of tolerate many kinds of faults. But FT is an expensive measure that will impact other system qualities, such as performance. Multiple fault-tolerant mechanisms for a component will definitely impose heavy penalty to the application. Moreover, intervenes among multiple FTSs will also make the configuration of FTSA much more complex and error-prone.

8 Conclusion and Future Work

The proliferation of the development using third-party components brings new challenges to high availability or reliability because these components are often treated as black boxes. We present two levels of abstraction for the problem: fault-tolerant mechanisms that can improve third-party components' availability or reliability are abstracted as a Fault-Tolerant Styles; and fault assumptions, fault-tolerant capabilities, and application-specific constraints are abstracted as properties. The virtual of the

model checking based approach proposed in the paper is making the selection of FTS for a specific application more confidently. The approach is more meaningful when components' failure characteristics and execution environment change continually.

The work presented in the paper does not cover the problem of how to merge «FTFaci» components into an applications' architecture, which also affects the correctness of an FTSA. We assume the merge is performed by FT experts in the paper but we are working on an automatic model merging to do the task, similar to existing work [19, 22]. We are also going to finish the development of a GUI tool to integrate FTS modeling tool, translating tool, and Spin model checker.

Acknowledgments. This work is sponsored by the National Key Basic Research and Development Program of China (973) under Grant No. 2009CB320703; the Science Fund for Creative Research Groups of China under Grant No. 60821003; the National Natural Science Foundation of China under Grant No. 60873060; the High-Tech Research and Development Program of China under Grant No. 2009AA01Z16; and the EU Seventh Framework Programme under Grant No. 231167..

References

1. Workshop on Architecting Dependable Systems, http://www.cs.kent.ac.uk/wads/
2. Anderson, T., Lee, P.A.: Fault Tolerance: Principles and Practice. Prentice-Hall, Englewood Cliffs (1981)
3. Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. on Dependable and Secure Computing 1(1), 11–33 (2004)
4. Bernardeschi, C., Fantechi, A., Gnesi, S.: Model checking fault tolerant systems. Software Testing Verification and Reliability 12, 251–275 (2002)
5. Bose, P.: Automated Translation of UML Models of Architectures for Verification and Simulation Using SPIN. In: Proceedings of the 14th IEEE Int'l Conference on Automated Software Engineering, pp. 102–109. IEEE Computer Society Press, Los Alamitos (1999)
6. Brito, P.H.S., Lemos, R., Rubira, C.M.F.: Verification of Exception Control Flows and Handlers Based on Architectural Scenarios. In: Proceeding of the 11th IEEE High Assurance Systems Engineering Symposium (HASE), pp.177–186 (2008)
7. Candea, G., et al.: JAGR: an autonomous self-recovering application server. In: Proc. of the 5th Int'l Workshop on Active Middleware Services, Seattle, USA, pp. 168–177 (2003)
8. Ebnenasir, A., Cheng, B.H.C.: Pattern-Based Modeling and Analysis of Failsafe Fault-Tolerance. In: 10th IEEE International Symposium on High Assurance System Engineering (HASE), Dallas, Texas, USA, November 14–16 (2007)
9. ECperf webpage, http://java.sun.com/developer/earlyAccess/j2ee/ecperf/download.html
10. Garlan, D., Chung, S., Schmerl, B.: Increasing system dependability through architecture based self-repair. In: Proc. Architecting dependable systems. Springer, Heidelberg (2003)
11. de Guerra, P.A.C., Rubira, C.F., Romanovsky, A., de Lemos, R.: A fault-tolerant software architecture for COTS-based software systems. In: Proc. of ESEC/FSE-11, Helsinki, Finland, pp. 375–378 (2003)
12. Holzmann, G.J.: The Model Checker SPIN. IEEE Trans. on Software Engineering 23(5) (1997)

13. Issarny, V., Banatre, J.: Architecture-Based Exception Handling. In: Proc. of the 34th Annual Hawaii International Conference on System Sciences, vol. 9, p. 9058 (2001)
14. Lan, L., Huang, G., Wang, W., Mei, H.: A Middleware-based Approach to Model Refactoring at Runtime. In: Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC 2007) (2007)
15. de Lemos, R., Guerra, P., Rubira, C.: A fault-tolerant architectural approach for dependable systems. IEEE Software 23(2), 80–87 (2006)
16. Mei, H., Huang, G.: PKUAS: An Architecture-based Reflective Component Operating Platform. In: IEEE Int'l Workshop on Future Trends of Distributed Computing Sys. (2004)
17. Mei, H., Huang, G., Liu, T., Li, J.: Coordinated Recovery of Middleware Services: A Framework and Experiments. Int. J. Software Informatics 1(1), 101–128 (2007)
18. Muccini, H., Romanovsky, A.: Architecting Fault Tolerant Systems. Technical report, University of Newcastle upon Tyne, CS-TR-1051 (2007)
19. Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., Zave, P.: Matching and Merging of Statecharts Specifications. In: Proc. 29th Int'l Conference on Software Engineering, pp. 54–64 (2007)
20. Object Management Group, UML(TM) Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms, `http://www.omg.org/docs/ptc/04-09-01.pdf`
21. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. SIGSOFT Software Engineering Notes 17(4), 40–52 (1992)
22. Pottinger, R.A., Bernstein, P.A.: Merging models based on given correspondences. In: Proc. 29th int'l Conference on Very Large Data Bases, pp. 862–873 (2003)
23. Romanovsky, A.: A Looming Fault Tolerance Software Crisis? ACM SIGSOFT Software Engineering Notes 32(2) (2007)
24. Salatge, N., Fabre, J.C.: Fault Tolerance Connectors for Unreliable Web Services. In: Proc. of 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2007), Edinburgh, UK, pp. 51–60 (2007)
25. Seo, C., et al.: Exploring the Role of Software Architecture in Dynamic and Fault Tolerant Pervasive Systems. In: Proc. of SEPCASE 2007, Minneapolis, MN, USA (2007)
26. Sözer, H., Tekinerdogan, B.: Introducing Recovery Style for Modeling and Analyzing System Recovery. In: Proc. of 7th IEEE/IFIP Working Conference on Software Architecture, Vancouver, Canada, pp. 167–176 (2008)
27. Yuan, L., Dong, J.S., Sun, J., Basit, H.A.: Generic Fault Tolerant Software Architecture Reasoning and Customization. IEEE Trans. on Reliability. 55(3), 421–435 (2006)