# Model Driven Configuration of Fault Tolerance Solutions for Component-Based Software System

Yihan Wu[1,2], Gang Huang*[1,2], Hui Song[1,3], Ying Zhang[1,2]

1 Key Lab of High Confidence Software Technologies (Ministry of Education)
2 School of Electronic Engineering & Computer Science, Peking University, China
3 Lero: The Irish Software Engineering Research Center, Trinity College Dublin, Ireland
wuyh10@sei.pku.edu.cn, hg@pku.edu.cn(*corresponding author),
hui.song@scss.tcd.ie, zhangying06@sei.pku.edu.cn



**Abstract.** Fault tolerance is very important for complex component-based software systems, but its configuration is complicated and challenging. In this paper, we propose a model driven approach to semi-automatic configuration of fault tolerance solutions. At design time, a set of reusable fault tolerance solutions are modeled as architecture styles, with the key properties verified by model checking. At runtime, the runtime software architecture of the target system is automatically constructed by the code generated from the given architectural meta-model. Then, the impact of each component on the system reliability is automatically analyzed to recommend which components should be considered in the fault tolerance configuration. Finally, after which components are guaranteed by what fault tolerance solution is decided by the system administration, the architecture model is automatically changed by merging with the selected fault tolerance styles and finally, these changes are automatically propagated to the target system. This approach is evaluated on Java enterprise systems.

**Keywords:** fault tolerance, component-based system, dynamic configuration, mode driven approach, software architecture.

## 1 INTRODUCTION

Fault tolerance is well studied and practiced in the past decades. For different types of systems or different sources of faults, we need different fault tolerance solutions [15]. For example, if the fault is caused by temporary race between the current re-quests, only re-issuing the requests will significantly decrease the rate of fault response. Alternatively, if the fault is caused by an accumulated reason, such as the memory leak, rebooting the system or a part of it is usually necessary. Such fault tolerance solutions consist of different mechanisms for detecting the faults, buffering the requests, rebooting the components, recovering the responses, etc.

In today's popular component-based systems, fault tolerance solutions themselves also become more componentized, that is, fault tolerance mechanisms are implement-

ed as a set of reusable components by the component framework and can be configured to guarantee or ignore the given system components.

However, it is not easy to properly reuse the fault-tolerance solutions in complex component-based systems. The challenge is twofold. One is how to specify the reusable fault tolerance solutions on a specific platform. From the structural aspect, the specification should make clear the types of components required by the solution, the property of each component, and the relation between these components. The difficulty here is how to ensure the automated deployment, and in the meantime make the specification easy to understand. From the behavioral aspect, the specification should make clear the proper context for each solution, i.e., what type of faults the solution fits for, and the effect after deploying the solution. The difficulty here is how to classify the faults and how to verify the effect before really deploying the solution. Having the proper specification of fault-tolerant solutions, the second challenge is how to deploy them automatically. The first problem here is how to assist the system administrators choosing the part of the system to deploy the solution, and the proper solution to deploy. After choosing the solution, the remaining problem is how to automatically deploy and configure the reusable fault tolerance mechanism according to the solution.

In this paper, we present a model driven approach to specification and semi-automatic configuration of fault tolerance solutions for component-based systems, on the software architecture level. Based on our initial idea of supporting fault tolerance at software architecture level with the help of middleware [6], and an existing framework named SM@RT [2] [24] to support runtime model, we provide a systematic and automated framework with the help of runtime models, called SM@RT. The whole approach is divided into two phases. In the specification phase, the experts of the given system or platform define the fault tolerance mechanisms implemented by the system, in the form of a specific kind of components named fault tolerance facilities. Based on the components, the experts specify the reusable fault tolerance solutions as partial architectures composed by some of the existing facilities. The experts also list the fault tolerance properties satisfied by each of the solutions, as a reference indicating what kind of faults is proper to be fixed by this solution. In this phase, our framework provides the code generation support to wrap low-level fault tolerance mechanisms as reusable facilities, the meta-model to construct the partial architecture, and the model checking support to verify if the solution satisfies the declared fault tolerance properties. In the configuration phase, our framework helps the system administrators to semi-automatically deploy the proper fault tolerance solutions on the system. Specifically, our framework first reflects the system as runtime software architecture, and then uses this runtime architecture to calculate the key component that has the maximal influence to the global system reliability. With these two pieces of information as references, the administrator evaluates the type of faults, and chooses the proper solution. Finally, our framework automatically deploys the solution to the system, by merging the current architecture with the partial architecture specified by the solution, and then calculating and executing the required changes between the original and the result architecture.

The main contributions of this paper can be summarized as follows. Firstly, we analyze the component's impact on system reliability, and recommend key component(s). Secondly, we realize the model merging of runtime software architecture with fault tolerance style automatically. Thirdly, a systematic and semi-automated configuration framework is proposed, which is used to configure the fault tolerance solutions into component-based systems.

The rest of this paper is organized as follows: section 2 gives an overview of our approach and a motivating example of fault tolerance for an EJB component. Section 3 describes the concept of FTS (fault tolerance style) and the verification of FTS by model checking. Section 4 describes the details of analyzing the key component(s), selecting FTS, configuring RSA (runtime software architecture) with FTS, and propagating RSA changes to the target system. Section 5 shows how to use the approach to solve the problem in the motivating example. Section 6 shows the related work, and section 7 shows the discussion and future work on our approach.

## 2 APPROACH OVERVIEW

In this section, we first illustrate the fault-tolerance solutions on a real component-based system. Based on this example, we give a brief overview of the complete approach for modeling and configuring the fault tolerance solutions.

### 2.1 Illustrative Example

ECperf [12] 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 a Software Architecture (SA) model to depict the relations among EJBs in this scenario in Fig.1 (NFTUnit is none-fault-tolerant component). We assume these EJBs are black boxes. The structural model of ECperf comes from runtime information analysis, with the monitoring support provided by SM@RT.

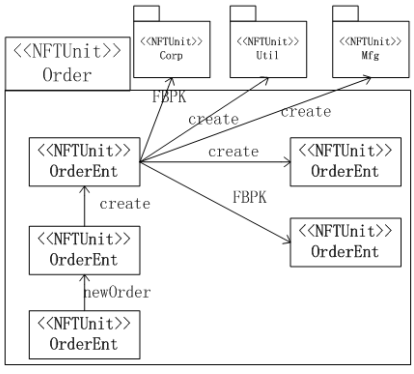


**Fig. 1.** The SA of ECperf in the scenario of Create-a-New-Order

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. The availability of *ItemBmpEJB* may be imperiled by database faults. 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-tolerance requirement is to make *ItemBmpEJB* capable of tolerating environment-dependent and non-transient (EDNT) faults.

### 2.2 Approach Overview

For the above example, the Ecperf is without any fault tolerance function. Our approach for fault tolerance configuration has four steps as follows:

(1) Locating the key component(s). Component is the basic unit of component based system, and the key component is the one whose reliability matters the most to the reliability of the whole system. In this paper, we use a scenario-based reliability analysis approach to analyze the reliability of the system and locate the key component.

(2) Selecting suitable FTS. To alleviate the difficulty in the selection of the fault-tolerance mechanisms, these mechanisms are abstracted as FTS at first. Then the required fault-tolerance capabilities are specified as fault-tolerance properties, and the satisfactions of the required properties for candidate FTSs are verified by model checking [6]. The system administrator just needs to input the fault tolerance capabilities which need to be satisfied.

(3) Configuring RSA with FTS. In this step, we perform fault tolerance by model merging at the architecture level. The two models which are merged are RSA and FTS. The components in RSA which need to be configured are analyzed in step 1. And the suitable FTS is chosen in step 2.

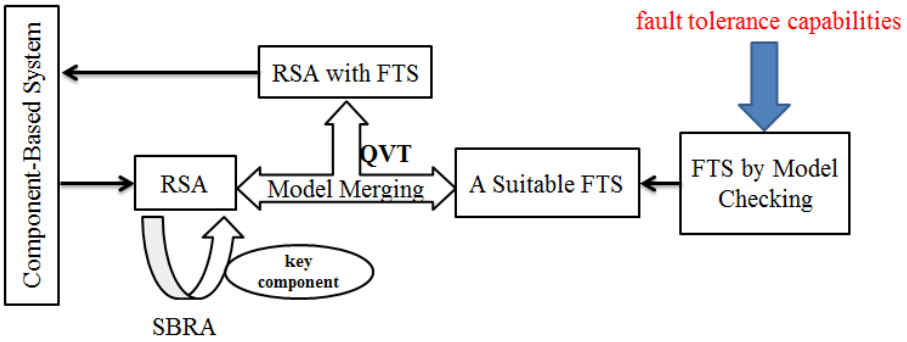


**Fig. 2.** Model driven configuration of fault tolerance solution

(4) Propagating the RSA changes to the target system. After step 3, we have performed fault tolerance at architecture level. For the sake of realizing fault tolerance,

we use SM@RT [2][24] to propagate the change to the target system. SM@RT provides a domain-specific modeling language and a code generator to support model-based runtime system management. Figure 2 shows the whole process of our approach.

# 3 SOLUTION MODELING

## 3.1 The Concept of Fault-Tolerance Styles

The primary activities of different fault-tolerance mechanisms are similar. They control the messages passed in, 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]. Entities in a fault-tolerance mechanism are modeled as components, interactions among the entities are modeled as connectors, and constraints in a mechanism are modeled as an FTS, from the point of view of architectural style. The architecture of a fault-tolerant application is a Fault-Tolerance Software Architecture (FTSA), which conforms to an FTS and tolerates a kind of fault.

## 3.2 Modeling Solutions as Fault-Tolerant Styles

We use a UML profile for both SA and FT [20, 25] and made necessary extensions to specify FTSs and FTSAs. There are three kinds of components in this UML profile: «NFTUnit», «FTUnit» and «FTFaci» components. «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. An «NFTUnit» component and its attached «FTFaci» components, which interact with each other in a specific manner, form a composite «FTUnit» component. There are two kinds of connectors: «FTInfo» and «FTCmd» connectors. «FTInfo» connectors are responsible for conveying a component's states to another. «FTCmd» connectors are responsible for changing an «NFTUnit» component's states.

Based on the profile, we model fault-tolerant mechanisms as FTSs. Each mechanism's structure is modeled in UML2.0 component diagram. Micro-reboot mechanism [7] is an illustrative mechanism to be modeled as FTS. A Micro-reboot style consists of four «FTFaci» components (*ExceptionCatcher*, *Reissuer*, *FTMgr*, 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.

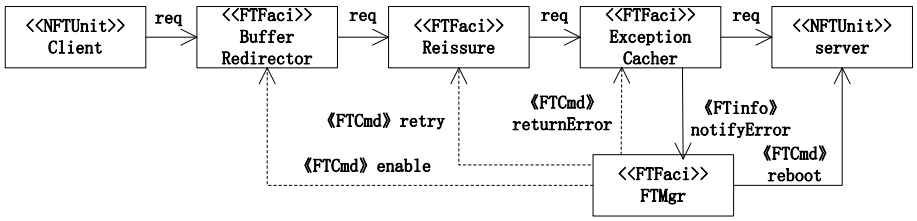


**Fig. 3.** The component diagram of Micro-reboot style

### 3.3 Validation of Solutions

In this section, we abstract both fault-tolerant capability requirements and fault assumptions on components as fault-tolerant properties. And then we translate a FTS's behavioral models in the 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 1.** Fault assumption and generic fault-tolerant capabilities

| Type | Property Name & Description |
|---|---|
| Fault Assumption | **Transient fault assumption (P1)**: 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 (P2)**: 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) (P3)**: 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 (P4)**: If an error is detected in a component, other components would not be aware of the situation.<br>**Fault isolation (P5)**: When a failed component is being recovered, no new incoming requests can invoke the component.<br>**Fault propagation (P6)**: 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.<br>**Coordinated error recovery (P7)**: If a global error, which affects more than one component, happened, the error can be recovered. |

**Fault-Tolerant Properties**

Fault assumption 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 P1 to P3 shown in Table 1 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 1 as P4 to P7 and form another dimension of the selection.

For fear of error propagation, P4 stipulates that the source of a failure should be masked. P5 stipulates that new incoming requests cannot arrive at a failed component. Because not all errors can be masked, property P6 states that if a failed component cannot be recovered, the error should be allowed to propagate to others to trigger a global recovery process. This is important for some faults that can be tolerated by coordinated recovery among several dependent components. P7 means that both of the failed component and the components which depend on it need to be recovered.

It should be noted that the above fault-tolerant properties only cover some important and typical ones, and they are distilled from a study of FT. Other properties, such as those presented in Yuan et al.'s study [22], can also be appended to the table.

**Verification of FTS**

We verify the FTS by translating the intuitive behavior description into the formal specification in Promela, and then evaluate the formal on by the model checker SPIN.

We predefine a set of templates to automatically translate the extended UML2.0 sequence diagrams into Promela. The automatic translation of standard elements in UML2.0 sequence diagram has been addressed in related literature [23]. 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). And the details are illustrated in our previous work [6].

### 3.4 Solutions provided by Java Application Server

Using the specification mechanism, we summarized four FT solutions on a concrete system type, the Java Application Server. These solutions are widely used in the JEE systems, and the reusable facilities constituting them are able to implement based on the JEE techniques. An experiment implementation of all these facilities on the JBoss application server can be found in our previous work [9]. And the four solutions are listed as follows:

- Simple retry style: send the failed request again.
- N-copy programming style: send a request to several instances of a component, avoiding the failure of a few instances.
- Micro reboot style: initialize the failed component and recover it to original state.

- Retry block style: send the request to a component instance. If the return result is an error, modify the request, restore the environment state and resend the request.

# 4 SOLUTION CONFIGURATION

## 4.1 Construct Runtime Software Architecture

As our fault-tolerant approach is performed at the architecture level, we need to do the following steps to get the runtime system information:

1. Define the meta architectural model of the target system.
2. Define the access model of the target system.
3. Generate the code and instantiate the RSA.

The meta-model of the target system defines the structure of RSA, including property, class, and association between classes. The access model defines the methods which used to get the runtime system information. The method *get()* is used to get the system information and *set()* is used to modify the system properties. In this paper, we use SM@RT to define the access model, generate the synchronization code, and instantiate the RSA.

## 4.2 Analyze Component's Reliability Impact

In this section, a scenario based reliability analysis approach is described, and then we introduce a SBRA-based algorithm to find out the key components which have the crucial influence to the system.

**Scenario-Based Reliability Analysis Approach**

SBRA is a reliability analysis technique for component-based software, which was proposed by Sherif Yacoub et al. in [4]. Using scenarios of component interactions, they construct a probabilistic model named Component-Dependency Graph (CDG). Based on CDG, a reliability analysis algorithm is developed to analyze the reliability of the system as a function of reliabilities of its architectural constituents.

A CDG is defined as follows:

$CDG=<N,E,s,t>$

$N=\{n\}$,which is a set of nodes in the graph; $E=\{e\}$,which is a set of directed edges in the graph; $s$ and $t$ are the start and termination nodes.

$n=<NC_i, RC_i, EC_i>$ $n \in N$, models a component $C_i$, $NC_i$ is the name of component $C_i$, $RC_i$ is the reliability of component $C_i$, and $EC_i$ is the average execution time of the component $C_i$.

$e=<T_{ij}, RT_{ij}, PT_{ij}>$ $e \in E$, models the control flow transfer from one component to another. $T_{ij}$ is the transition name from node $n_i$ to $n_j$, denoted $< n_i, n_j >$, $RT_{ij}$ is the transition's reliability, and $PT_{ij}$ is the transition's execution probability. Fig. 4 shows the CDG of a system consisting of four components.

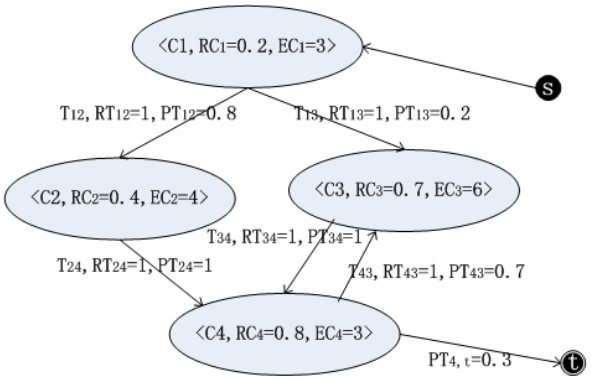


**Fig. 4.** A sample CDG

A CDG is an input parameter of SBRA, and the other input is $AE_{appl}$, which is the average execution time of the application. SBRA is shown in Fig.5. The output is $R_{appl}$, the reliability of the application.

***Parameters***
Consumes CDG, $AE_{appl}$
Produces $R_{appl}$
***Initialization:***
$R_{appl}$=0;Time=0;$R_{temp}$=1;
***Algorithm***
Push tuple<$C_1$, $RC_1$, $EC_1$>, Time, $R_{temp}$
While Stack not EMPTY do
Pop <$C_i$, $RC_i$, $EC_i$>, Time, $R_{temp}$
if Time>$AE_{appl}$ or $C_i$=t;
$R_{appl}$+=$R_{temp}$;
else
<$C_j$,$RC_j$,$EC_j$> ∈children($C_i$)
push(<$C_j$,$RC_j$,$EC_j$>,Time+=$EC_i$,
$R_{temp}$=$R_{temp}$*$RC_i$*$RT_{ij}$*$PT_{ij}$)
end
end while

**Fig. 5.** SBRA

***Parameters***
Consumes CDG, $AE_{appl}$
Produces
componentList <componentName, $R_{appl}$>
***Initialization:***
$R_{appl}$=0;
for each components $RC_i$=0.8;
***Algorithm:***
for each component <$C_i$,$RC_i$,$EC_i$>
$RC_i$=RCi+0.2;
$R_{appl}$=SBRA();
componentList.add(<Ci,$R_{appl}$>);
$RC_i$=0.8;
for <$C_i$,$R_{appl}$> in componentList
sorted by $R_{appl}$ decending;

**Fig. 6**. Select Key Components

**SBRA-based Algorithm for Selecting Key Components**

The reliability of an application is affected by several attributes in SBRA, such as the reliability and use frequency of each component, which means some components have a stronger influence on the reliability of the whole system than others. The frequency of each component depends on the scenarios. Several techniques have been proposed to estimate the reliability of software components, such as fault injection,

testing, and retrospective analysis. In order to discover the key components, we value the reliability of components statically and run the SBRA-based algorithm in Fig.6.

In the above algorithm, *SBRA()* returns the reliability of the whole system, which is calculated from the CDG. We assume the reliability of each component is 0.8 at first. Before invoking SBRA each time, the current component's reliability is improved to 1. So the value of $R_{appl}$ is the reliability of the system after the current component's reliability improved. That means the component with the maximum value of $R_{appl}$ is the key component.

### 4.3 Select proper solutions

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. We assist the system administrators in selecting the proper solutions by providing them the following to guidance: fault assumptions and fault-tolerant capabilities.

**(a) Simple retry style**

| | P4 | P5 | P6 | P7 |
|---|---|---|---|---|
| P1 | √ | √ | √ | × |
| P2 | × | × | × | × |
| P3 | × | × | × | × |

**(b) N-copy programming style**

| | P4 | P5 | P6 | P7 |
|---|---|---|---|---|
| P1 | √ | × | √ | × |
| P2 | √ | √ | √ | √ |
| P3 | × | × | × | × |

**(c) Micro-reboot style**

| | P4 | P5 | P6 | P7 |
|---|---|---|---|---|
| P1 | √ | √ | √ | √ |
| P2 | √ | √ | √ | √ |
| P3 | × | × | × | × |

**(d) Retry blocks style**

| | P4 | P5 | P6 | P7 |
|---|---|---|---|---|
| P1 | √ | √ | √ | √ |
| P2 | × | × | × | × |
| P3 | × | × | × | × |

**Fig. 7.** The satisfaction of properties for Simple retry style, N-copy programming style, Micro-reboot style, and Retry blocks style. ( √ : preserve; × : do not preserve). Fault assumptions form a dimension; other fault-tolerant properties form another dimension.

In model checking process, Spin simulates a FTS's behavior and traverses all its states combinations. A component's states are defined and stored in variables. 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. The result in Fig.7 shows four fault-tolerant styles' satisfaction to fault-tolerant properties.

### 4.4 Configure RSA with fault-tolerant styles

The fault-tolerant styles define the topological structure and behavior restriction of fault-tolerant components and the business components. For the sake of implementation of fault tolerance at the architecture level, we just need to merge the RSA with a suitable FTS. And the change can be propagated to the system by the modification on middleware. This process is accomplished automatically, which avoids configuration errors and reduces the burden of system architects. The inputs of this kind of configu-

ration are components which need to be configured, a selected FTS and the RSA of the application. The output is a fault-tolerant runtime software architecture.

The key of the configuration process is the automatic composition of RSA with FTS. In this paper, it is achieved based on Model Merging or Model Composition [18] [19]. Model Merging is a special kind of model transformation, and the function of Model Merging is merging two models MA and MB, conforming to meta-model MMA and MMB, respectively, and the result is MC, conforming to meta-model MMC [19]. MA is called Receiving Model, MB is called Merged Model, and the merging process is to merge the elements in MB into MA, and produce a Resulting Model MC. In general, there are two phases in Model Merging: comparison and merging. In the first phase, it needs to determine the match relationship automatically between the elements in Receiving Model and the elements in Merged Model. The second step, merging, adds the elements in Merged Model to the Receiving Model automatically in light of the match relation.

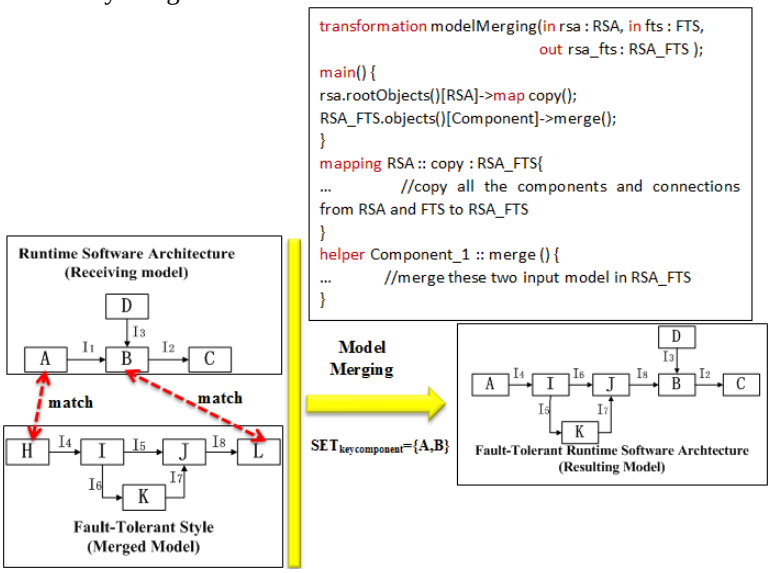


**Fig. 8.** The merging of RSA with FTS and the QVT implementation

In this paper, the Merged Model is FTS, and the Receiving Model is RSA. The Resulting Model is fault-tolerant runtime software architecture. The process is illustrated in Fig.8, and we use QVT (Query/View/Transformation) to implement the merging, which is a standard set of languages for model transformation defined by the Object Management Group.

### 4.5 Propagate RSA changes to the target system

For the sake of getting the real system with fault tolerant, we realized the following steps: firstly, we provide fault-tolerant sandboxes for application components. And then, encapsulate the operations which are used to add (remove) a component or a

connection between two components. Thirdly, we use QVT to realize model comparison [11], which is used to compare the original RSA with FTSA. And the comparison result will guide the modification of the target system.

# 5 EVALUATIONS

In this section, we use the approach to configure ECperf with fault-tolerant solutions.

## 5.1 Select Key Components on ECperf

We obtain the CDG of ECperf via runtime information analysis, with the monitoring support provided by a reflective JEE Application Server (AS), which is shown in Fig.9. After executing the SBRA, the result is shown in Fig.10, the component name as the abscissa, and the ordinate is the reliability of the system after the reliability of the corresponding component improved.

Figure 10 shows that there are 3 key components: ItemEnt, OrderSes, and OrderEnt. The reliability of the application will be maximized, if the reliability of these three components is improved. The component ItemEnt is invoked 403 times in the process of creating a new order, while others are invoked no more than 10 times. And ItemEnt is invoked by OrderEnt, which is invoked by OrderSes. There is a strong dependence between them. So it is easy to understand the three components are key components. The QVT code of SBRA can be downloaded from our google code project[3].

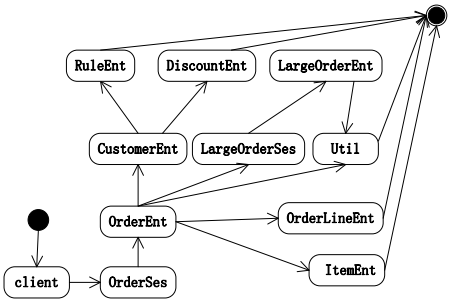


**Fig. 9.** The CDG of ECperf

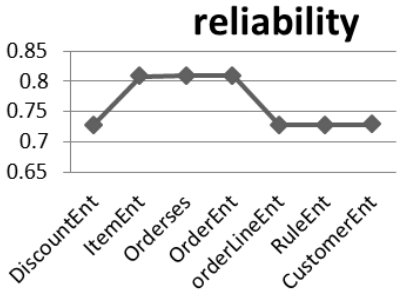


**Fig. 10.** The reliability of system

## 5.2 Select FTS for the Key Component

ECperf runs on a sequential execution environment (JEE AS), so N-Copy Programming style cannot be used because they require concurrent execution support. 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 proper FTS from existing ones, we select the most suitable FTS by applying the procedure presented in 4.3.

The fault assumption of the three components is EDNT. And we need the FTS to satisfy the property P4-P7. The result is shown in Fig.7. And Micro-reboot style is the winner because it fits the EDNT fault assumption and supports all the properties, but the other three styles cannot.

### 5.3 Merge Ecperf with FTS

In section 5.2, the set of key components is acquired, and $SET_{key\text{-}comp}$={ItemEnt, OrderSes, OrderEnt }. In section 5.3, we find out that the micro-reboot style is suitable. Each element of the $SET_{key\text{-}comp}$ corresponds to the component "server" in FTS. As there are three components that need to be configured, the merging process has three steps. And the match relationship is shown in Fig.11.

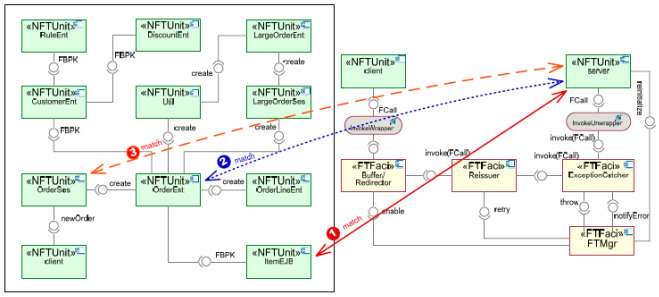


**Fig. 11.** The match relationship of RSA and FTS

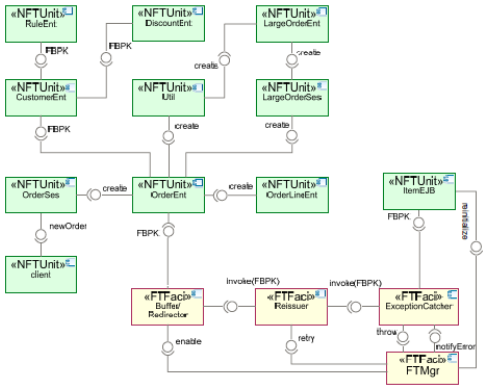


**Fig. 12.** The software architecture after first merging

The first step is to configure ItemEnt with micro-reboot style. In this process, "ItemEnt" corresponds to the "server", and "OrderEnt" corresponds to the "client". And the invocation between OrderEnt and ItemEnt disappeared. The result is shown in fig 12. The next two steps are similar. And we implement the model merging by QVT, the source code can be downloaded in our google code project [3];

We create three different versions of fault-tolerant ECperf by modifying its original SA. Each version conforms to one of the above three FTS. 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 as external utility mechanisms, with the supports of SM@RT. 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 (Fig.13). It is clear that Micro-reboot style works better than Simple Retry style. The experimental result is consistent with the model checking result. It should also be minded that the fault tolerance induces performance penalty. When no exceptions are injected into the experiments, the response time is 19.12s with no FT, and 21.71s with micro-reboot (Fig.13), which increases 13.49% of response time on average.

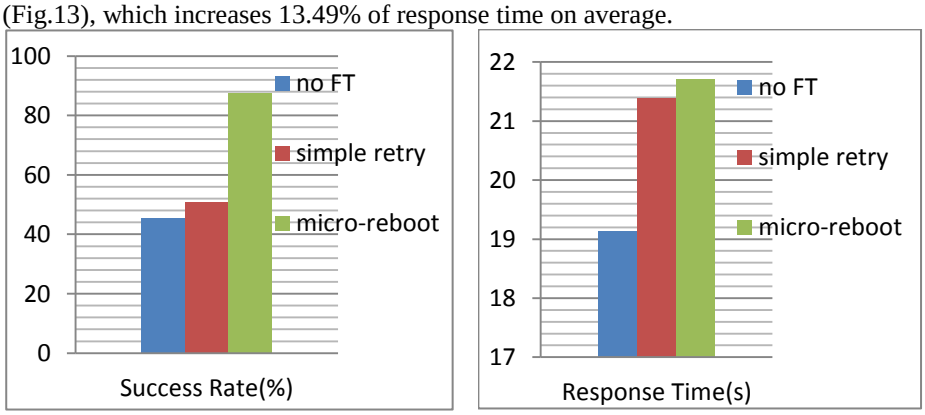


**Fig. 13.** The comparison success rate and response time

### 5.4 Discussion

From the evaluations of ECperf, we can see that the whole configuration process is more automatic than our previous work [6, 9], system advocates just need to specify the fault-tolerant properties that the target system needs to satisfy. In this section, we have a discussion after the experiment.

**FTS Specification.** We have described four fault tolerance solutions in this paper, and abstracted them into FTS: micro-reboot style, simple retry style, N-copy programming style, and retry block style. There are still some other solutions (such as Recovery Blocks style, N-Version Programming style, etc.), and we can leave them as future work.

**Key Component Recommendation.** In this paper, we use SBRA to estimate the reliability of the whole system, and recommend key components. And the analysis result is tallied with the actual situation. We choose SBRA mainly because it is a typical method for component-based system with a "CDG" model, which is more in

favor of the analysis at architecture level. Some other component-based algorithms, such as the K. Goseva-Popstojanova's approach [27], can be integrated into our framework, by specifying the process in QVT.

**Configuration Framework** For a complex component-based software system, the fault tolerance configuration process is complicated and challenging as follows: the components which need configuration are indeterminate; the fault-tolerant solutions are undefined; and the configuration process is without a guide. In this paper, we successfully handle these problems at architecture level: we abstract the target system into SA, which helps to identify the key component; we abstract the fault tolerance solution into FTS, which helps to check the fault tolerance properties and the selection of solutions; and the configuration process is under the model merging's guidance. In common cases, users only need to choose the FTS and target component, based on our recommendation. No future configuration or coding work is required. If users want to define their own FTS, integrate other analysis algorithms, etc., they just use the MOF standard languages to define their extension work at the model level.

## 6 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 focuses on how to model a specific fault-tolerant mechanism [10, 13, 14, 25, 26], for example, exception handling-based mechanism [14, 26]. A few studies consider the reasoning or analysis on an FTS. Yuan et al. [26] specify a Generic Fault-Tolerant Software Architecture (GFTSA), which obeys idealized Fault-Tolerant Component style, in formal language Object-Z, and performs 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. [25] 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.

The study of Architecting Fault-Tolerant Systems aims to achieve better fault-tolerant software by including FT in earlier development phase to bridge the gap between the requirement to build dependable software systems and the implementation to deal with failures in the software. As one of the important fault-tolerance mechanisms, exception handling is widely used in the study of architecting fault-tolerant software systems. A notable study is the CORRECT project [8] in Luxembourg, which introduces the Coordinated Atomic Actions (CAAs) mechanism in SA specification phase. The resulting SA specification with fault-tolerance notations is transformed into CAAs model automatically and further, transformed to an implementation framework. The output of such approach is a skeleton code that satisfies the functional and fault-tolerant requirements. It is based on model-driven techniques, and

separates the function design from the fault-tolerant design of the system in the model layer. However, it only supports a specific fault-tolerant technology without any others. The approach in this paper abstracts the fault-tolerant technologies as FTS, and identifies the components which need to be configured automatically, which means the assemblers just need to input the properties which need to be satisfied.

## 7 CONCLUSION AND FUTURE WORK

This paper proposes a model driven configuration of fault tolerance solution for component-based system. It needs to be polished in the future. At one side, the approach just provides an enablement to satisfy the fault-tolerant capabilities in the situation of a fault assumption. What capabilities should be satisfied and the type of fault assumption are given by developers. Therefore, how to facilitate the use of the enablement is critical for practice. For example, a powerful exception analysis support can alleviate the burden of developers in identifying which type of fault assumption the fault belongs to. The fault detection mechanisms help to decide what kind of fault-tolerant capabilities should be satisfied. On the other side, the fault-tolerant configuration may be more general and efficient. However, since the number of popular middleware in a period is relatively few, we argue that making a concrete middleware more powerful on exception handling is more important. This study is being carried out now, with the help of our Runtime Software Architecture [5][24].

ACKNOWLEDGMENTS. This work is supported by the National Basic Research Program of China (973) under Grant No. 2009CB320703; the National Natural Science Foundation of China under Grant No. 61121063, 60933003; the European Commission Seventh Framework Programme under grant no. 231167; the Science Foundation Ireland grant 10/CE/I1855 to Lero; and the NCET.

## References

1. Workshop on Architecting Dependable Systems, http://www.cs.kent.ac.uk/wads/
2. SM@RT: Supporing Models at Run-Time, http://code.google.com/p/smatrt/
3. SM@RT fault tolerance configuration, http://code.google.com/p/smatrt-ftc/downloads/list
4. Sherif Y., Bojan C., and Hany H. Ammar. A Scenario-Based Reliability Analysis Approach for Component-Based Software. IEEE transactions on reliability 2004, vol.53.
5. G. Huang, H. Mei, F-Q Yang. "Runtime Recovery and Manipulation of Software Architecture of Component-based Systems." International Journal of Automated Software Engineering, Springer, 2006, 13(2): 251-278.
6. J. Li, X. Chen, G. Huang, H. Mei, and F. Chauvel. Selecting Fault Tolerant Styles for Third-Party Components with Model Checking Support. Component-based Software Engineering (CBSE) 2009.
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. A. Capozucca, N. Guelfi, P. Pelliccione, H. Muccini, "An Archi-tecture-driven Methodology for Developing Fault-Tolerant Sys-tems," Software Engineering Competence Center Technical Re-port nr. TR-SE2C-05-10, SE2C, Luxembourg, 2005

9. G. Huang, Y. Wu. Towards Architecture-Level Middleware-Enabled Exception Handling of Component-based Systems. Component-based Software Engineering 2011.
10. Garlan, D., Chung, S., Schmerl, B.: Increasing system dependability through architecture based self-repair. In: Proc. Architecting dependable systems. Springer, Heidelberg (2003)
11. H. Song, G. Huang, Franck C., W. Zhang, Y. Sun, W. Shao, H. Mei, Instant and Incremental QVT Transformation for Runtime Models(MODELS), 2011,273-288
12. ECperf webpage, http://java.sun.com/developer/earlyAccess/j2ee/ecperf/download.html
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. A. Avizienis, J-C. Lapri, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing", IEEE Transactions on Dependable and Secure Computing, Vol.1, No.1, January-March 2004, pp.11-33
16. A. Avižieni, and J. P. J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments", IEEE Computer, Vol. 17, No. 8, 1984, pp. 67-80.
17. P. E. Ammann, J. C. Knight, "Data Diversity: An Approach to Software Fault Tolerance", In Proceedings of the 17th International Symposium on Fault-Tolerant Computing Systems (FTCS-17), Pittsburgh, PA, pp. 122-126, 1987.
18. R.A. Pottinger and P.A. Bernstein, "Merging models based on given correspondences," Proc. 29th international Conference on Very Large Data Bases (VLDB '03), Sept, 2003.
19. D.S. Kolovos, R.F. Paige, and F.A.C. Polack, "Merging Models with the Epsilon Merging Language (EML)," Proc. ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems, pp. 215-229, 2006.
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. 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)
23. 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)
24. H. Song, G. Huang, F. Chauvel, Y. Xiong, Z. Hu, Y. Sun, H. Mei. "Supporting Runtime Software Architecture: A Bidirectional-Transformation-Based Approach". Journal of Systems and Software, doi:10.1016/j.jss.2010.12.009.
25. 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)
26. 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)
27. K. Goseva-Popstojanova, K. Trivedi, "Architecture-based approach to reliability assessment of software systems," Performance Evaluation, an International Journal, vol. 45, pp. 179–204, 2001.