EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Implementation and experimentation with adaptive fault management in component based software systems

Yang, F.

Award date:
2007

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

Implementation and Experimentation With adaptive fault management In component based software systems

By
Fang Yang

Supervisors:

Michel Chaudron (TU/e)
Johan Muskens (Philips Research)
Rong Su (TU/e)

Eindhoven, Aug 2007

Index

1. Introduction ... 2
  1.1 Background and problem statement ... 2
  1.2 Poker game application ... 4
2. Literature Review ... 9
3. Theoretical part ... 11
  3.1 Failure detection ... 11
  3.2 Middleman architecture & process ... 13
  3.3 Adaptive failure recovery algorithm ... 17
    3.3.1 Failure recovery model ... 17
    3.3.2 Repair rule evaluation ... 20
    3.3.3 Learning part ... 26
4. Implementation ... 31
  4.1 Runtime instantiation of the Middleman ... 32
  4.2 Failure detection ... 34
    4.2.1 Specification format ... 34
    4.2.2 Input / output parameters check ... 36
    4.2.3 Behavior deviation check ... 37
    4.2.4 Time constraints check ... 40
    4.2.5 Excessive concurrent call check ... 41
    4.2.6 Internal errors catch ... 43
  4.3 Failure recovery ... 44
5. Testing and results ... 51
  5.1 Testing approach ... 52
  5.2 Results ... 55
    5.2.1 Results of behavior deviation check ... 56
    5.2.2 Results of input / output check ... 58
    5.2.3 Results of time constraints check ... 59
    5.2.4 Results of excessive concurrent call check ... 60
    5.2.5 Results of internal errors catch ... 61
    5.2.6 Performance of adaptive repair rule selection ... 62
6. Conclusion and future work ... 65

1. Introduction

1.1 Background and problem statement

Trust4All is a joint research project of various European companies. It builds on the Space4U architecture [1] and the Robocop component model [2]. The aim for Trust4All is to define an open, component-based framework for the middleware layer in high-volume embedded appliances that enables robust and reliable operation, upgrading and extension. A Trust4All system in this paper contains a collection of software entities called components. Each component offers the functionality through a set of services, which are equivalent to a Class in the object-oriented languages. The implementation of the services is unknown to the outside, except by means of interfaces from which the users can invoke the service. An individual service sometimes requires calling other services in order to provide specific functionality. Establishing a connection between two services is called binding. Finally, an application is an executable software entity that assembles different service instances to fulfill a task. As our reliance on computers increases, so does the need for high reliability of software. In order to achieve high reliability of software application, it is important to achieve reliability of each service. Currently fault management mechanisms have been developed, which is to encapsulate existing services at runtime with fault management logic. By intercepting operation calls to and from the encapsulated service instance, the Middleman can detect deviations from the specification, and take a repair action if necessary.

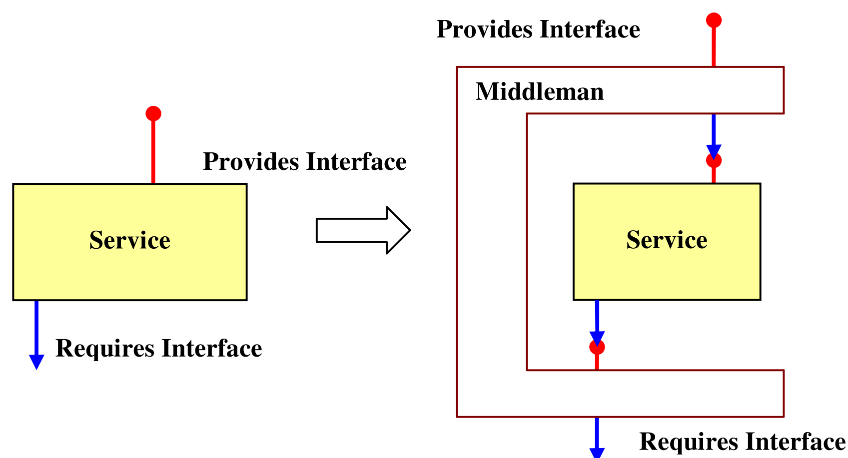

Figure 1: Middleman

In the existing solution, a number of standard failure detection and recovery units are inserted into the Middleman. But they are quite straight forward and static. In this paper, we investigate and prototype a more advanced and intelligent Middleman. We want to find the solution for the following questions:

(1) How to implement the Middleman with failure detection and recovery in an object oriented environment?
(2) How to realize the adaptive failure repair rather than statically testing a list of repair rules (the result of Space4U)?
(3) How to validate the approach? Does it improve the reliability of the component based software system?

Thus, later in this paper we will introduce an adaptive repair rule selection strategy which makes use of the knowledge from the previous experiences so that the Middleman can be adaptive to the general cases.

In the next section, we will first give a brief introduction of an application called poker game, which is constructed in a component based architecture. This application will be used in the rest chapters as an example. In Chapter 2, the literature reviews will present the existing solutions. After that, a theoretical part will elaborately discuss the internal architecture of the Middleman and the strategies for adaptive failure recovery. In Chapter 4, how to implement those strategies mentioned in Chapter 3 is introduced in an object-oriented environment. Chapter 5 contains the testing results of those strategies in the poker game application. Finally, there is a section of conclusion and future work.

1.2 Poker game application

As a vehicle for our research, we build a poker game application, which consists of four components: Game Console, Game View, Bank and Player. Each component contains only one service that has the same name as the corresponding component. The service *Game Console* mainly controls the flow of the game. It contains five interfaces: the requires interfaces **IDialog**, **IFrame**, **IBank**, **IPlayer** and the provides interface **IGameControl**. The service *Bank* serves as a virtual bank from which the players could withdraw and transfer the money. It contains two interfaces: the requires interfaces **IDialog** and the provides interface **IBank**. The service *GameView* is responsible for the view of user interaction. It contains three interfaces: the requires interface **IGameControl** and the provides interfaces **IDialog** and **IFrame**. The service *Player* implements a player instantiations. It contains two interfaces: the requires interace **IGameControl** and the provides interface **IPlayer**. Figure 2 depicts the structure of the poker game application.

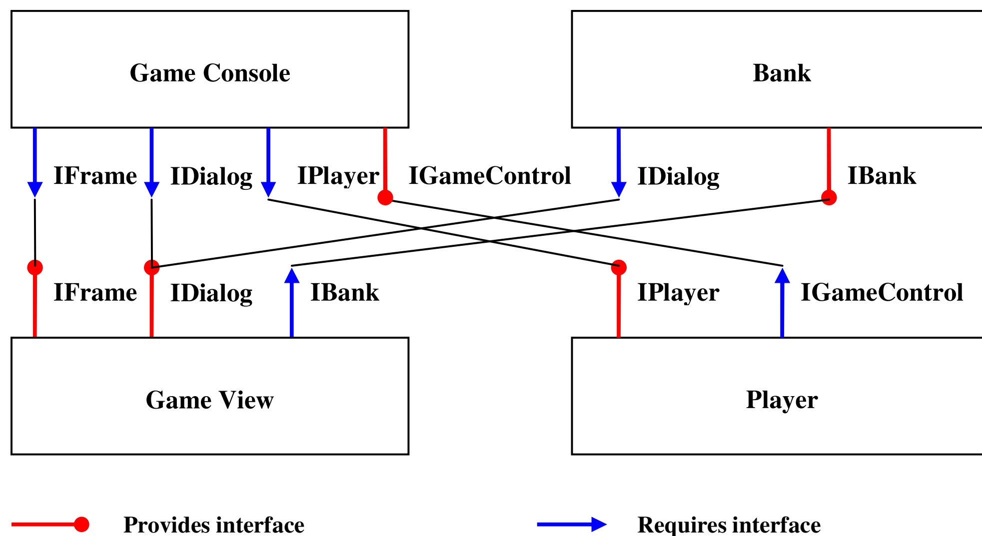

Figure 2: Structure of poker game application

In this game, there are two virtual players and one human being player. The game console controls the game sequence, and instructs the human being player with the game.

For the virtual players, their behavior is simulated by two separate threads. At the beginning, the players join the game through interface **IGameControl**. Waiting for a certain period time, the game console confirms that all the players are ready. Then each player is required to first put the stake. During this procedure, the Game Console sends a withdrawal request to Bank through **IBank**. And the bank confirms the information with the players. After all these done, the game console starts the game through **IPlayer**. In each round, the players call the interface **IGameControl** to either ask for a card or refuse a card to make their total points close to 21 points. Once the game is over, the game console calculates the results and transfers the money through **IBank**. All the game view and interactions are realized through **IFrame** and **IDialog**. Since there is a list of available operations of provides interface, the poker game application invokes the functionalities by calling an ordered sequence of operations. Figure 3 shows the sequence of the poker game with three main services.

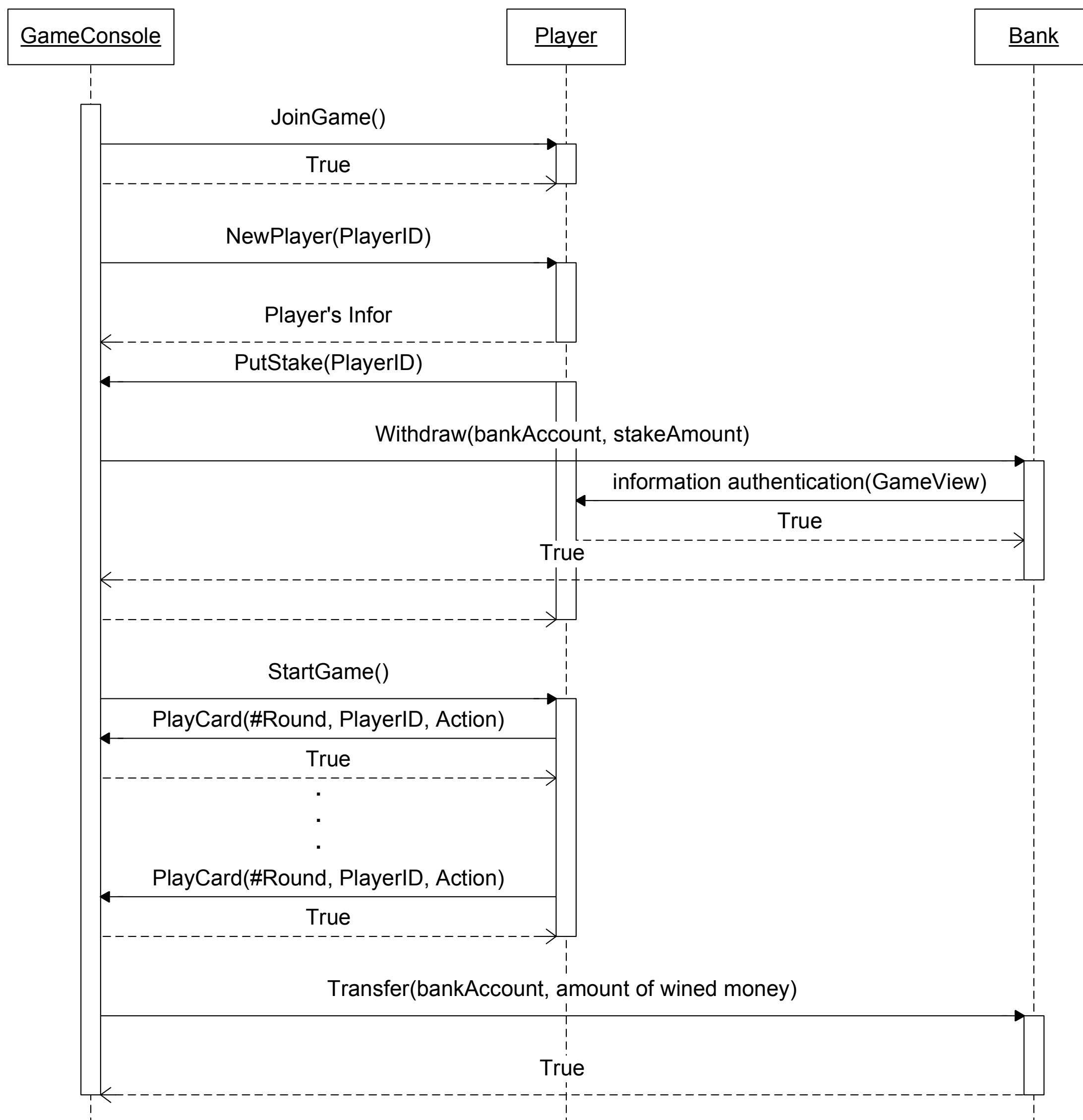

Figure 3: Sequence of poker game

Now let us see more detail for each service. In service *Bank*, provides interface **IBank** has two operations:

- **Withdraw**: called by *Game Console* to withdraw the stake from the virtual bank.

Parameter name	Format	Domain	
		Lower	Upper
Input_*BankAccount*	String of 9 digits		
Input_*Amount_of_the_stake*		100	5000
Output_*Status_of_withdrawal*	0: successful withdrawal 1: invalid bank account 2: not enough deposit to withdraw		

This operation internally calls the operations through the interface **IDialog** to make the confirmation of the withdrawal information with the players.

- **Transfer**: called by *Game Console* to transfer the money according to the game results.

Parameter name	Format	Domain	
		Lower	Upper
Input_*BankAccount*	String of 9 digits		
Input_*Amount_of_money_wined*		100	
Output_*Status_of_transfer*	True : successful transfer False : failed transfer		

In service *Game Console*, provides interface **IGameControl** has four operations:

- **JoinGame**: called by *Player* to participate into the game.

Parameter name	Description
Output_*Status*	True : succeed in joining the game False : fail to join the game

This operation assigns a player ID and internally calls the operation of **IPlayer** to instantiate the information of the newly joined player.

- **PutStake**: called by *Player* to put the stake.

Parameter name	Format	Domain	
		Lower	Upper
Input_*playerID*		0	2

This operation internally calls the operations through the interface **IBank** to withdraw the stake for each player.

- **PlayCard**: called by *Player* to request or refuse a card in each round.

Parameter name	Format	Domain	
		Lower	Upper
Input_*Round*		0	10
Input_*PlayerID*		0	2
Input_*Action*	True: get a card False: refuse a card		
Output_*Status_of_the_request*	True : succeed in dealing with the request False : fail to deal with the request		

- **Console**: called by the main application to control the whole flow of the game.

In service *Player*, provides interface **IPlayer** has three operations:

- **NewPlayer**: called by *Game Console* for instantiating the new players.

Parameter name		Format	Domain	
			Lower	Upper
Input_*playerID*			0	2
Output_*Player's_infor*	playerID		0	2
	#Round		0	10
	Bank account	String of 9 digits		
	Amount of stake		100	5000
	Action	True: get a card False: refuse a card		
	Status	0: idle / lose the game 1: ready / continue with the game 2: win the game		

- **StartGame**: called by *Game Console* to give the permission of starting the threads for the virtual players.

In service *Game View*, there are two provides interfaces **IDialog** and **IFrame**. Since the operations provided by these two interfaces are only responsible for the game interaction and view, we will not give much detail about this part.

2. Literature Review

There is a large volume of publications on failure tolerance, diagnosis and repair of software systems. Components have long promised to encapsulate data and programs into a box that operates predictably without requiring that users know the specifics of how it does so. This has brought about widespread software reuse, spawning a market for components usable with such mainstream software buses as the Common Object Request Broker Architecture (CORBA) and the Distributed Component Object Model (DCOM). But after that, people starts to ask how can we trust a component? What if the component behaves unexpectedly, either because it is faulty or simply because we misused it? Before we can trust a component in mission-critical applications, we must be able to determine, reliably and in advance, how it will behave. In 1999, A. Beugnard, J.-M. Jezequel, N. Plouzeau and D. Watkins defined a general model of software contracts and show how existing mechanisms could be used to turn traditional components into contract-aware ones.[15] They apply contracts to components such that contracts are divided into four levels of increasingly negotiable properties. The first level, basic, or syntactic contracts is required simply to make the system work. The second level, behavioral contracts improves the level of confidence in a sequential context. The third level, synchronization contracts improves confidence in distributed or concurrency contexts. The fourth level, quality-of-service contracts, quantifies quality of service and is negotiable.

Paper [1] introduces the meaning of terminology *failure* and *fault*. A *failure* is an event that occurs when the delivered service of a system deviates from correct service. Examples of failures are unexpected behavior of a system with respect to its specification like delivery of wrong data to the user. A *fault* is the adjudged or hypothesized cause of a failure. Often faults are mistakes made by a human, but they can also be caused by external factors like defective hardware. Examples of faults are incorrect thinking of the system developer that makes him produce design or programming mistakes, the toggling of memory bits due to electromagnetic disturbances in the environment where a given hardware operates, or a network congestion beyond its buffer capacities. From paper [1]

and [7], fault tolerance is the technique of designing a system that is failure free despite of faults. Fault tolerance mechanisms are based on detecting, resolving and hiding errors by wrapper pattern.

In paper [4], C. Fetzer and X. Zhen propose the idea of a wrapper, which sits between an application and its shared libraries. This approach is trying to improve the robustness of C libraries without source code access. The wrapper generation process consists of two phases. In the first phase, the system extracts the C type information for a shared library using header files and manual pages. Then it generates for each global function a fault injector (introducing the faults into the code) to determine a "robust" argument type per each argument. Based on this information, the system generates a robustness wrapper that performs careful argument checking before invoking C library functions.

In the deliverable of Space4U project [1], a Fault Management Framework used in Space4U based systems is elaborately described. It provides mechanisms for adding fault tolerance techniques (e.g. error detection and recovery) to an existing system. These techniques deal with errors occurring in Service Instances created from Components that are not trusted, without modifying these Components and without relying on reflection. The strategies applied in this Framework are similar as those in paper [3]. In paper [3], R. Su, M.R.V. Chaudron and J.J. Lukkien propose the similar solution of wrapper but to improve reliability of applications in component based system. The main idea is to construct a runtime configurable fault management mechanism (FMM) which detects deviations from the service specifications by intercepting interface calls. When a repair is necessary, FMM picks a repair action that incurs the best tradeoff between the success rate and the cost of repair. During this procedure, FMM is designed to be able to accumulate knowledge and adapts its capability accordingly.

3. Theoretical part

3.1 Failure detection

The main strategy of failure detection is to compare the real execution with the given specification which serves as a standard. Now let's discuss them in more detail.

- Input/output values check

In the specification, there is a description for input/output parameters such as the order of the parameters, data type, data domain and data format. The Middleman compares this with the real input/output to see whether there are some mismatches.

- Behavior deviation check

Sometimes, the operations of provides interfaces need the support from other services. In such cases, the operations will call requires interfaces which bind with corresponding provides interfaces in other services. This procedure is visible for the Middleman. In the specification, a behavior model is specified to describe such procedure, so the Middleman can compare the real behavior with the model to see whether a deviation occurs. Suppose there is an operation *A* provided by interface_1 in service *S*, which internally calls the operations provided by other services as in the Figure 4.

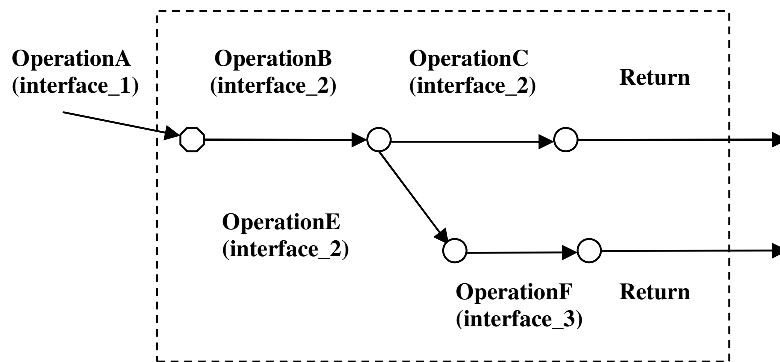

Figure 4: Example behavior model in the specification

Thus, in the implementation of requires interfaces in *MiddleMan_S*, there will be a behavior deviation check before transferring the call to other services.

- Time constraints check

In many applications, there are time constraints for the interaction response or duration of a complex computation. Thus, we introduce the concept of the timer to monitor the total execution time of the operations. Once the deadline is missed, a time out notification will be sent to the Middleman. This may indicate that the system is stuck in an infinite loop or recursion during the execution.

- Concurrent call check

In some cases, the number of synchronized calls for one operation is limited. The Middleman applies a counter to record such number. If the number exceeds the limit defined in the specification, there is an excessive concurrent call.

- Internal error check

Since the implementation of the services is a black box for the users, it is hard for the Middleman to detect the internal errors directly. Nevertheless, the Middleman can detect some of them by catching the exceptions thrown from the service that it encapsulates.

3.2 Middleman architecture & process

Since a service in a component based software system is a black box which can be accessed only through service interfaces, we use the wrapper pattern [5] to perform failure diagnosis. During the runtime execution, in addition to instantiating each service instance, a software entity called Middleman is runtime instantiated to totally encapsulate the service instance. By intercepting every interface call to or from another service instance or other users, the Middleman can detect the deviations from the specification and take a repair action if necessary. Taking the service *Bank* in the poker game case as an example, the wrapper pattern could be represented as shown in Figure 5.

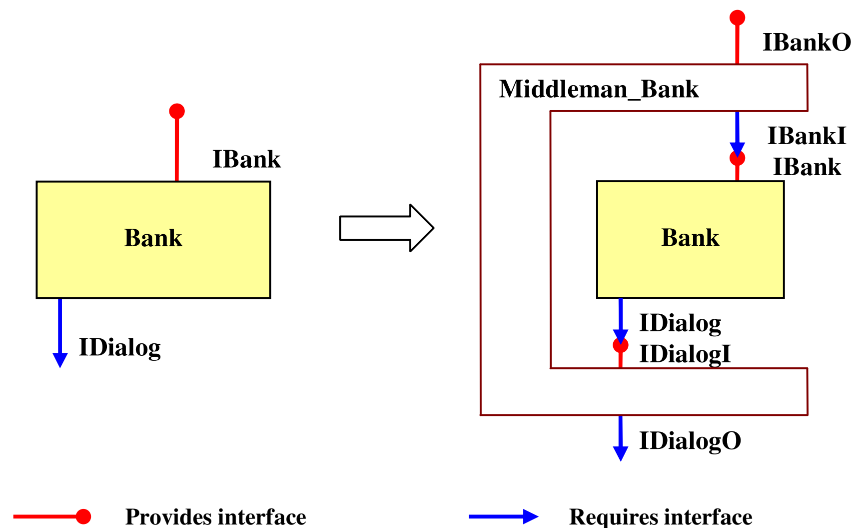

Figure 5: Middleman for service *Bank*

The Middleman implements the same interfaces as its encapsulated service instance. For each interface implemented by service instances, the Middleman implements two corresponding interfaces: one is used for internally binding with real service instance (e.g. **IBankI**, **IDialogI**); the other is used for connection with outside (e.g. **IBankO**, **IDialogO**). Calls from the external interface to the internal interface will be checked so that the failures could be detected and repaired if exist.

Inside the Middleman, a failure is detected by comparing the observation with the specification of the service instances. In general, a service instance specification allows us to identify the following failures:

1) value mismatches such as illegal input parameter values and illegal return values of an operation call;
2) unspecified operation calls within an individual provides operation such as non-existing promised service functionality and unexpected service functionality;
3) Unexpected resource usage for an individual operation. [3]

In the poker game case, the expected detectable failures are

1) Input/output mismatches;
2) Time out (an operation misses its deadline);
3) Behavior deviation;
4) Excessive concurrent calls.

For each failure, there is a unit implemented to detect it. In addition, there are another three units: *failure adaptor*, *error recovery*, and *central unit*. The unit *failure adaptor* is mainly responsible for finding the possible faults regarding to the failures; the unit *error recovery* is responsible for finding the repair rules corresponding to faults; the *central unit* internally binds with other units, and concentrates on the flow of the failure diagnosis and recovery. Figure 6 depicts the architecture of the Middleman.

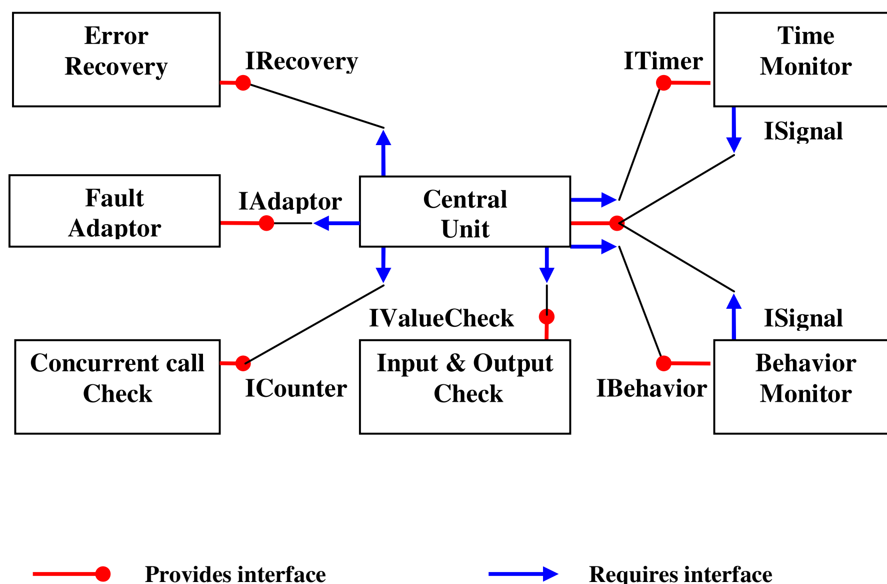

Figure 6: Architecture of Middleman

As shown in Figure 6, each unit has its own interfaces. Considering the multithread cases, the central unit will first register the information in different units such as Time Monitor and Behavior Monitor through corresponding interfaces. After that, the Central Unit will follow an ordered sequence of actions by calling other units. During this procedure, **ICounter** is first called to determine whether the call has permission to obtain immediate execution. If so, input parameters are checked through **IValueCheck**. If no failure occurs in this step, the timer will be started through **ITimer**, parallel executing with the real operation in the service instance. When there is an internal call for functionalities in other service instances, the behavior deviations are checked through **IBehavior**. Finally, outputs of the operations are checked through **IValueCheck**. If a failure occurred in the whole process, the central unit will turn to *Fault Adaptor* and *Error Recovery* for recovery. This will be discussed later.

Now, we will discuss how to detect the failures based on the above architecture. In general, there is a specification which describes:

1) A list of requires interfaces;
2) A list of provides interfaces;
3) Detailed information for each operation in provides interfaces:
    - input/output parameters
    - time constraints
    - behavior model
    - Upper bound for the number of the concurrent calls to one operation

The failure detection is mainly implemented in provides interfaces of the Middleman as shown in Figure 7.

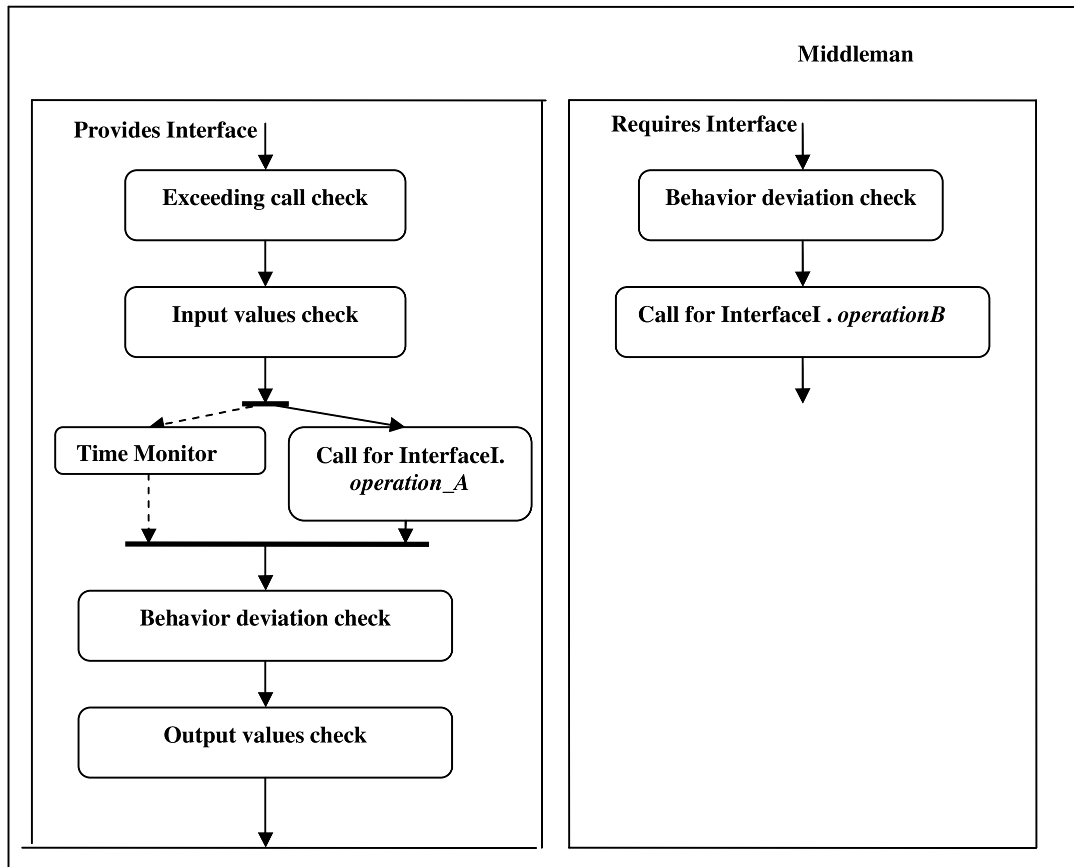

Figure 7: Process for failure detection

Once there is a call for ***Interface.operation_A*** in the service, it will be first transferred into the provides interface of the Middleman to experience first the concurrent call check and then input values check. After that, the call will be transferred into the internal interface. At the same time, the time monitor will be started to parallel running with the main sequence. When the call returns, it will experience the behavior deviation check and finally output values check. Suppose there is an internal call for ***Interface.operation_B***. It will be transferred to the requires interface, which implements the behavior deviation check and then calls the external interface.

3.3 Adaptive failure recovery algorithm

3.3.1 Failure recovery model

Once a failure is detected, removing it by some repair rule becomes our goal. A collection of faults may be *correlated* [6] with each other in the sense that:

1) The subsequent failures of a collection of faults may not be simply the union of failures caused by each individual fault;
2) Failures of one fault may be masked by failures of others;
3) The repair of one fault is indispensable of some other's repair.

To avoid unnecessary complexity, in this paper we only consider *independent faults*. Thus, we build a Failure-Fault-Repair model under this assumption. [3]

After summarizing most ideas and concepts in the existing literature [8, 14], we classify a list of commonly occurred failures and a list of design faults or faults in the environment. Let $F = \{ F_1, F_2, ...F_n \}$ denote this fault set and $f = \{ f_1, f_2, ...f_m \}$ denote the failure set. Since one failure could be caused by different faults, and one fault could result in different failures, we try to build a relation between these two sets as shown in Figure 8. These two sets and their relations will be stored in a database, and could be dynamically updated according to the real facts.

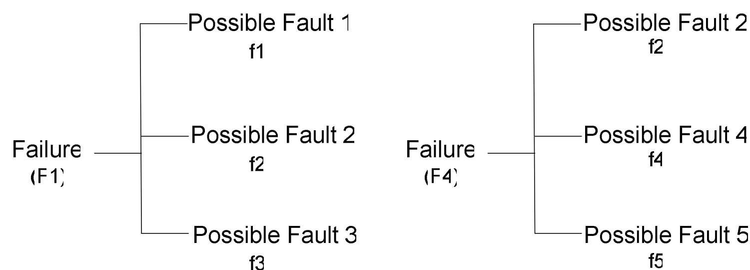

Figure 8: Example relations between failures and faults

From the figure, it is obvious to see that failure F_1 could be resulted from three different faults, while fault f_2 could be the reason for either failure F_1 or failure F_4 or both. In addition, we also summarize a list of repair rules that are possible to remove different

kinds of faults. Let $R = \{ R_1, R_2, ...R_p \}$ denote the repair rules set. Similarly we construct a relation between each fault and repair rules as shown in Figure 9.

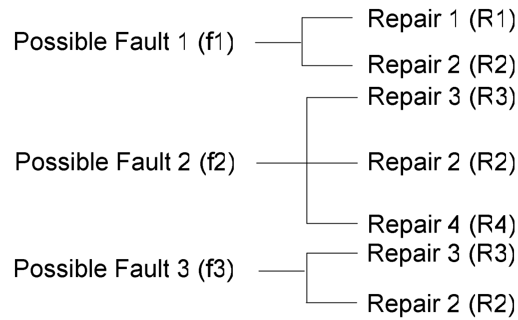

Figure 9: Example relations between faults and repair rules

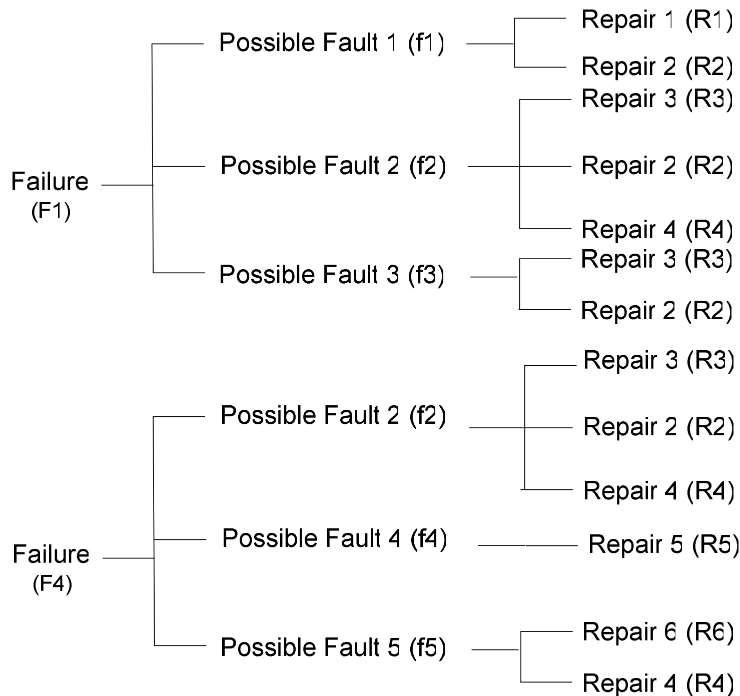

Figure 10: Example of Failure-Fault-Repair model

Hence, by combining relations in Figure 8 and Figure 9, we derive:

$$Failure - Fault \ \& \ Fault - Re\ pair \Rightarrow Failure - Re\ pair$$

Finally we form our Failure-Fault-Repair model, which could be illustrated by the example in Figure 10. Suppose we detect the failure F_1 and F_4. According to relations between failures and faults, we derive the corresponding faults set:

$$Fault(\{F_1, F_4\}) = \{f_1, f_2, f_3, f_4, f_5\}.$$

Then from relations between faults and repair rules, it is possible to obtain a list of repair rules:

$$Re\,pair(Fault(\{F_1, F_4\})) = \{R_1, R_2, R_3, R_4, R_5, R_6\}.$$

Our goal is to remove the failure efficiently. Here “efficiently” has two meanings:

1) With high success rate;
2) With low cost.

Randomly choosing a repair rule is not a wise solution. Thus, an evaluation is requested to be applied to help us make a good selection during the repair rules set.

3.3.2 Repair rule evaluation

In this section, we introduce some new concepts for the repair rules evaluation. In Figure 10, some repair rules can cover two failures. Hence, to construct a good evaluation procedure, we make full use of the properties disclosed from constructed relations. Defining a set of influential parameters is indispensable. In our evaluation function, there are three influential parameters:

- Probability of fault f_j $(1 \leq j \leq m)$ occurred when the failure F_i $(1 \leq i \leq n)$ is detected in the specific operation;
- Success rate of repair rule R_k to remove fault f_j in operation O_l;
- Cost of each repair rule;

Suppose we have already built the following two tables to represent the relations between failures, faults and repair rules.

Table 1: Example relations between failures and faults

Pair ID	Failure	Fault
0	F_1	f_1
1	F_1	f_2
2	F_1	f_3
3	F_4	f_2
4	F_4	f_4
5	F_4	f_5

Table 2: Example relations between faults and repair rules

Pair ID	Fault	Repair
0	f_1	R_1
1	f_1	R_2
2	f_2	R_3
3	f_2	R_2
4	f_2	R_4
5	f_3	R_3
6	f_3	R_2
7	f_4	R_5

8	f_5	R_6
9	f_5	R_4

Since in different operations, the probability of one fault occurrence corresponding to a failure is different, an operation label is needed for distinguishing. Assume each operation could be uniquely identified by its name and related interface name. Let $O = \{ O_1, O_2, ...O_q \}$ denote the set of all the operations of provides interfaces. And $O_l (1 \le l \le q)$ uniquely represents an operation. The probability of fault f_j $(1 \le j \le m)$ when the failure F_i $(1 \le i \le n)$ is detected in operation $O_l (1 \le l \le q)$ is denoted by $P(F_i, f_j, O_l)$. The probability could be updated during the iterative failure detection and recovery procedure. Therefore, we use a table that keep track at runtime.

Table 3: Probability

Operation	$Pair(F_i, f_j)$	**Probability**
O_1	F_1, f_1	1/3
O_1	F_1, f_2	1/3
O_1	F_1, f_3	1/3
O_1	F_4, f_2	1/3
O_1	F_4, f_4	1/3
O_1	F_4, f_5	1/3

Since failure F_1 is possible to occur because of three different faults in $Fault(\{F_1\})$, initially we set these probabilities equally as 1/3 ($\frac{1}{|Fault(\{F_1\})|}$). For failure F_4, we do the setting similarly. The value will be updated according to the repair facts. This will be discussed later in this chapter.

Because it is possible for us to record the attempt times and successful times for each repair rule when attempting to remove some failure, we define success rate as the result of successful times divided by the attempt times. Since this parameter can be uniquely identified by the operation $O_l (1 \le l \le q)$, fault f_j and repair rule R_k, we denote it as $SR_fault(f_j, R_k, O_l)$. The parameters are stored in the table as the following:

Table 4: Success rate

Operation	$Pair(f_j , R_k)$	Successful times	Test times
O_1	f_1 , R_1	1	2
O_1	f_1 , R_2	1	2
O_1	f_2 , R_3	1	2
O_2	f_2 , R_2	1	2
O_2	f_2 , R_4	1	2
O_2	f_3 , R_3	1	2
O_3	f_3 , R_2	1	2
O_3	f_4 , R_5	1	2
O_3	f_5 , R_6	1	2
O_3	f_5 , R_4	1	2

We initially set the successful times to be 1 and the attempt times to be 2 so that the success rate is 50%. There is another solution for the success rate's initial setting which will be discussed later in the learning part.

The last parameter "cost" is currently set to be a constant for each repair rule, denoted by $Cost(R_k)$. Now we start to define our evaluation procedure based on the above three influential parameters.

Step1:

Compute the success rate for removing the failure F_i in operation O_l by repair rule R_k:

- $SR_failure\ (R_k , F_i , O_l) \ = \sum_{f \in coverage(F_i)} P(F_i , f , O_l) * SR_fault(f , R_k , O_l)$

  - $coverage(F_i)$: A set of the possible faults corresponding to the failure F_i. Illustrated by an example, $coverage(F_1) = \{ f_1 , f_2 , f_3 \}$, while $coverage(F_4) = \{ f_2 , f_4 , f_5 \}$.
  - $SR_fault(f_j , R_k , O_l)$: The success rate of the repair rule R_k applied to repair fault f_j in operation O_l. Since sometimes the repair rule R_k can not cover all the

faults in the $coverage(F_i)$, let the success rate in this condition to be zero. For example, $SR_fault(f_4, R_4, O_l)=0$ due to the fact that the repair rule R_4 is not a possible solution for repair the fault f_4.

Based on the case given in the Figure 10, initially set "success rate" to be 50%. We can do the following calculation:

$$SR_failure(R_1, F_1, O_1) = P(F_1, f_1, O_1) * SR_fault(f_1, R_1, O_1) = \frac{1}{3} * 50\% = 16.7\%$$

$$\begin{aligned} SR_failure(R_2, F_1, O_1) &= P(F_1, f_1, O_1) * SR_fault(f_1, R_2, O_1) \\ &+ P(F_1, f_2, O_1) * SR_fault(f_2, R_2, O_1) \\ &+ P(F_1, f_3, O_1) * SR_fault(f_3, R_2, O_1) \\ &= \frac{1}{3} * 50\% + \frac{1}{3} * 50\% + \frac{1}{3} * 50\% = 50\% \end{aligned}$$

$$\begin{aligned} SR_failure(R_3, F_1, O_1) &= P(F_1, f_2, O_1) * SR_fault(f_2, R_3, O_1) \\ &+ P(F_1, f_3, O_1) * SR_fault(f_3, R_3, O_1) \\ &= \frac{1}{3} * 50\% + \frac{1}{3} * 50\% = 33.3\% \end{aligned}$$

$$SR_failure(R_4, F_1, O_1) = P(F_1, f_2, O_1) * SR_fault(f_2, R_4, O_1) = \frac{1}{3} * 50\% = 16.7\%$$

Sometimes, multiple failures occur. Suppose it is a subset of failure set F, let us denote it as F'. In this condition, we extend the above evaluation function into the following way:

- $SR_failure(R_k, F', O_l) = \frac{1}{|F'|} \sum_{F_i \in F'} SR_failure(R_k, F_i, O_l)$

  - $|F'|$ denote the number of elements in the set F'.

Taking the example that $F' = \{F_1, F_4\}$, we do the following calculation:

$$\begin{aligned} SR_failure(R_3, F', O_1) &= \frac{1}{2}(P(F_1, f_2, O_1) * SR_fault(f_2, R_3, O_1) \\ &+ P(F_1, f_3, O_1) * SR_fault(f_3, R_3, O_1) \\ &+ P(F_4, f_2), O_1) * SR_fault(f_2, R_3, O_1)) \\ &= \frac{\frac{1}{3} * 50\% + \frac{1}{3} * 50\% + \frac{1}{3} * 50\%}{2} = 25\% \end{aligned}$$

$$SR_failure(R_5, F', O_1) = \frac{1}{2}P(F_4, f_4, O_1) * SR_fault(f_4, R_5, O_1)$$

$$= \frac{\frac{1}{3} * 50\%}{2} = 8.33\%$$

Step2:

Compute the effectiveness which is defined as the success rate per cost:

$$E(R_k, F', O_l) = SR_failure(R_k, F', O_l) / cost(R_k).$$

Then we try to rank the repair rules by the results:

$$E(R_k, F', O_l) > E(R_{k+1}, F', O_l) \Leftrightarrow rank(R_k) > rank(R_{k+1})$$

Here set F' should include at least one element failure. If there are two or more repair rules getting the same rank, we will randomly choose one as a trial. Since each repair rule R_k has a constant cost, we do the following assumption:

$$Cost(R_1) = 20, Cost(R_2) = 50, Cost(R_3) = 70, Cost(R_4) = 30$$

Based on the result we did in step 1, we try to rank the repair rules $\{R_1, R_2, R_3, R_4\}$ when only failure F_1 occurs in the operation O_1.

$$E(R_1, \{F_1\}, O_1) = SR_failure(R_1, \{F_1\}, O_1) / cost(R_1) = \frac{16.7\%}{20} = 0.835\%$$

$$E(R_2, \{F_1\}, O_1) = SR_failure(R_2, \{F_1\}, O_1) / cost(R_2) = \frac{50\%}{50} = 1\%$$

$$E(R_3, \{F_1\}, O_1) = SR_failure(R_3, \{F_1\}, O_1) / cost(R_3) = \frac{33.3\%}{70} = 0.476\%$$

$$E(R_4, \{F_1\}, O_1) = SR_failure(R_4, \{F_1\}, O_1) / cost(R_4) = \frac{16.7\%}{30} = 0.567\%$$

Thus, we can derive the rank for each repair rule:

$$rank(R_2) > rank(R_1) > rank(R_4) > rank(R_3)$$

According to the above results, we will first try the repair rule R_2. If it succeeds in removing the failure F_1 occurred in the operation O_1, we won't try the other repair rules any more. Otherwise, we will try the next repair rule until it succeeds or all the repair

rules have been tried. If all the rules fail to remove the failure, a message will be sent to the Fault Manager for manual solution.

3.3.3 Learning part

Although from the failure we can not tell which fault is the origin in certainty, we can learn some related information from the iterative repair procedure. Besides, the success rate is dynamically changing after each repair action. Hence, we bring in the idea of "learning", which means applying some rules to updating the influential parameters in our evaluation procedure. This could be divided into two parts:

1) Updating the success rate for repair rule R_k to remove the fault f_j;
2) Updating for probability of fault f_j occurred when the failure F_i is detected in the specific operation.

To update the success rate, we will increase both the successful times and attempt times by one if the corresponding repair rule succeeds in removing the failure. Otherwise, we only increase the attempt times by one and retain the successful times. Then we will turn back to its initialization. Since we already mentioned one method in the previous section, here we will give another solution and also their individual advantages and disadvantages.

- Method 1: the initial success rate $SR_fault(f_j, R_k, O_l)$ is set to be 50% represented by 1 success times: 2 attempt times. (mentioned in section 3.2.2)

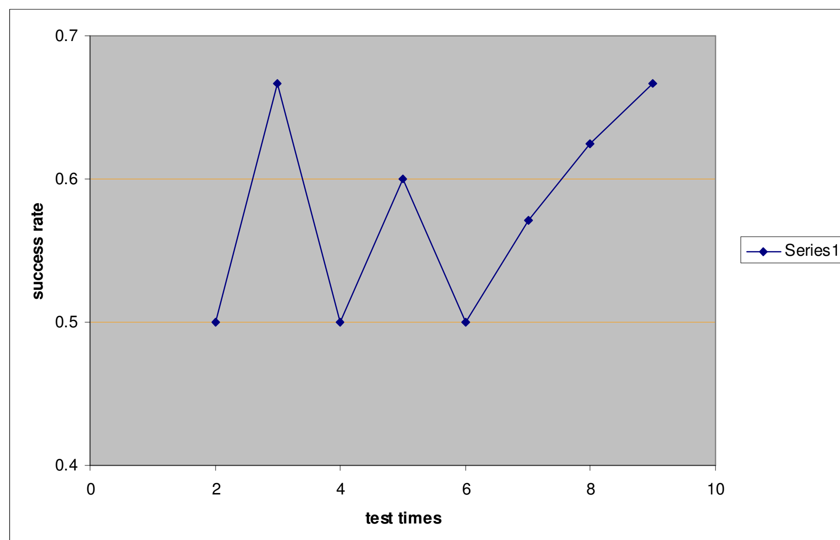

Figure 11: Example tendency of success rate by Method 1

Draw a graph to show the trend of the success rates with the attempt times as an example. It is obvious to see that this procedure is quite sensitive to the modification, especially for small number of attempt times. Therefore, at the beginning phase, there may be some fluctuation that can not accurately show the real trend of the success rates. In addition, if the success rate decreases to a very low value at the beginning, it may loss the trial chance from then on even it is an effective repair rule.

- Method 2: the initial success rate for each repair rules is also set to be 50%. But before n (a large integer) test times, the success rate will remain 50%. Suppose n = 200, we derive the following diagram:

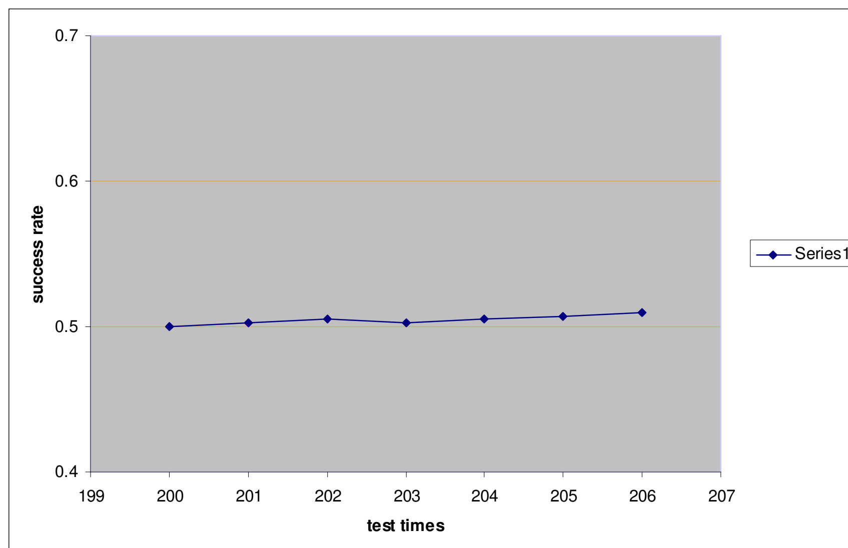

Figure 12: Example tendency of success rate by Method 2

It could be observed from the diagram that this method is not very sensitive to the modification. But it will slowly approach the real trend without obvious fluctuations.

It is difficult to tell which method is better. But if it is possible to quickly obtain a large number of samples, Method 2 is more preferable. No obvious fluctuations could present the tendency of success rate more accurately without making some repair rules lose trial chances. On the situation that only a small number of samples are derived, it is better to choose Method 1 which could show the tendency of success rate. Though there are some fluctuations, it is still better than no changes shown by Method 2.

Since the above two methods update the success rate based on the result last time, one specific repair rule may be always chosen after a large number of attempt times. This will cause other repair rules to lose the attempt chances. To improve this condition, we introduce the third method.

Method 3: windowed success rate. Supposing we already had a large number of sampled success rate which denoted by $\{t_1, t_2, ...t_n\}$. When calculating t_{n+1}, we first try to get the average of samples $\underbrace{\{t_i, t_{i+1}, ...t_j\}}_{size\ \ k}$ ($\subseteq \{t_1, t_2, ...t_n\}$) which likes a window of size k. Then based on this result, we update the success rate. To calculate t_{n+2}, the window slides with a sample to obtain the average from samples $\underbrace{\{t_{i+1}, t_{i+2}, ...t_{j+1}\}}_{size\ \ k}$.

Now we will move to the update for probability of fault f_j $(1 \le j \le m)$ when failure F_i $(1 \le i \le n)$ is detected in the specific operation. If repair rule $R_k (1 \le k \le p)$ succeeds, let $cov(R_k)$ denote the fault set that R_k could cover. Before update, $P(F_i, f_j, O_l) = \dfrac{weight(F_i, f_j)}{\sum_{f \in coverage(F_i)} weight(F_i, f)}$. At the beginning, $weight(F_i, f_j)$ is set to be 1, which denotes fault f_j takes the weight one. $\sum_{f \in coverage(F_i)} weight(F_i, f)$ denotes the total weight of faults when failure F_i is detected. Then we update

$$if \quad f_j \in cov(R_k), \quad weight(F_i, f_j) = weight(F_i, f_j) + 1;$$
$$if \quad f_j \notin cov(R_k), \quad retain \ \ weight(F_i, f_j);$$

With the newly updated weight for each fault, we recalculate the probability according to the definition.

Supposing a failure F_l occurs in the operation O_l, let us take the example in Figure 13 for illustration.

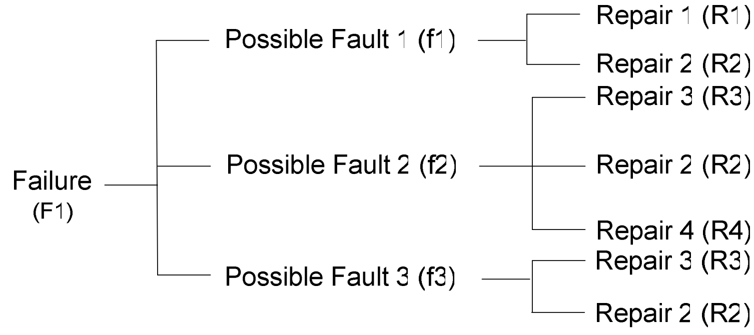

Figure 13: Example of relations between failures, faults and repair rules

Initially,

$$P(F_1, f_1, O_1) = \frac{1}{3},\ P(F_1, f_2, O_1) = \frac{1}{3},\ P((F_1, f_3, O_1) = \frac{1}{3}.$$

Then we do the following update:

- if R_1 is successful: $cov(R_1) \cap coverage(F_1) = \{f_1\}$

$$P(F_1, f_1, O_1) = \frac{1+1}{3+1} = \frac{1}{2}$$

$$P(F_1, f_2, O_1) = \frac{1}{3+1} = \frac{1}{4}$$

$$P(F_1, f_3, O_1) = \frac{1}{3+1} = \frac{1}{4}$$

- if R_2 is successful: $cov(R_2) \cap coverage(F_1) = \{f_1, f_2, f_3\}$

$$P(F_1, f_1, O_1) = \frac{1+1}{3+3} = \frac{1}{3}$$

$$P(F_1, f_2, O_1) = \frac{1+1}{3+3} = \frac{1}{3}$$

$$P(F_1, f_3, O_1) = \frac{1+1}{3+3} = \frac{1}{3}$$

- if R_3 is successful: $cov(R_3) \cap coverage(F_1) = \{f_2, f_3\}$

$$P(F_1, f_1, O_1) = \frac{1}{3+2} = \frac{1}{5}$$

$$P(F_1, f_2, O_1) = \frac{1+1}{3+2} = \frac{2}{5}$$

$$P(F_1, f_3, O_1) = \frac{1+1}{3+2} = \frac{2}{5}$$

- if R_4 is successful: $cov(R_4) \cap coverage(F_1) = \{ f_2 \}$

$$P(F_1, f_1, O_1) = \frac{1}{3+1} = \frac{1}{4}$$

$$P(F_1, f_2, O_1) = \frac{1}{3+1} = \frac{1}{4}$$

$$P(F_1, f_3, O_1) = \frac{1+1}{3+1} = \frac{1}{2}$$

Hence, our adaptive failure repair is an iterative procedure combining the repair rule evaluation and its learning update.

4. Implementation

In this chapter, we mainly focus on implementing the aforementioned concepts into a real application in Java environment. To briefly explain more, we focus on the poker game application. There are three main parts:

- Runtime instantiation of Middleman
- Failure detection
- Failure repair

4.1 Runtime instantiation of the Middleman

In Chapter 3, we already mentioned that the Middleman is a software entity runtime instantiated to encapsulate the service instance. To implement this in applications, it no longer directly creates a service instance, but instantiates a Middleman which will internally create the service instance. Since the existence of Middleman is unknown to the users, a table is created to internally bind the call for service instances with its Middleman. Therefore, when the users call for a service by its name, the system will internally search for its related Middleman from the table and rebind the call to the Middleman.

The Middleman itself implements the same provides and requires interfaces as those in the service that it encapsulates. But inside each operation of provides interfaces there are some logical part doing prior and post checks. If we describe this in a diagram, it could be like Figure 14.

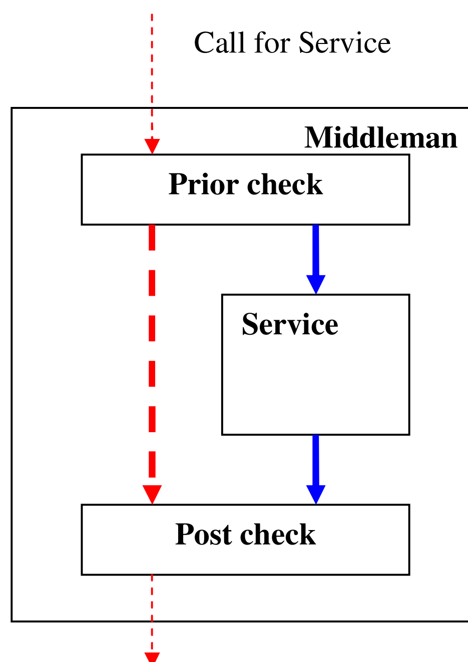

Figure 14: Middleman

There is a call for Service executed by an observer thread in dashed line. It first experiences a set of prior logic checks. Then a new thread called observed thread in bold

line is created to internally call the operation in the real service instance through provides interfaces. The observer thread begins to sleep when the observed thread starts, which is denoted by the bold dashed line. When the operation finishes in the observed thread, the observer thread derives the results and does a set of post logic checks. Here prior logic checks include input parameters check, concurrent call check and behavior deviation check and post check is mainly output parameters check.

4.2 Failure detection

In this section, we introduce the implementation for failure detection. Since the specification plays an important role as a comparison standard, we will first discuss the specification format.

4.2.1 Specification format

In each operation's specification, there is a description containing its functionality, input/output parameters and normal behavior. It is requested to derive and transfer the useful information to a format that the Middleman could access and use. Here is a list of information for each operation derived from the specification:

- Operation name
- Interface name
- Time constraints
- Upper bound for the number of the current calls to one operation
- Behavior model
- Input parameters' constraints
- Output parameters' constraints

"Operation name" and "Interface name" are used to uniquely identify one specific operation. In some case, an operation is required to complete within a constrained period of time, so we bring in "Time constraints". Take the bank as an instance, the clients won't be patient to wait for more than five minutes to withdraw 20 Euro. Hence, it is needed to monitor the response time. If it misses the deadline, some actions should be taken.

In the poker game, there are three players whose behavior is controlled by three separate threads. Suppose one player is calling for the operation "withdraw", which is already called by another and not completed yet. And it is not allowable for two players doing this action together. Then the "Upper bound of the number of the current calls to one operation" is applied to temporarily block the call from the latecomer. A behavior model is similar to a state machine. Let us consider the operation "JoinGame" of interface

IGameControl. This operation receives the participation request from *Player*, and internally checks whether the player is allowed. If so, a player ID will be generated. Then "JoinGame" calls "newPlayer" through interface **IPlayer** to instantiate detail information with new player ID. In the implementation, this behavior is represented by a sequence in Figure 15.

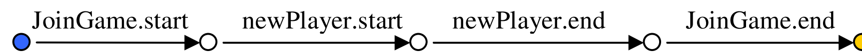

Figure 15: Behavior model of operation "JoinGame"

If there are branches in the behavior, it could be represented as:

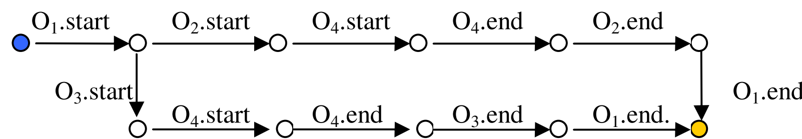

Figure 16: Behavior Model of an example operation

How to use this model to check behavior deviation will be discussed later. As to input/output parameters' constraints, a set of information are defined in the specification:

- Object type
- Data name
- Data type
- Format (only for "String" data)
- Domain (the lower bound and upper bound only for the numerical data)
- Default value

Since sometimes input or output parameters could be objects, "Object type" is applied to distinguish those from simply primitive types such as integer. Due to the properties of java, "Format" is defined by java regular expression [13]. For example, "\d" denotes a digit between 0-9, "X{n}" denotes character X exactly n times. Thus, a bank account consisting of 9 digits, can be defined as "\\d{9}" as its format. In the poker game application, the amount of stake is set 500 as the default based on which the players could increase. Then we record "Default value" for operations if it exists, which could provide a repair suggestion when a input failure is detected.

4.2.2 Input / output parameters check

This functionality is provided by service "Input & Output Check". There is one operation "dataCheck" of provides interface **IValueCheck**.

Input parameters:

- Real input (output)
- Specified input (output)

Since different operations have different number and types of input (output) parameters, "Vector" is used to dynamically record them. The output of this operation is an array, which records the checking results corresponding to each input element.

If there is an object O existing in the input (output) parameters, we first use the following code to derive the value of the primitive data in it.

```
Line 1.    Class c = O.getClass();
Line 2.    Field[] m = c.getDeclaredFields();
Line 3.    Field f = m[index];
Line 4.    field.setAccessible(true);
Line 5.    field.get(o);
```

Line 1 Returns the runtime class of the object O. Line 2 obtains an array of Field objects reflecting all the fields declared by the class c. And the last three lines finally get the value of the field represented by *f*, on the specified object O.

Once the value of primitive type is derived, it will experience two different checks accordingly. They are "format check" and "domain check". "Format check" is only applied for the data in String, while "domain check" is only for the numerical data. Suppose we have a string called "bankAccount". Since the data "Format" is defined by Java regular expression as "\\d{9}", directly calling a method of class String *bankAccount.matches("\\d{9}")* can tell whether "bankAccount" is in the correct format. As to the "domain check", it is similar to the mathematical comparison between the lower bound and upper bound according to the different data types.

Checking input / output value mismatches is just before and after the execution of the operation in the observer thread so that such failure could be caught in time.

4.2.3 Behavior deviation check

This functionality is provided by service “Behavior Monitor”. Behavior check is implemented not only in the provides interfaces but also in the requires interfaces. Considering the multiple threads in the poker game application, it is the common cases that three threads execute the same operation at the same time. Taking the example in Figure 16, three threads are currently in the execution of that operation, but at the different steps. Suppose thread *A* is already at the end state, while thread *B* is in the state before *O2.end*, and thread *C* is just at the initial state. Hence we have to distinguish them in such situation. A concept of behavior information is brought in, which record the call’s information such as thread (which thread it belongs to), current step (which step the call is currently at) and so on.

In the provides interface **IBehavior**, there are four operations:

- *Init*: responsible for initializing the basic information for each operation call.
- *stepBehaviorCheck*: responsible for behavior deviation check by comparing the real behavior with the specified behavior.
- *updateBehaviorInfo*: responsible for updating the behavior information (this is not indicated in Figure 17, since it is called during the failure recovery).
- *removeBehaviorInfo*: responsible for removing the behavior information when the operation call is completed.

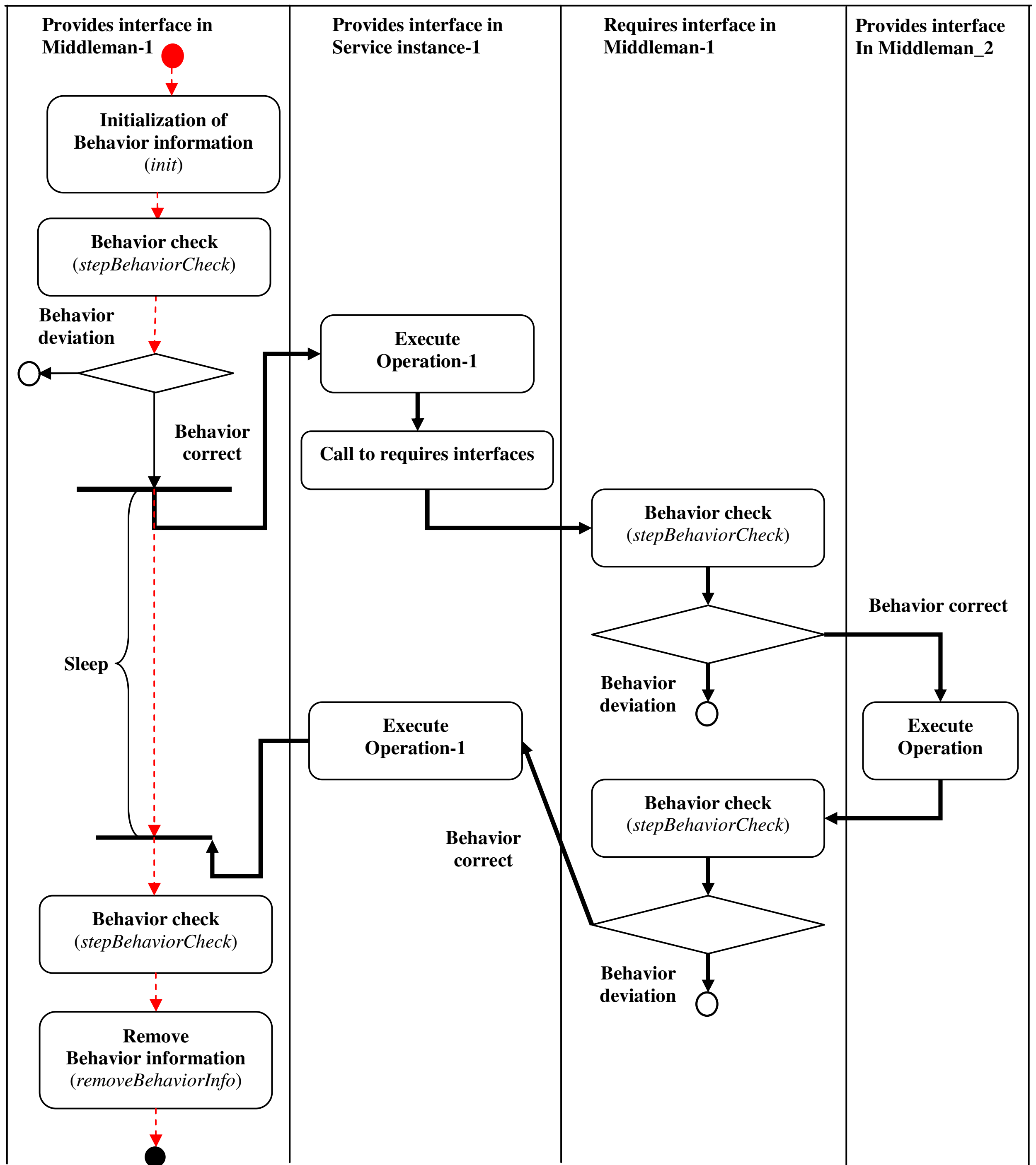

Figure 17: Behavior deviation check, activity diagram

Figure 17 describes how to implement the behavior deviation check. The dashed arrows form the observer thread in the Middleman-1, while the bold arrows form the observed thread for the Operation-1 execution. When the Middleman-1 receives a call for the

service_instance-1, it first initializes the behavior information. Then take the behavior check. If a deviation exists, it will call for failure recovery denoted by the hollow circle. Otherwise, a new thread is created to execute operation-1. Assume there is a call to some functionality in other service instances during the procedure. Then behavior checks will be taken in the requires interface. Once the internal call is completed, the operation-1 in the provides interface will continue. The main thread is sleeping to wait for the return from operation-1. When operation-1 returns, a behavior check is invoked to see whether the operation is correctly completed without any loses of internal behavior etc. Finally, the behavior information is removed to save the space.

4.2.4 Time constraints check

This functionality is provided by service "Time Monitor", implemented by the following codes:

```
Line 1.    Timer timer = new Timer();
Line 2.    timer.schedule(new TimerTask() {
Line 3.        public void run() {
                   //alarm to the observer thread
Line 4.        }
Line 5.    } , deadline);
```

Line 1 instantiates a new timer which will run in the background thread parallel with the observed thread. From Line 2 to 5, a timer task is scheduled in this timer, which will give an alarm to the observer thread when the deadline is missed.

Figure 18 describes how it is implemented. At the beginning, the observer thread in dashed lines, instantiates the timer. Then, the operation is started in the observed thread, at the same time the timer is started in the background thread. Once the deadline is missed, a notification is sent to the observer thread. Then the observer thread will turn to the failure recovery part in hollow circle. During this procedure, the operation in the observed thread keeps executing until there is some repair rule applied.

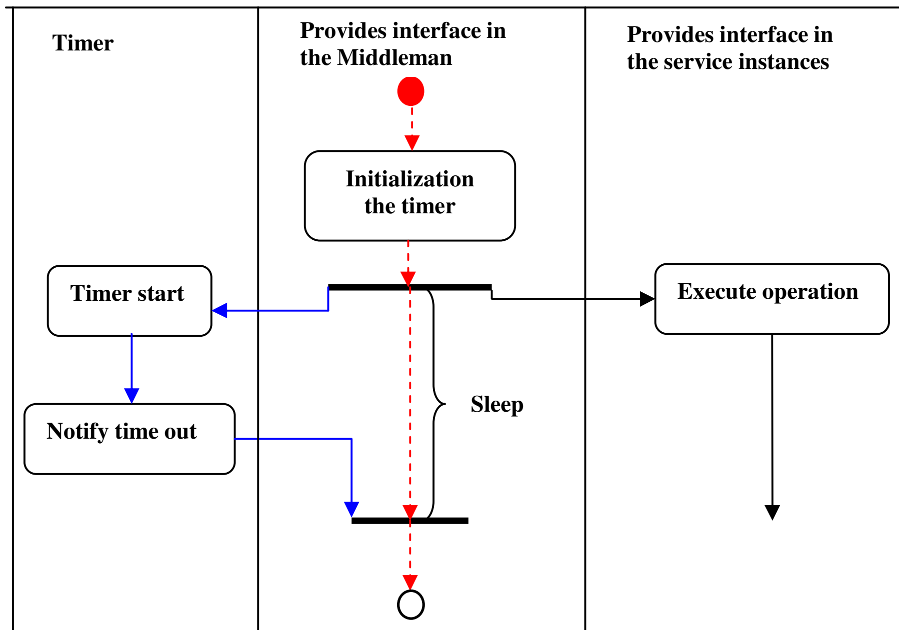

Figure 18: Time constraints check, activity diagram

4.2.5 Excessive concurrent call check

This functionality is provided by service "Concurrent call check". It is implemented only in the provides interface in the Middleman. In some applications, this could prevent data inconsistency. For example, there is an operation that could access and modify the internal data. Two users *A* and *B* want to call this operation. By setting the upper bound of the counter to 1, it is guaranteed that only one user can execute the operation untill the end. In other cases, the exceeding call check could ensure the application is running within its processing capability. For instance, the game console can only deal with three players at a certain period, thus this check could temporarily block more requests from the other players.

In the provides interface "**ICounter**", there are two operations:

- *decreaseCounter*: responsible for two tasks:
  1) Initialize the counter with the specification when the operation is called first time;
  2) Check the counter whether it is still larger than zero. If so, decrease the counter by 1 to give the permission to invoke the operation.
- *increaseCounter*: responsible for releasing the permission by increase the counter by 1.

Figure 19 describes the implementation of the excessive concurrent call check. The dashed arrows form the observer thread in the Middleman. When a call arrives, the counter will check whether the permission could be given. If so, a new thread in bold arrows will be created to execute the operation in the service instance. Otherwise, failure recovery will be called from the hollow circle. The observer thread is sleeping from the operation invoked in the service instance untill its completion. Finally, the permission is released to the counter.

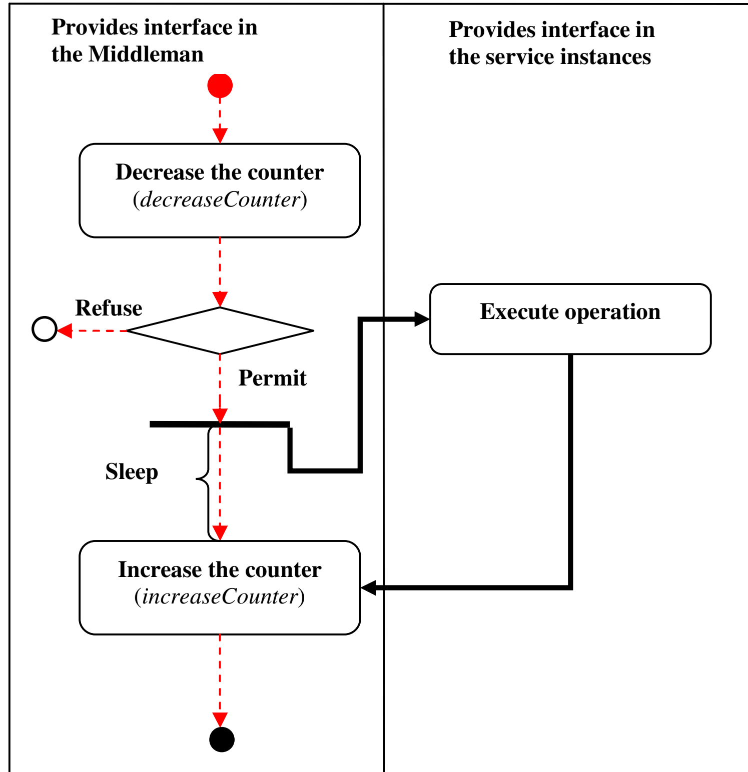

Figure 19: Excessive concurrent call check, activity diagram

4.2.6 Internal errors catch

Since a service is a black box, it is hard for the Middleman to directly monitor the errors inside the service. In Java, it is possible to throw the internal exception to the outside. Thus, in the implementation of provides interfaces in the Middleman, it catches the exceptions thrown from the interfaces by the following code:

Line 1. *try{*
Line 2. *interface.operation A*
Line 3. *}*
Line 4. *catch(Exception e){*
Line 5. *Exception manager in central unit of Middleman*
Line 6. *}*

In the exception manager, it will show the exception message on the Java console, and try to remove the exception by "retry" or "replace".

4.3 Failure recovery

In this section, we focus on the implementation of failure recovery part. There are two parts in this section. First, we will briefly introduce how to select a repair rule once a failure is detected. After that, we will discuss some repair rules in detail.

- **Adaptive failure recovery**

In the section 3.2, we have already illustrated the strategy of adaptive repair rule. Due to the Failure-Fault-Repair model, once there is a failure detected, the Middleman will try to remove the failure as described in Figure 20.

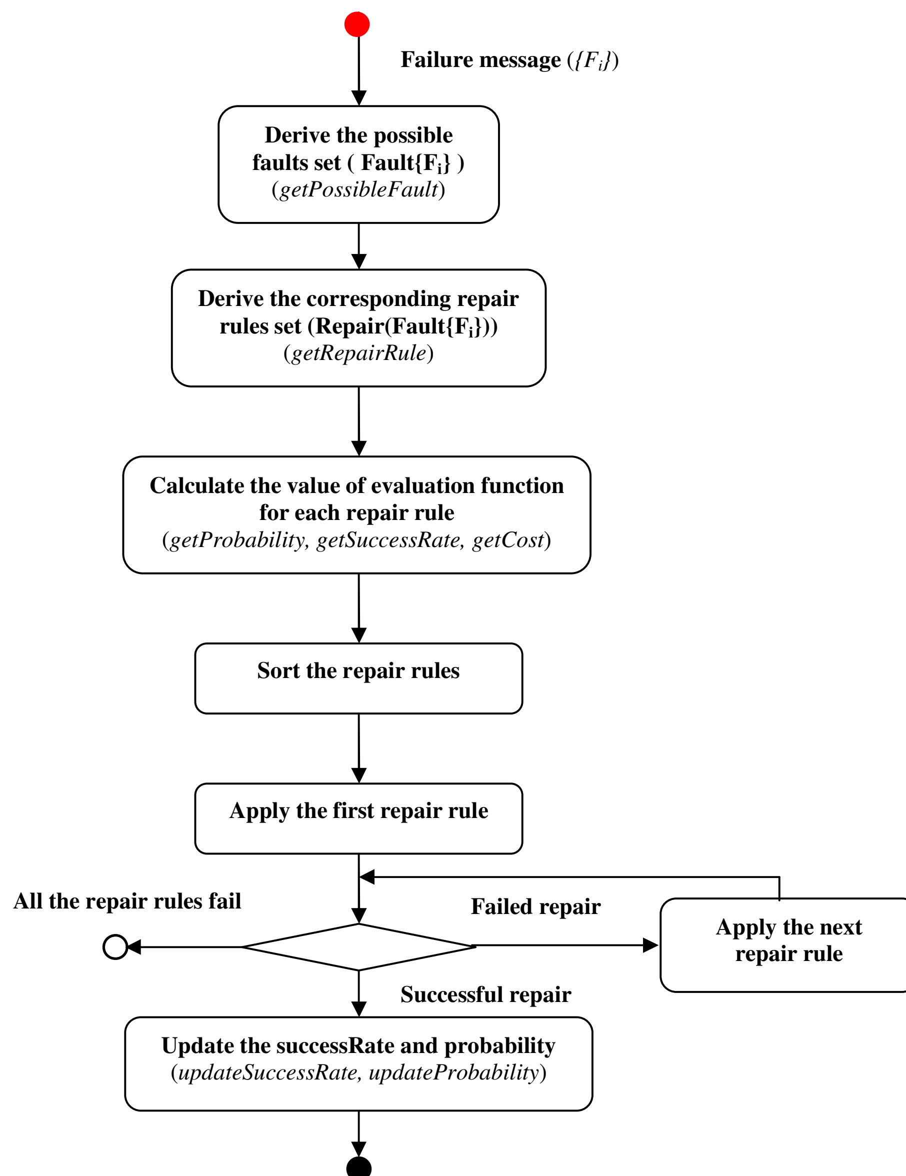

Figure 20: Adaptive failure recovery, activity diagram

At the initial state, the failure message is transferred into the central unit of the Middleman. First, operation *getPossibleFault* in the provides interface **IAdaptor** is called to derive the possible faults corresponding to the failure message. According to this result, we obtain a list of repair rules by the operation *getRepairRule* through the provides interface **IRecovery**. Then we do the calculation to evaluate each repair rule by calling operation *getProbability* of **IAdaptor**, *getSuccessRate* and *getCost* of **IRecovery**. After sorting the repair rules, we test each rule in order until the failure is removed or all the rules are failed. If the failure is removed, the success rate and probability are updated by operations *updateSuccessRate* of **IRecovery** and *updateProbability* of **IAdaptor**. Otherwise, Fault Manager in hollow circle is referred to as a manual solution.

- **Some repair rules**

Before giving the details of the implementation about failure recovery, we list a set of faults injected in the poker game application as well as their corresponding failures and repair rules.

Fault	Failure	Repair	Constraints
Infinite loop	Time out	Create a new service instance to replace the previous one	No side effect on the system states
		Retry	Temporary error
Unintended calls	Behavior deviation	Create a new service instance to replace the previous one	No side effects on the system states
		Retry	Temporary error
Invalid input	- Input wrong format - Input out of domain	Get input from elsewhere	Elsewhere is available
		Apply the default value	The default value is sensible and available
Invalid output	- Output wrong format - Output out of domain	Retry	Temporary error
		Get output from elsewhere	Elsewhere is available
		Apply the default value	The default value is sensible and available
		Create a new service instance to replace the previous one	No side effect on the system states
Untimely response	Time out	Wait for more time	
		Retry	
		Create a new service instance to replace the previous one	No side effect on the system states
Excessive concurrent call	Call counter overflow	Temporary block the call and wait for the entry	
Internal error	Exception	Retry	
		Create a new service instance to replace the previous one	

- **Apply the default value (AD) & get from elsewhere (GE)**

These two rules are mainly applied when some input (output) failures occur. A default value is provided by the specification if it exists. Elsewhere could be another service instance which provides the similar functionality, or manually input (output) from the keyboard.

Line 1. *input = new byte[n];*

Line 2. *System.in.read(input);*

Line 3. *System.in.skip(System.in.available());*

Line 4. *String s = new String(input);*

Suppose, the input parameter is a string of size n. Line 2 derives the input stream from the keyboard, while Line 3 gets rid of the redundancy from the input stream. Finally, Line 4 creates the string obtained from the input.

Once the rule AD is applied, the program will sequentially set the default values to the valid parameters. If the parameter is an Object, it will be first decomposed into elements as the way mentioned in section 4.2.2. As to the rule GE, it is almost the same as the rule AD, only that it requires keyboard input. Take the poker game application as an example. Suppose the bank account is invalid, then GE can be applied to get the value from the users. If the amount of stake is out of domain, we apply AD to set the value as 500 Euro.

- **Temporary block the call and wait for the entry**

This rule is mainly applied when the excessive concurrent call of one operation is detected. Described in the Figure 21, the observer thread in dashed arrows is sleeping until it obtains the permission. Then the observed thread in bold arrows is created and started to execute the operation in the service instance.

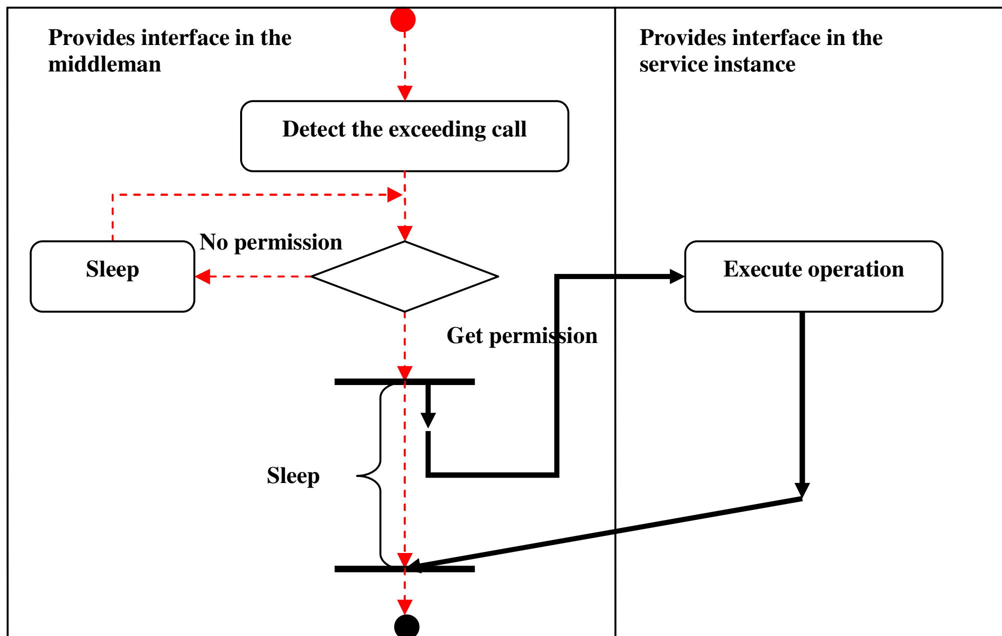

Figure 21: Temporary block the excessive call and wait for the entry, activity diagram

- **Retry**

This rule is mostly applied in the case of temporary errors. Since the observer thread is separate from the observed thread which executes the operation in the service instance, “Retry” is implemented by stopping the observed thread and starting a new again.

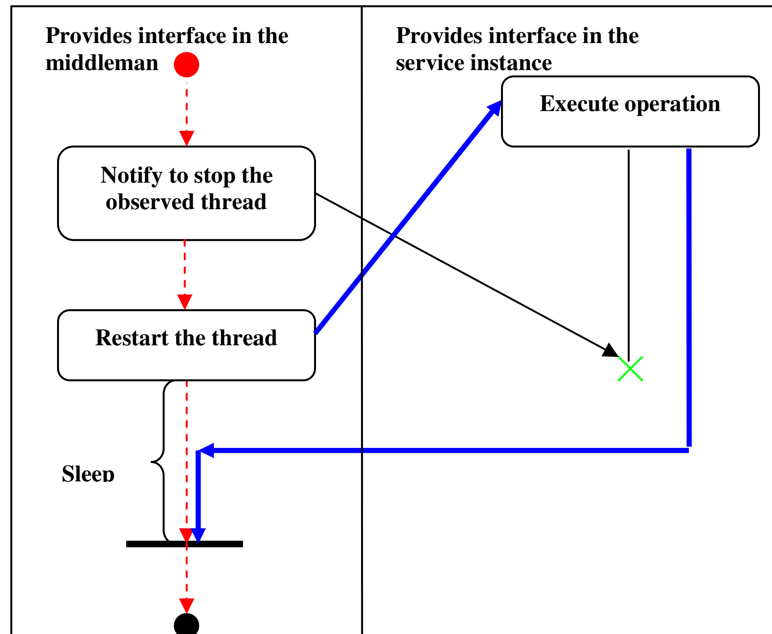

Figure 22: Retry, activity diagram

From Figure 22, it is obvious to see that the observer thread dashed arrows stops the observed thread at the cross, and restart the thread to execute the operation in the service instance by bold arrows.

- **Create a new service instance to replace the previous one**

This rule is mostly applied as the last option when the other repair rules fail. It is quite similar to the repair rule "Retry" by first stopping the observed thread, then creating a new service instance to replace the old one, and finally restarting the observed thread. In this procedure, the interfaces between the Middleman and the service instance will be rebound.

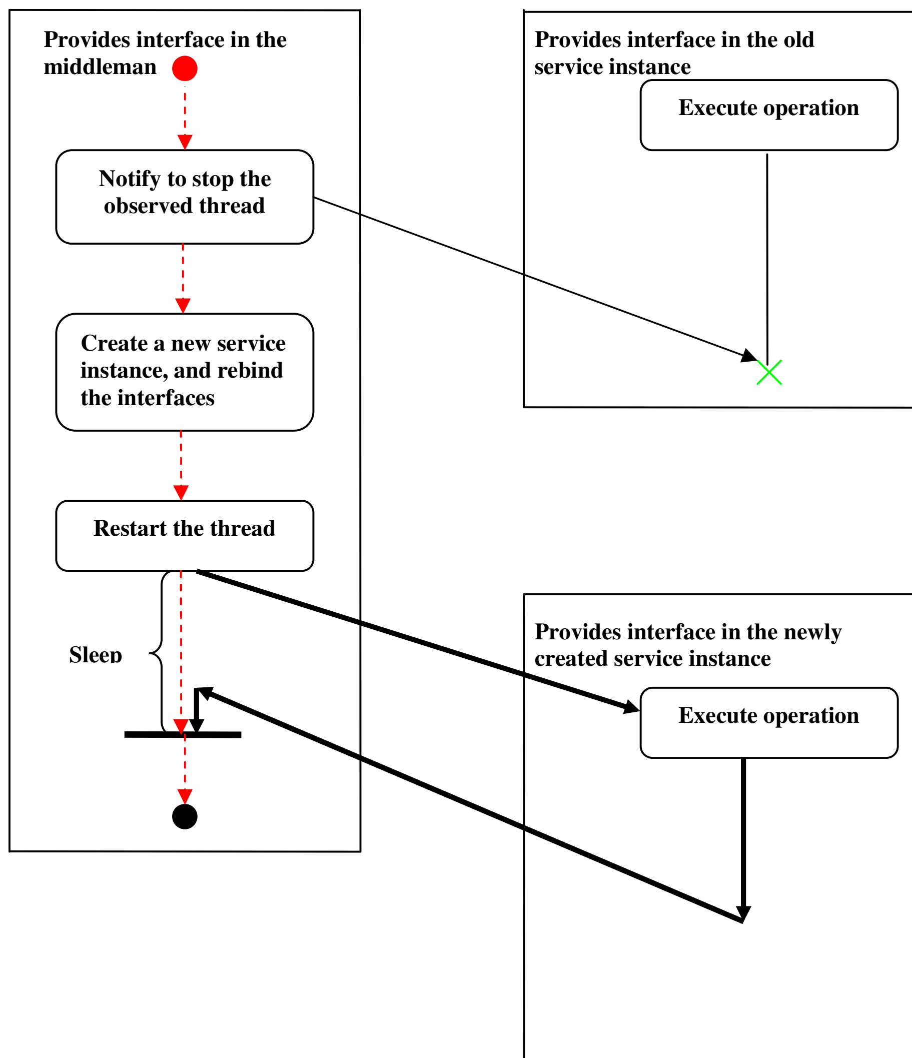

Figure 23: Create a new service instance to replace the previous one, activity diagram

5. Testing and results

Since the failure detection and recovery mechanisms are already implemented into the poker game application, we did some tests to validate the effectiveness of those mechanisms. Thus, in this chapter, we mainly focus on the testing approaches and the results.

5.1 Testing approach

Before starting the tests, we first generate the test cases by injecting different kinds of faults. The introduced faults may propagate to the observable failures which may be caught by our mechanisms. In this way, we can test how those mechanisms work to answer the following questions:

1. Are failures detected by the Middleman?
2. If the answer of question 1 is yes, are failures removed by the Middleman?
3. If the answer of question 1 is no, why can the failure not be detected? How to improve the mechanisms if possible?
4. If the answer of question 2 is no, why are failures not removed? How to improve if possible?

There are two approaches applied for fault injection in our tests:

- Code mutation tool
- Manually fault injection

Here, we introduce a code mutation tool called ***µJava*** (***muJava***). It is a mutation system for Java programs developed by two universities, Korea Advanced Institute of Science and Technology (KAIST) in S. Korea and George Mason University in the USA [8, 9, 10]. It automatically generates mutants for both traditional mutation testing and class-level mutation testing. ***µJava*** can test individual classes and packages of multiple classes. In our tests, this mutation tool generates lots of test cases which can cause the failures covering time out, unintended behavior, input / output mismatches. For the failures out of our scope such as internal logical error, we just filter them out. Since ***µJava*** is designed highly based on the characteristics of Java, there are some cases not covered by this tool. Hence, we manually introduce some faults into the code in order to complement the limitation of ***µJava***.

The manually injected faults are designed according to different categories. For the behavior check part, some operations have internal calls to other services, while others do not. Thus, we divide test cases into two main categories:

- The operations do not have internal calls (**B1**):

- Insert unintended internal calls (*B1.1*)

- The operations have internal calls (**B2**):
  - Insert unintended internal calls (*B2.1*)
  - Disorder the sequence of internal calls (*B2.2*)
  - Delete some internal calls (*B2.3*)

If we illustrate the second category in an example behavior model as Figure 24, the original behavior internally calls operation *A* then *B*. For the case *B2.1*, we insert the internal call to operation *C* into the real application. In case *B2.2*, we change the sequence such that the internal call to operation *A* has to wait for the completeness of operation *B*. In the last case, we delete the internal call to the operation *A*.

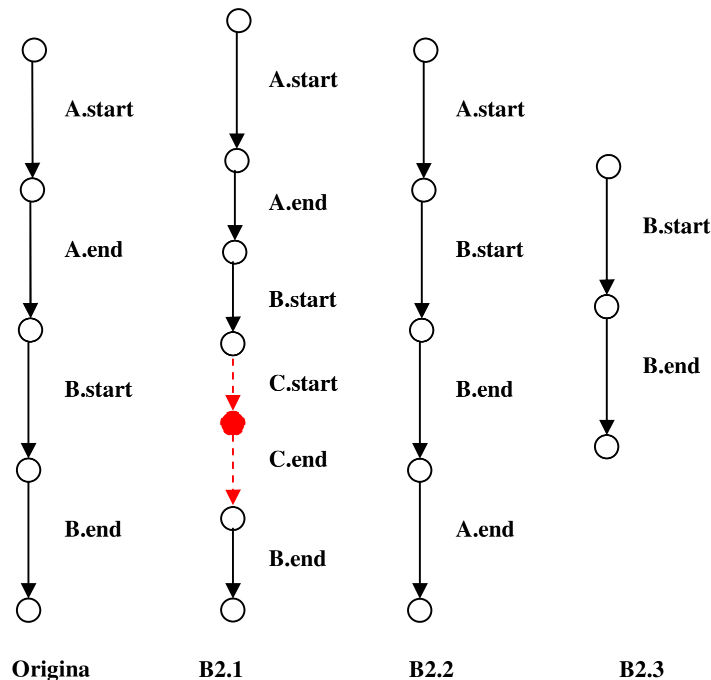

Figure 24: An example for behavior test cases

Now let's move to the test cases for the input and output mismatches. Since there are parameters in objects or primitive type in the input / output, we also divide the test cases into two categories:

- Parameters in type of Object (**I1**)
  - Object is null pointer (*IO1.1*)
  - The elements of the object are out of domain (*IO1.2*)

- The elements of the objects are of incorrect format (*IO1.3*)

- Parameters in primitive type (**I2**)
  - Out of domain (*IO2.1*)
  - Incorrect format (*IO2.2*)

We mainly focus on checking whether the parameters are out of domain or of wrong format. For the parameters in Object type, whether it is of null pointer is also in the checking scope.

As to the failures like time out and excessive concurrent calls, we did not do the manual injection, since code mutation tool has produced enough cases. All the injected failures we mentioned before are tested on two situations: persistent injection or transient injection. Here transient injection is implemented by using a parameter *random*:

```
Line 1.    random = (Math.random()*10)%2;
Line 2.    if(random==1)
Line 3.      //code with injected faults
Line 4.    else
Line 5.      //code without injected faults
```

Initially, randomly generate a number in the domain 0-10. If the number is odd, the code with injected faults is executed. Otherwise, the correct code is executed.

Finally, we did the test to see how the strategy of adaptive repair rule selection performs. This mainly depends on iteratively running the poker game application to monitor the modification of evaluation results.

5.2 Results

In this section, we will present the testing results. For easy illustration, we will discuss them according to different categories of failures in the following sequence:

1) Results of behavior deviation check;
2) Results of input / output check;
3) Results of time constraints check;
4) Results of exceeding call check;
5) Results of internal errors catch;
6) Performance of adaptive repair rule selection.

In each category, the detection results will be showed together with the recovery results. Before starting the discussion of the results, we will give some notices about the poker game application for later easy illustration. In the testing procedure, we mainly focus on the three services: *Player*, *Game Console* and *Bank*. The services *Player* and *Game Console* have their internal states, which mean they are not stateless during the game. For more detail information, please refer to section 1.2.

5.2.1 Results of behavior deviation check

Success rate of detection = Number of successful detection / number of test cases
Success rate of repair = Number of successful repair / Number of successful detection

Type of injected fault	Detailed fault category	Number of test cases	Number of successful detection	Number of successful repair	Success rate of detection	Success rate of repair
Behavior deviation	*B1.1*: Unintended internal call in the operation of set B1	20	20	15	100%	75%
	B2.1: unintended internal call in the operation of set B2	20	20	10	100%	50%
	B2.2: Disordered sequence of internal call in the operation of set B2	20	20	10	100%	50%
	B2.3: missed internal call in the operation of set B2	20	20	10	100%	50%
	Misbehavior in choosing branches of the behavior model	20	0	0	0	0
Total		100	80	45	80%	56.25%

All the test cases in the categories *B1.1*, *B2.1*, *B2.2* and *B2.3* are detected by the Middleman. If the faults are injected temporarily, they could be removed by the repair rule "retry". But if the faults are injected permanently, then the condition becomes complicated. For the service *Bank* which has no internal state, the repair rule "Create a new instance and replace the previous one (Replace)" is successful to remove the failure. But for the service *Player* and *Game Console*, the repair rule "Replace" fails to remove the failure in behavior, which means all the repair rules fail. For further improvements we maybe can try to record the internal states in the service during the execution, so that the states could be restored before "Replace". Since in component based software systems, the detail of the service is unknown to the outside, it maybe difficult for us to obtain the complete information about the system internal states. So, it may be a better solution to constraint the system design into a way of stateless. Once the service is designed stateless, "Replace" can remove the failures in behavior check easily.

The behavior model sometimes contains branches as in Figure 25. Suppose the left dashed path is path *A* and the right path is path *B*. In the specification, this model is represented as an array which contains path *A* and *B*, which means the behavior is allowed to choose either path *A* or *B*.

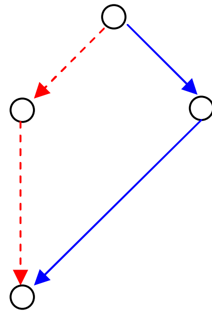

Figure 25: behavior model with branches

But in the real application, there are conditions to decide which path to go. For example, if the condition is true, go to path A; if the condition is false, go to path B. If we change the code such that when the condition is false, go to path A, then our current behavior model fails to detect this kind of failures. This is because we did not set the conditions for branches selection in the model. Maybe in the future, we could add this property into the behavior model to make the behavior check stricter.

5.2.2 Results of input / output check

Success rate of detection = Number of successful detection / number of test cases
Success rate of repair = Number of successful repair / Number of successful detection

Type of injected fault	Detailed fault category	Number of test cases	Number of successful detection	Number of successful repair	Success rate of detection	Success rate of repair
Invalid input/output	*IO1.1*: Null pointer for parameters in Object type	20	20	8	100%	40%
	IO1.2: Out of domain for elements in parameters in Object type	20	20	17	100%	85%
	IO1.3: Incorrect format for elements in parameters in Object type	20	20	20	100%	100%
	IO2.1: Out of domain for parameters in primitive type	20	20	14	100%	70%
	IO2.2: Incorrect format for parameters in primitive type	0	0	0	0	0
Total		80	80	59	100%	73.75%

All the test cases in the categories *IO1.1*, *IO1.2*, *IO1.3* and *IO2.1* are detected by the Middleman. There is no test case designed in category *IO2.2* because of the limitation of the poker game program. For removing the failures in the input parameters, "setting the default value" or "getting from elsewhere" is applied. In the poker game application, "getting from elsewhere" requests the user to give the input parameters from the keyboard. In other applications, elsewhere could also be implemented as a service which provides the required input parameters. But if the input parameter is null pointer, the current repair rules can not deal with this. In addition, if the invalid input parameter is quite related to the system internal states, such as the number of rounds in the poker game, it is infeasible to recover by the above two rules.

For the failures in the output parameters, there are another two repair rules: retry and replace. If the faults are injected temporarily, they can be removed by "retry". Otherwise, "replace" is applied. In the stateless service, this rule is successful to remove the failures. But in a state related service, this rule will fail similarly to the behavior check part.

5.2.3 Results of time constraints check

Success rate of detection = Number of successful detection / number of test cases
Success rate of repair = Number of successful repair / Number of successful detection

Type of injected fault	Number of test cases	Number of successful detection	Number of successful repair	Success rate of detection	Success rate of repair
Infinite loop	20	6	6	30%	100%
Untimely response	20	20	10	100%	50%
Others	10	10	5	100%	50%
Total	50	36	21	72%	58.33%

Time monitor is mainly used to check for untimely response or infinite loop in the poker game case. In the specification, we give an assumption of the time constraints for most operations except the function "*console*". Since the function "*console*" controls the whole sequence of the game, it is very hard to set a time constraint for it. Hence, the infinite loop occurred in this function is unable to be detected. But for the other cases, when the interaction is untimely or the infinite loop occurs, the Middleman can detect them by receiving the time out notification. For the temporary infinite loop, it could be removed by "retry"; for the permanent ones, it could be removed by "replace" for the stateless service like *Bank*, but not for the state related services.

In addition, time out sometimes occurs not because of untimely response or infinite loop. The timer in the Middleman monitors not only the execution time of the operations, but also the time taken by failure detection and recovery during the execution. Take the poker game application as an example, suppose there is one operation *A* which is quite simple and only takes up very little execution time. But the failure detection and recovery take large amounts of time, which finally leads to time out of the operation *A* itself, or even of the other operation which waits for the results from the operation *A*. Even worse, the timer monitor may disturb the failure repair in the operation A due to the time expiration. Thus it is better to achieve a balance between real applications and failure detection and recovery in order to not influence the real execution of the applications. In the poker game application, for operations without output, we just skip the output checking. For operations related to the game view, we skip most of the failure detections on them since there are not much data communications in them.

5.2.4 Results of excessive concurrent call check

Success rate of detection = Number of successful detection / number of test cases
Success rate of repair = Number of successful repair / Number of successful detection

Type of injected fault	Number of test cases	Number of successful detection	Number of successful repair	Success rate of detection	Success rate of repair
Excessive concurrent call	50	50	39	100%	78%

During the tests, we did not manually inject the test cases particularly to excessive concurrent call check. But the test cases generated by ***µJava*** sometimes propagate to observable excessive concurrent calls. Suppose there is an operation *A* which could be called by only one user at one time. If the user *a* is executing operation *A*, the later comers *b* and *c*'s calls to operation *A* are detected by the Middleman as excessive concurrent calls. They will get the entry in order after the user *a* completes the operation *A*. But if some failure occurs when the user *a* is executing operation *A* and the Middleman fails to remove the failure, then a deadlock may occur. This is because in the current model, only after user *a* successfully completes the operation *A*, then will it release the entry for the other users. This means if user *a* fails to execute the operation *A*, the others will always wait for the entry in queue.

5.2.5 Results of internal errors catch

Success rate of detection = Number of successful detection / number of test cases
Success rate of repair = Number of successful repair / Number of successful detection

Type of injected fault	Detailed fault category	Number of test cases	Number of successful detection	Number of successful repair	Success rate of detection	Success rate of repair
Internal errors	Array index out of bounds	30	20	15	66.67%	75%
	Not instantiated	30	20	6	66.67%	30%
	Others (internal logic errors, internal calculation errors)	40	0	0	0	0
Total		100	40	21	40%	21%

Although the Middleman can not know the detail inside the service, it can catch the internal exception thrown from the service. If the exceptions are temporary such as array's index out of bounds or object null pointer, they can be caught by the Middleman and removed by "retry". If the exceptions are permanent, they could be removed by "replace" in stateless services. But for the complicated exceptions, the Middleman can only catch them rather than remove them due to the property of component based software system. For those internal logic errors or calculation errors which do not throw exceptions or propagate to the visible failures in the interfaces, the current Middleman is not able to detect them and repair them.

For the service *Player*, there is another point that should be mentioned here. Since there are two separate threads in *Player* which run as two virtual players, they are invoked by the *Game Console*. After that, they run individually, the Middleman can not monitor their behavior as well as catch the exceptions in the current models. Thus, to improve this situation, it is better to avoid the creation of multiple concurrent threads in instances of one component when designing the system. Or we may consider applying the fault management to a single thread rather than to a service.

5.2.6 Performance of adaptive repair rule selection

Since it is hard to accumulate large amounts of field data in the poker game application, we initialize the success rate as 50% (1 successful times: 2 attempt times) and did the further tests based on this. After the testing, we observe that the mechanism for adaptive repair rule selection performs quite well. Figure 26 shows how the evaluation for two repair rules "apply the default value (AD)" and "get from elsewhere (GE)" goes, when the input parameter bank account is invalid. From the figure, at the beginning, the rule "AD" is tested but fails due to the results decrease. For the rule "GE", although it does not rank in the first order at the beginning, after 3 attempts, it goes ahead of the rule "AD". Therefore, the Middleman will not test rule "AD" any more so that its evaluation remains at 0.0125.

Figure 27 shows the trend of evaluation of repair rule, when the exceeding call is detected. Currently, in the poker game application, only "Wait for entry" is suggested as a repair rule. It could be observed that the increasing trend of the evaluation goes quite fast. If in the future, there are more corresponding repair rules, the better one can also stand out quite quickly after several tests.

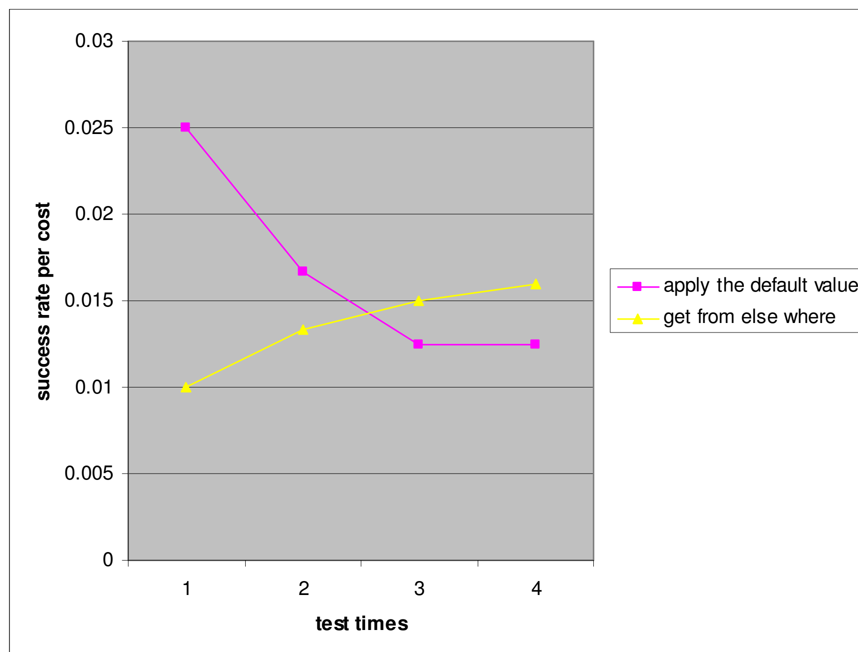

Figure 26: Invalid bank account for the operation "Withdraw"

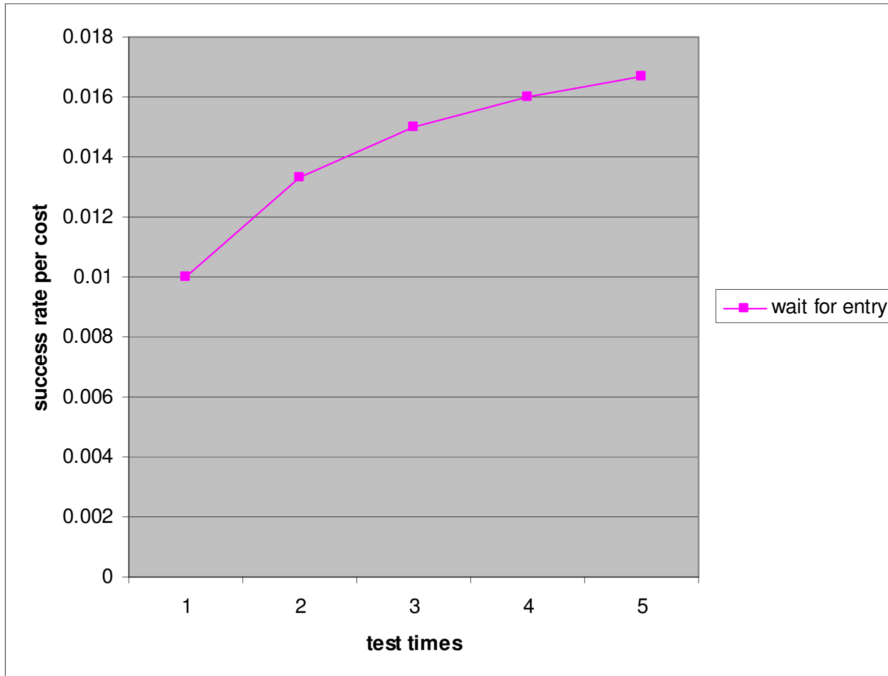

Figure 27: Excessive concurrent call

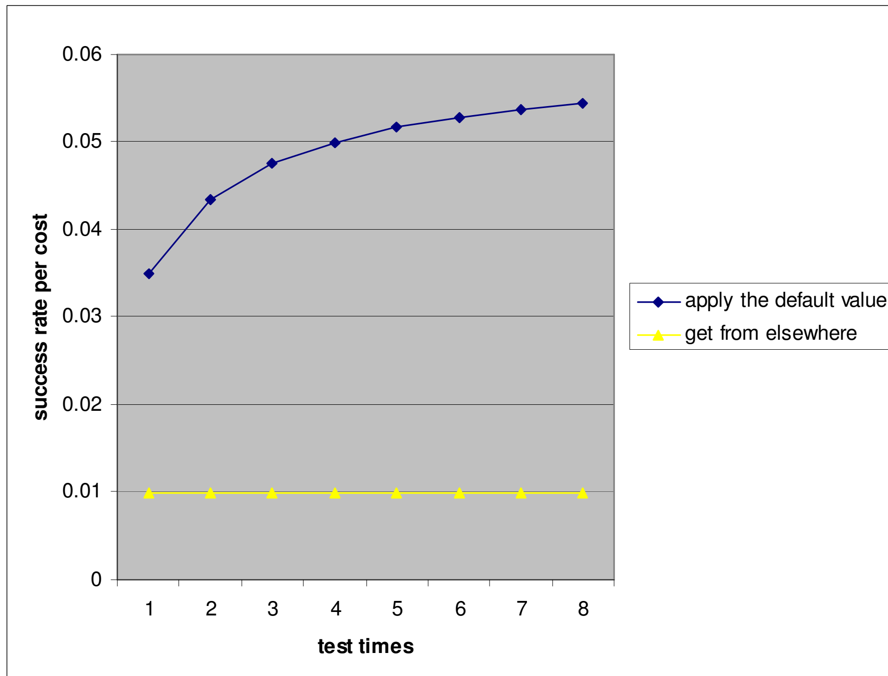

Figure 28: Stake out of bounds

The last figure presents the trend when the randomly generated amounts of stakes are out of bounds. Since we provide the default values in the specification, so the Middleman will always try the repair rule "apply the default value". Hence, its evaluation results are

continuously increasing, while the results of “get from elsewhere” remains due to no chance to test.

Concerning the limitation of poker game application, it may be better to test this adaptive selection mechanism in other applications in the near future. Besides, we can provide some system design guideline so that the system is more compatible with the Middleman in the current models.

6. Conclusion and future work

During this project, I first read the related literatures to learn about the Robocop component model and fault management mechanisms. Then I designed and implemented a test application which built on the component model. Based this test case, I implemented the adaptive fault management with extendible repair rules. And finally I did a validation on the approach.

From the description in the last section, we can see that most of the failures in the test cases can be detected by the Middleman. But for those unintended calls in the branched behavior model, our Middleman totally fails to deal with. As to the failure recovery, the Middleman's performance seems not so satisfactory, since it quite depends on the properties of the services, such as stateless. If the fault is transient, it could be removed by "retry" at most times. But if the service instances have the internal states, sometimes "retry" action will bring in some side effects. When the fault is persistent, "replace" could remove it within the stateless service instances. But for those with internal states, "replace" performs not good, since it loses the internal states and even the internal data. To improve this, it should be considered to make the services stateless during the design phase. In addition, we can consider defining Fault_Management_Interface in the component to communicate with the Middleman, so that the Middleman can export the states of the services. This will improve the Middleman on failure recovery.

For the adaptive repair rule selection, the results show that it could learn the experience to dynamically choose the repair rules. Though, in some specific application, it may be possible to get the information about which repair rule is better in advance. But for the general case, our strategy is more suitable so that we can decide which repair rule is better from the experience data.

Good performance of the Middleman depends a lot on the quality and architecture of the services. If the service is implemented decently rather than full of bugs, it will be much easier and highly successful to remove the failures. Currently, the Middleman applies the failure detection and recovery for each service. But in case of multiple threads, the

Middleman fails to monitor the behaviors and catch the internal errors. Hence, in the future we should consider about applying the fault management on each thread. Since the timer is applied to monitor the duration for an operation's completion, the time for failure detection and recovery is also included. Thus, it is important for us to set proper time constraints so as to avoid the performance overhead. In addition, the testing results already show that there is some relationship between different failures. The current repair rules are hard to deal with this condition, because sometimes one repair procedure will be influenced by the other operations. It is the future work to investigate the relationship between different failures, and how to improve the current failure recovery way into more effective and accuracy.

Reference:

1. CSEM, Ikerlan, Nokia, UPM, Philips and TU/e, "Space4U Deliverable 2.4 Specification of Framework", Jun. 2005
2. Robocop: Robust Open Component Based Software Architecture, URL: http://www.hitechprojects.com/euprojects/robocop/deliverables.htm
3. R. Su, M.R.V. Chaudron and J.J. Lukkien, "Adaptive runtime fault management for service instances in component-based software applications", *IET Software*, Vol. 1 (1), pp. 18-28, Feb. 2007
4. C. Fetzer and X. Zhen, "An Automated Approach to Increasing the Robustness of C Libraries", *Proceedings of the International Conference on Dependable Systems and Networks*, 2002
5. E. Gamma, R. Helm, R. Johnson and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", *Addison Wesley Professional, 1st edition*, 1994
6. K. Wu and Y.K. Malaiya, "A Correlated Faults Model for Software Reliability", *Proc. IEEE Int. Symp. on Software Reliability Engineering*, pp. 80-89, Nov. 1993
7. R.T.C. Deckers, P.L. Janson, F.H.G. Ogg, P.J.L.J. van de Laar, "Introduction to Software Fault Tolerance", *Technical Note*, Koninklijke Philips Electronics N.V. 2006
8. Mujava, URL: http://ise.gmu.edu/~ofut/mujava/
9. J. Offutt and Yu-Seung Ma, "Description of Class Mutation Operators for Java", URL: http://ise.gmu.edu/~ofut/mujava/mutopsClass.pdf, Nov. 2005
10. J. Offutt and Yu-Seung Ma, "Description of Method-level Mutation Operators for Java", URL: http://ise.gmu.edu/~ofut/mujava/mutopsMethod.pdf, Nov. 2005
11. C. Szyperski, "Component Software: Beyond Object-Oriented Programming", *2nd Edition, Addison Wesley Professional*, 2003
12. J. C. Laprie, B. Randell, A. Avizienis, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing", *IEEE Trans. Dependable Secure Computing*, pp. 11-33, 2004
13. Summary of regular-expression constructs in Java, URL: http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html

14. J.-C. Laprie, B. Randell, A. Avizienis, and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing”, *IEEE Trans. Dependable Secure Computing*, Vol. 1(1), pp. 11-33, 2004
15. A. Beugnard, J.-M. Jezequel, N. Plouzeau and D. Watkins, “Making components contract aware”, *Computer*, Vol. 32, pp. 38-45, Jul. 1999