International Journal of Web Engineering and Technology

ISSN online: 1741-9212 - ISSN print: 1476-1289
https://www.inderscience.com/ijwet

An architecture-based modelling of fault-tolerant SOA-based systems

Swati Goel

DOI: 10.1504/IJWET.2023.10056507

Article History:

Received:	15 May 2022
Last revised:	05 November 2022
Accepted:	30 November 2022
Published online:	31 May 2023

An architecture-based modelling of fault-tolerant SOA-based systems

Swati Goel

School of Computer and Systems Sciences,
Jawaharlal Nehru University,
New Delhi, India
Email: swatigoel96@gmail.com

Abstract: Service-oriented architecture (SOA) may effectively be implemented in distributed software development like cloud computing, internet of things, etc. Expectations from these systems, in terms of their reliability and availability, are increasing day by day. Fault-tolerance is an approach that ensures the correct functioning of the system even in the presence of faults. Various fault-tolerance mechanisms may be applied on a system. It is a cumbersome task to decide which mechanism is more suitable for a specific situation. Implementation of fault-tolerance involves as an additional cost factor due to the redundancy of software components. One has to intelligently decide what level of redundancy needs to be applied in a system. In this paper, a fault-tolerance policy, at an architectural level, is proposed for an SOA-based system. The proposed policy is based on the severity analysis of various software services in an SOA-based system. Architecture analysis and design language (AADL) is used for the modelling of the system. Fault tree analysis and functional hazard assessment have been used for severity analysis. A 'smart home security system (SHSS)' is used for the demonstration of the practicality of the proposed model. The proposed policy can be used as a fault-tolerance solution.

Keywords: fault-tolerance; service-oriented architecture; SOA; architecture analysis and design language; AADL; error model annex; EMA; functional hazard assessment; FHA; fault tree analysis; FTA.

Reference to this paper should be made as follows: Goel, S. (2023) 'An architecture-based modelling of fault-tolerant SOA-based systems', *Int. J. Web Engineering and Technology*, Vol. 18, No. 1, pp.4–28.

Biographical notes: Swati Goel received her PhD in 'Fault Tolerance Enhancement in Service Oriented Architecture' from SC&SS, JNU, New Delhi. Her research work in the areas of software engineering and fault tolerance has been published in several journals and conferences.

This paper is a revised and expanded version of a paper entitled 'Architecture level fault tolerance modeling for SOA based systems' presented at SSIC 2021, Manipal University Jaipur, India, 22–23 January 2021.

1 Introduction

Service-based systems are distributed in nature. Services may be obtained from multiple sources and integrated through loose coupling providing flexibility and scalability during system development. An involvement of various service components in a system causes new faults sources. Faults may come due to single services and their compositions. For designing reliable systems, these newer faults have to be considered and the possibility of system functionality in the presence of faults. Safety-critical systems cannot underestimate faults because these faults may lead to hazards. Hence, fault-tolerance is one of the important aspects of a service-oriented architecture (SOA)-based system development that needs to be handled for delivering quality software.

The basis of fault-tolerance is redundancy, i.e., to have redundant software components so that in case of failure of one component, control can be switched to redundant component and failure can be masked. Due to the redundancy cost, it is not practical to apply fault-tolerance to all the services in a SOA-based system. Different services may have different failure impacts. For appropriate implementation of fault-tolerant mechanisms, one must have to identify the components, in the system, which requires redundancy. Identification of such components is a challenging issue because components need to be identified based on some criticality criteria. There are different fault-tolerance mechanisms for software systems. It is also an important question that what type of fault-tolerance mechanism is required in specific circumstances.

There is several fault-tolerance approaches observed in the literature for SOA-based systems. Dobson et al. (2005) have presented a container-based approach for fault-tolerance in SOA-based systems. “The container is built with an XML fault tolerance policy model which supports fault tolerance mechanisms to be applied at an application level.” Bin Zheng et al. (2015) mentioned the design of static and dynamic fault-tolerance strategies, as well as the major problems while designing fault-tolerance strategies. Qiu et al. (2014) proposed a component ranking model, named ROCloud. ROCloud identifies significant components whose failures would have a great impact on application reliability based on the application structure information and components’ reliability properties. Zheng et al. (2012) have proposed FTCloud, a component-based ranking framework to design fault-tolerant applications running on the cloud. Although the above contributions deal with fault-tolerance policies in service-based systems, their approach is based on the ranking of software services. Furthermore, these approaches miss the data and control propagations among software services and criticality analysis based on these propagations. The avoidance of the propagation of erroneous information within the system is an important concern. We humbly extend the above contributions further by proposing an architecture-level fault-tolerance implementation policy based on service criticality.

The propagation of fault has to be avoided in the early phase, otherwise a fault will be transformed into a failure and in the end there is high probability of overall system getting collapsed. A system will be called as high degree fault-tolerant system if it is able to detect errors with the most possible brevity, after its occurrence and also if it is able to avoid the propagation of erroneous information within the system. Therefore, to addresses this issue – a fault-tolerance policy at an early stage may help to understand the system in a better way and one may proceed for reliable design and development of the

system. In this paper, an architecture-level fault-tolerance policy is proposed based on the criticality analysis. The major contributions of this paper are presented in a three-fold manner:

- The understanding of the faults in the context of a SOA-based system is very essential. In the first section, the possible faults that may occur, in a SOA-based system, are identified and demonstrated with the architecture analysis and design language (AADL).
- In the second section, functional hazard assessment (FHA) of software services, in a SOA-based system, is performed by the failure modes and effects analysis (FMEA) and fault tree analysis (FTA) techniques of AADL.
- A fault-tolerance policy is proposed based on the criticality analysis and FHA.

In this way, the probability of software malfunction due to the occurrence of faults can be minimised by handling faults at an architectural level. A case study of 'smart home security system (SHSS)' is designed and developed, using AADL and error model annex (EMA) and behaviour model annex, for the demonstration of the proposed approach.

The rest of the paper is structured as follows. The significant work related to our approach is briefly summarised in Section 2. In Section 3, the AADL along with EMA and behaviour model annex is briefly mentioned. The proposed architecture level fault-tolerance model for SOA-based system is introduced in Section 4 and in the following subsections. The description of the 'SHSS' using AADL is given in Section 5. Finally, the work is concluded in Section 6.

2 Related work

In literature, many methods and technologies have been proposed to describe, analyse and tolerate hardware faults at an architecture level. It is difficult to explain software faults properly as fault propagates at each level, and identifying the root cause of the fault is a tedious task (Sun et al., 2007). In the literature, we found few works that deals with tolerating or handling faults at an architecture level in SOA-based systems through AADL. Mahdian et al. (2009) in their work, have proposed an approach for faults detection and tolerance in service-based systems at an architecture level. They have used redundancy-based FT mechanism for adding new components to the presented architecture. Gabsi et al. (2016) in their work, have proposed a model-driven approach and firstly, they have generated fault-tolerance code using AADL. Secondly, they have defined transformation rules from the EMA annex, which is a sublanguage of AADL. The work of Mahdian et al. (2009) and Gabsi et al. (2016), are similar as both have implemented fault-tolerance at an architecture level to stop the propagation of faults to further stages but the difference lies in their FT mechanism implementation. Mahdian et al. (2009) have used redundancy whereas Gabsi et al. (2016) have handled faults through AADL code itself without replicating components. The limitations of the above discussed works are that they directly dealt with the faults tolerance, without identifying the possible faults. Therefore, the discussed approach may not be an efficient technique in tolerating faults in service-based systems at an architecture level

Some of the works that we found in the literature deals with model-based approach and designing algorithm to tolerate faults at an early stage of system development. Feiler

and Rugina (2007), in their work, have used AADL to show architectural patterns in any system being analysed to find out all possible issues in the system. To address fault-tolerance concisely, they have used the modelling capability of AADL which allowed them to describe the redundancy feature of the system's architecture. Buys et al. (2011), in their work, have presented a novel dependability technique which supports advanced redundancy management, which aims to autonomously tune its internal configuration because of changing context. Sokolsky and Chernoguzov (2014) have used AADL to capture the architecture of embedded systems in terms of software and platform in a component-oriented manner. Fekih et al. (2019) in their work have proposed a sensor and repair-based algorithm to adjust service-based applications, which is very efficient whenever a service composition process fails. In addition, this approach can detect faulty services very rapidly. The discussed approach is capable of handling faults but the authors have not defined any specific criteria according to which fault-tolerance can be applied. Therefore, the approaches may not be practically feasible. Hence, a well-defined fault-tolerance policy is highly required in future.

In the literature, we found that the application of fault-tolerance is not limited to one field but it is expanding in every domain. One of the work, Zhang et al. (2021), in their work, has proposed a fault-tolerant model for a fog system's performance optimisation. The fault-tolerant methodology proposed in this work is based on calculating the steady-state probabilities and substituting the faulty fog nodes with the most appropriate ones. The experimental results have been performed on a real-time system. Another work, Zhang et al. (2018), have presented an online fault detection technique on cloud. The proposed online fault detection technique can largely help cloud managers to take corrective actions on time before fault occurrence in clouds. The work discussed by Zhang et al. (2021) is little bit similar to our work as it has discussed the fault-tolerance model for dynamic fog nodes.

The work presented in this paper is different in the aspect that along with fault modelling at the architecture level, it also considers criticality analysis of software services. It describes how various fault-tolerance mechanisms may be effectively applied in a system based on severity analysis of software services.

3 AADL

AADL is an architecture design language, which focuses on system design specification by using formal semantics that can be used to analyse systems already in use and integrate new systems (Feiler et al., 2006). AADL is standardised by SAE International, used for describing system components and their interactions with its operating environment (i.e., processors, bus, devices) (Delange et al., 2014). "The AADL framework supports an architecture-centric, model-based development approach throughout the system lifecycle" (Carnegie Mellon University, no date).

The AADL EMV2 Error Library is a rich, semi-formal embedded system modelling language which is deeply integrated into AADL (Larson et al., 2013). The errors included in the library have formalised semantics and the library is designed to be easily extended by system developers to become domain- or system-specific. Specifically, the EMV2 annex provides an ontology of system errors and formal specifications of the semantics of the error types in the library (Procter, no date). Specifically, it allows engineers to formally specify errors, error propagation and error mitigation (Larson et al., 2013).

The AADL behaviour annex [SAE, Annex X Behavior Annex (AS5506-X Draft-2.13), no date] is an AADL sublanguage that is used to define the behaviour of an AADL application model. The behaviour annex addresses many challenges which are as follows:

- First, it requires to parse and analyse several sub-languages.
- Secondly, to complete its analysis requires consistency with the core language.
- Thirdly, the internal representation of the annex needs to be compliant with the core language-internal representation (Lasnier et al., 2011).

Figure 1 Demonstration of components interactions via bus and processor (see online version for colours)

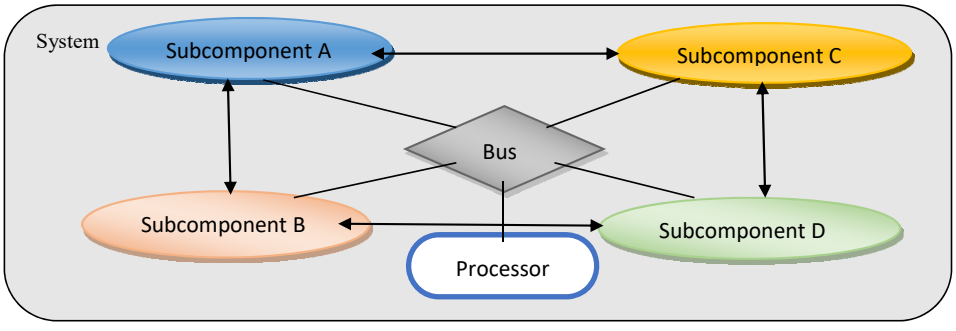
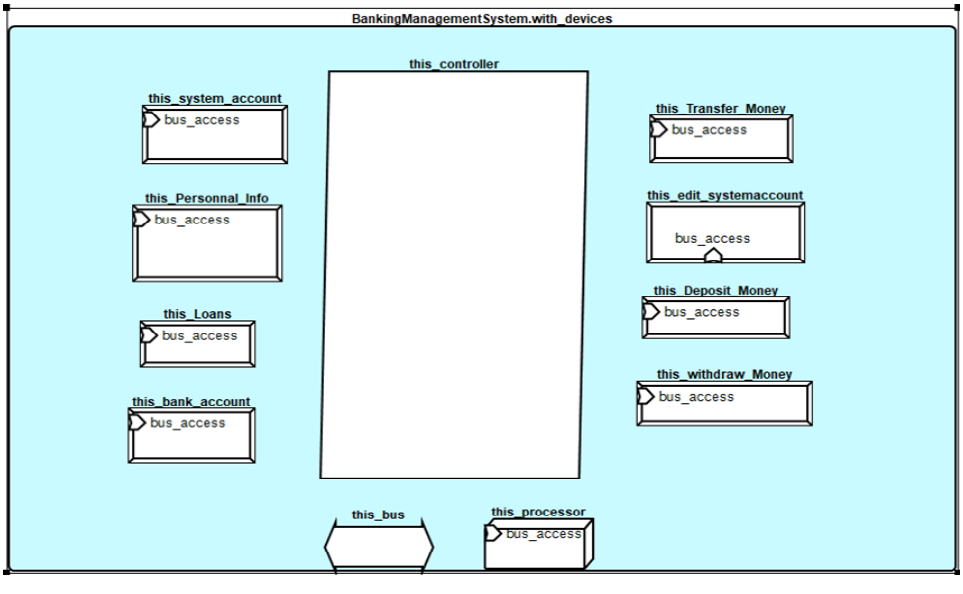


Figure 2 'Online banking system' and its subcomponents (see online version for colours)



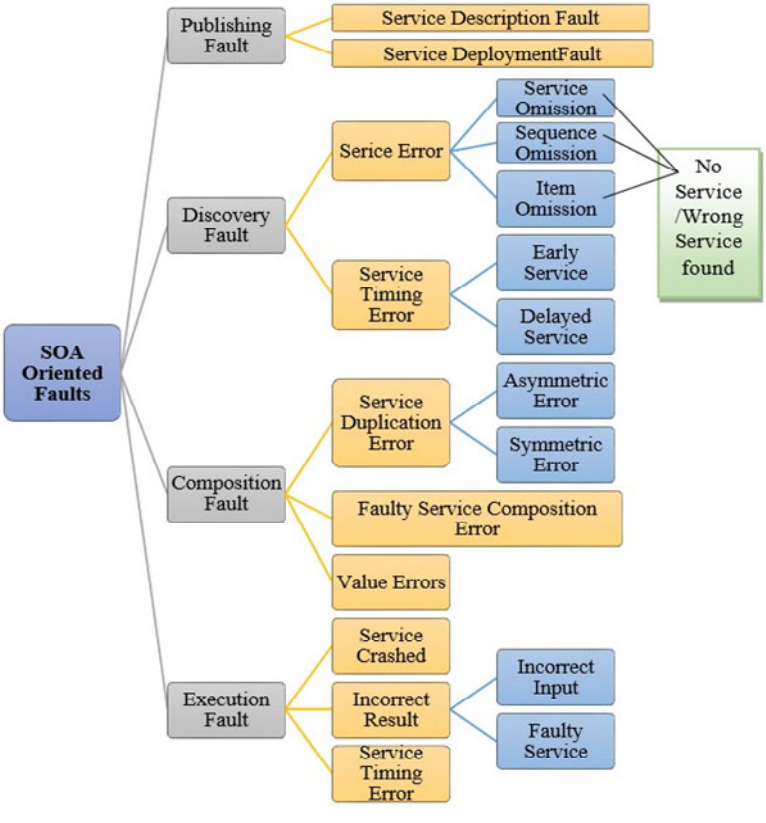
A pictorial representation is drawn here in Figure 1 to demonstrate how a system can be modelled using AADL. The system is modelled in terms of its subcomponents. The system component and its various subcomponents are organised hierarchically. The 'bus'

represents a virtual logical connection among various subcomponents. The component 'processor' represents the processing unit. Figure 2 shows modelling of 'OnlineBankingSystem' in AADL.

Figure 3 Error path specification between subcomponents (see online version for colours)

```
device ApplyLoans
    features
    Apply_Loans:in event port;
    Loans_Approved:out data port;
    bus_access:requires bus access HWconnection;
    annex emv2 {**
                use types Error_Library;
                use behavior Error_Library::FailAndRecover;
            error propagations
            Apply_Loans: in propagation {DiscoveryFaults};
            Loans_Approved: Out propagation {ServiceCrashed};
            flows
                f3: error path Apply_Loans{DiscoveryFaults}->Loans_Approved{ServiceCrashed};
            end propagations;
```

Figure 4 A possible summarisation of SOA-specific faults (see online version for colours)



In the 'online banking system', a subcomponent like *CreateBankAccount* is propagating an error. It can be written in AADL as shown in equation (1).

$$BankAcct\_out: out\ propagation\ \{ServiceOmission, SequenceOmission, ItemOmission\} \tag{1}$$

that means *CreateBankAccount* component is propagating *ServiceOmission*, *SequenceOmission* and *ItemOmission* errors. Similarly, error flow path form in port to out port in AADL between two subcomponents is shown in equation (2).

$$ef: error\ pathEditSysAcct\_in\{Executionfaults\} \rightarrow BankAcct\_out\{ServiceOmission, SequenceOmission, ItemOmission\} \tag{2}$$

An error path is used to specify how an error propagates into the component of a system, remains inside the component, and goes out of the component through an outgoing feature or binding. Error path of the device *ApplyLoans* is shown in Figure 3. On the event port, *Apply_Loans*, *DiscoveryFault* is occurring, which is inside the *ApplyLoans* component and it goes out through an out propagation on out data port *Loans_Approved*. *Discovery fault* was converted to *service crashed fault*, which is an extension of *execution fault* as shown in Figure 4.

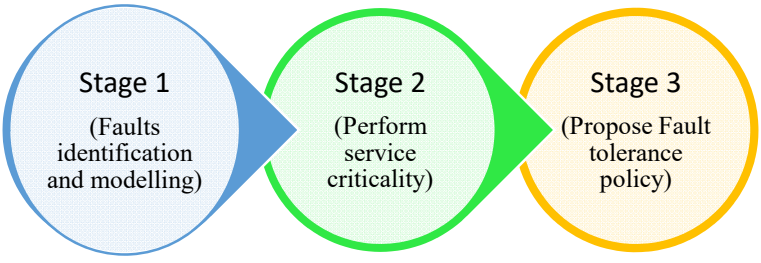
4 The proposed architecture level fault-tolerance model for SOA-based system

SOA provides the mechanism to develop software with the help of web services via a loose coupling concept. This flexibility makes the architecture feasible for distributed applications like cloud computing, internet of things, service-based systems, etc. As its use is vastly increasing day-by-day, consumers start to expect the reliability and continuous availability of such systems even in the presence of faults. It drives us to think beyond fault identification and their handling. Fault-tolerance provides a facility that masks the failure into the system through redundancy. Earlier approaches of fault-tolerance were based on hardware redundancy and they have the thought in the mind that hardware components are degraded with time. Most of the approaches were based on operational faults. This concept is not applicable in the case of software components as their performance is not degraded with time, but the business and performance requirements may be changed. In the case of service-based fault-tolerance approaches, one has to deal with design faults because software services do not have operational faults. One has to reconsider the approaches keeping software services features in mind. The proposed approach is presented in a three-fold manner as shown in Figure 4.

Figure 5 shows the complete proposed methodology adopted to tolerated faults at an architectural level. First, an understanding needs to be developed regarding the possible faults in a SOA-based system. SOA-specific faults are identified and modelled using AADL. The idea is to obtain the criticality of various software services and propose an appropriate fault-tolerance policy based on their criticality and failure impact. Therefore, in the second step, criticality analysis of various software services, in an SOA-based system, is performed through FHA. In the third step, an appropriate fault-tolerance policy

is proposed based on the information obtained from criticality analysis. The detailed model is described in the following subsections.

Figure 5 Proposed methodology (see online version for colours)



4.1 SOA-specific fault modelling through AADL

In this subsection, an effort has been made to find out the possible faults that may occur in a SOA-based system. A *fault* can be described as a defect in physical structure, irregularity, or flaw that occurs in a software service. Hence, it can be said that faults can cause errors and errors can lead to the failure of a system. To perform fault-tolerance analysis for a SOA-based system, at the architecture level, a clear understanding of all possible faults is necessary. For this purpose, SOA-specific faults are identified and summarised in Figure 5. SOA-specific faults can also be characterised by the phases in which they are introduced, i.e., *design faults and operational faults*. Here, only design faults are discussed.

Conventionally, there are five stages in SOA, which are *publishing*, *discovery*, *composition*, *binding* and *execution* (Niknejad et al., 2020). Faults may occur during all steps of a SOA-based system. These errors can cause deviation from computational accuracy, which will result in a failure unless the SOA-based system is capable of tolerating those errors. In Figure 4, possible faults at each stage are mentioned. An 'online banking system' domain is used as an example for demonstrating various faults.

4.1.1 Publishing faults

For the sake of the selection of a service, its description is to be made public. *Service description faults* mostly occur because of the incorrect service description. The description may itself be faulty (*incorrect description*) or there might be some functional/performance mismatch during the deployment of the service. *Service deployment fault* occurs when there is an error in deploying the service on the target platform. For example, Figure 6 shows that in 'online banking system', *UpdatePersonnalInfo* subcomponent is propagating publishing fault throughout data port, i.e., *Update_Info*. The component *CreateSystemAcc*. is propagating incorrect service, i.e., *service description error*. The details entered by the user while creating a system account that is mismatched with the deployed service by the service provider.In case the user enters the details while creating a system account that is mismatching with the deployed service by the service provider. In this case, the *CreateSystemAcc* is getting *ServiceDescriptionError*. Figure 6 shows the AADL modelling of *publishing faults*.

Figure 6 AADL modelling of *PublishingFaults* (see online version for colours)

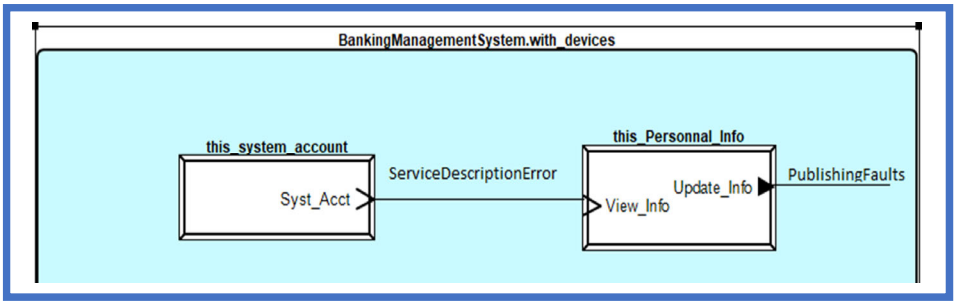
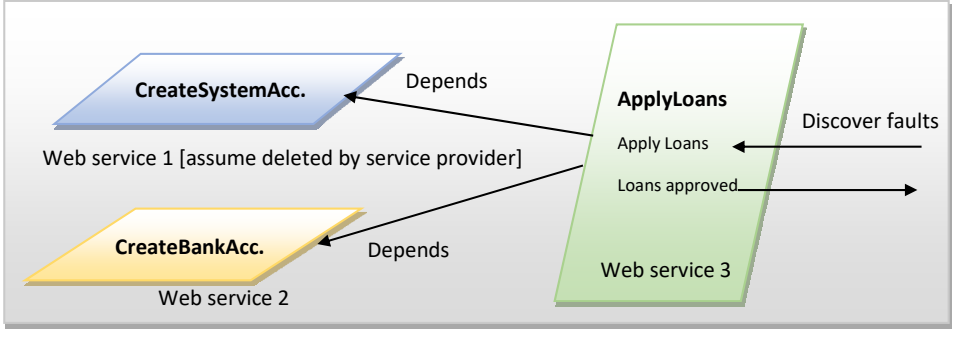
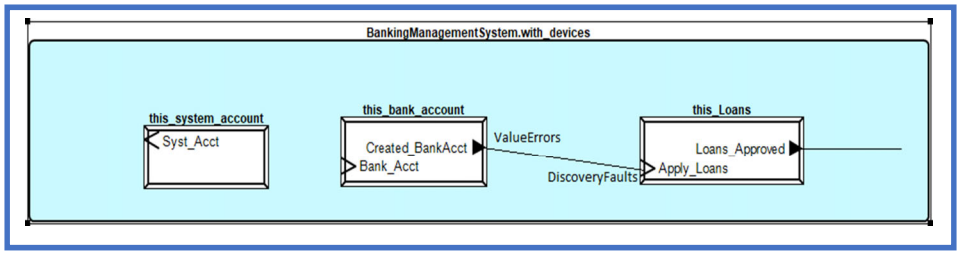


Figure 7 (a) *DiscoveryFault* functioning (b) AADL modelling of *DiscoveryFaults* (see online version for colours)



(a)



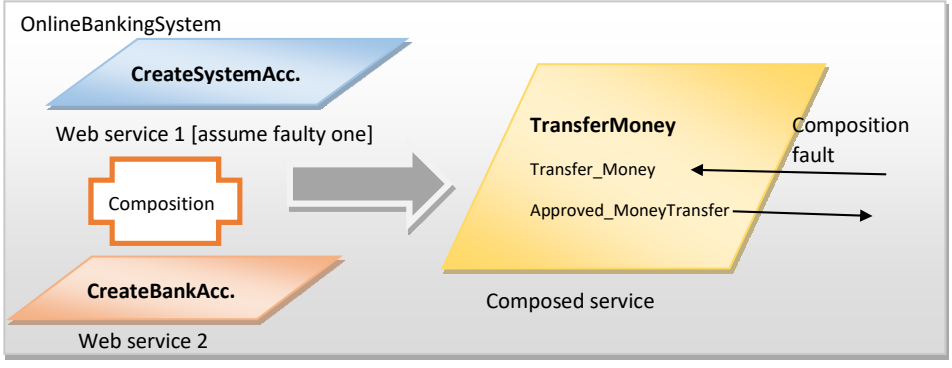
(b)

4.1.2 Discovery faults

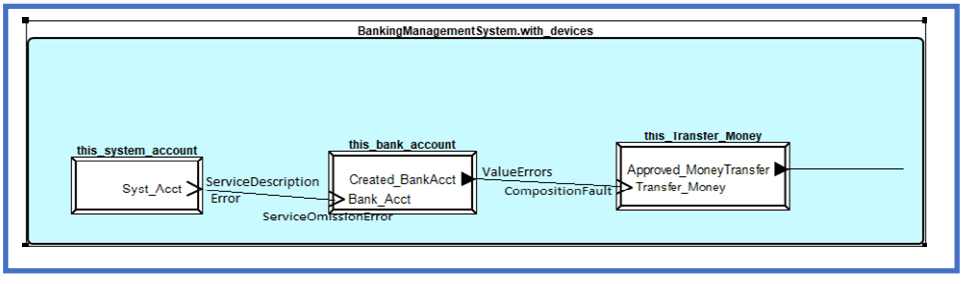
Discovery fault may occur during the search process of suitable services at service provider locations. It can be of two types – *service errors* and *service timing errors*. *Service error* is further divided into service omission, sequence omission and item omission. *Service omission error* occurs while communicating with an unreachable service that means now the service which the user is trying to search has been deleted by the service provider. Similarly, a *sequence omission error* occurs when the service provider has deleted interlinked services from the service repository. *Item omission error* occurs when some specific functionality has been omitted from the service. In the above

three service errors, the fault information that will be passed to users is *no service/wrong service found*. *Service timing error* occurs when the service is not available at the required time it will be either before the time (*early service arrival*) or after the time (*delayed service*) in an unsynchronised manner. For example, Figure 7(b) shows in an 'online banking system', the *ApplyLoans* component may receive *DiscoveryFault* at in event port, i.e., *Apply_Loans* because this component is linked to various other components *like CreateBankAccount and CreateSystemAccount* to complete its functionality. Hence, if web service 1 is deleted by the service provider, the web service 3 while fetching web service 1 will get *DiscoveryFaults* because it is trying to fetch that service that has been deleted. Figure 7(a) shows *discovery faults* functioning whereas Figure 7(b) shows AADL modelling of *discovery faults*.

Figure 8 (a) A possible execution fault in a system (b) AADL modelling of a possible *CompositionFault* (see online version for colours)



(a)



(b)

4.1.3 Composition faults

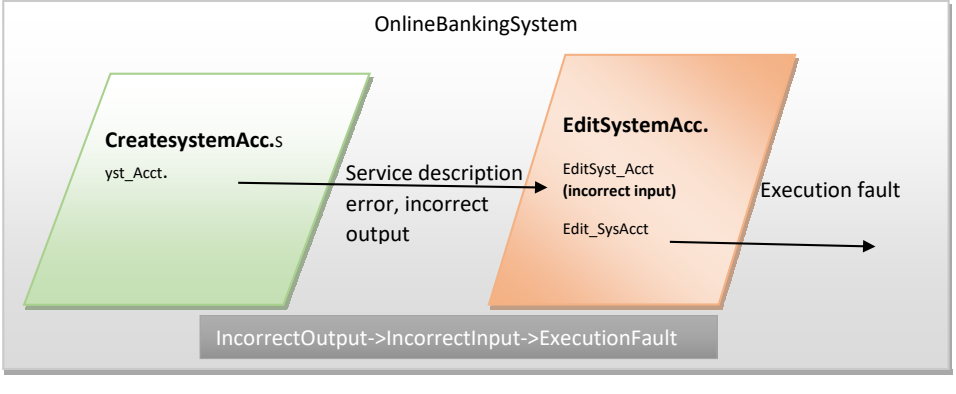
The process of composing a new service from existing individual services is called web service composition. Composition fault occurs mainly when individual services are not able to integrate properly or after integration, the functionality/performance of the composed service degrades. There can be many reasons for this like while composing a new service some faulty services have been used (*faulty service composition error*) *or* usage of redundant services (*service duplication error*). During the service composition process asymmetric and symmetric matching between service attributes is performed.

The mismatch between service attributes results in respective *asymmetric and symmetric errors* (Kushal et al., 2017). For example, Figure 8(b) shows *TransferMoney* subcomponent in 'online banking system' may get composition fault at in event port, i.e., *Transfer_Money* because this service has been composed of multiple other services *like CreateSystemAccount*, *CreateBankAccount*, etc. Therefore, if any service in the composition process is wrong then there will be a possibility of composition fault occurrence. Figure 8(a) shows composition fault working and Figure 8(b) shows the AADL modelling of composition fault.

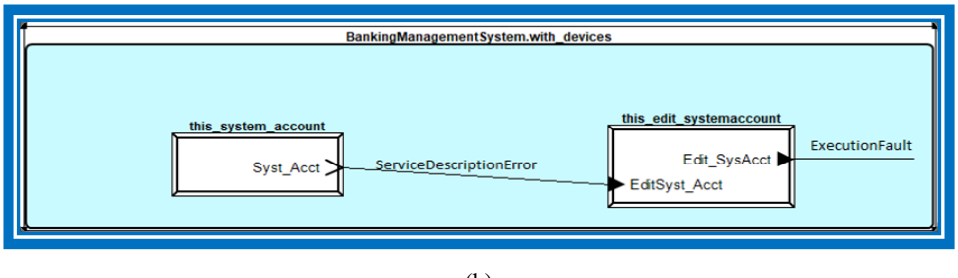
4.1.4 Execution fault

Execution fault occurs during the execution of service or when the output of the service does not match with the expected output of the executing service (*incorrect output*) and there can be many reasons for this like *incorrect input, faulty service* that is due to software malfunction. There is also a possibility of service getting crash (*service crashed*) the server will notice this and notify the client about the failure. For example, Figure 9(a) shows *Edit_SysAcct* out data port of *EditSystemAccount* subcomponent is receiving *Executionfault* in 'online banking system' because of *CreateSystemAcc*. Subcomponent which is transmitting *servicedescriptionerror* due to which *Edit_SysAcct* is receiving *incorrect input* and causing execution fault through *Edit_SysAcct*. Figure 9(b) shows AADL modelling of execution fault.

Figure 9 Execution fault functioning (b) AADL modelling of *ExecutionFault* (see online version for colours)



(a)

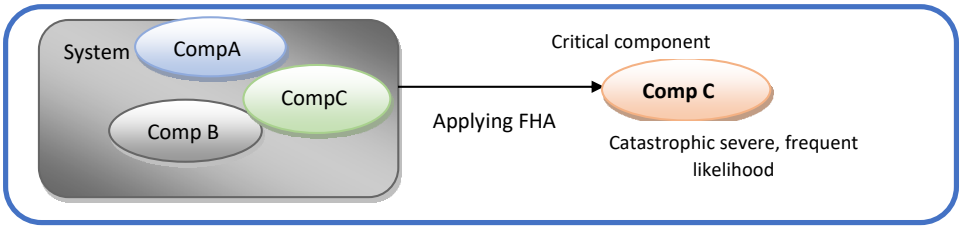


(b)

4.2 Criticality analysis through FHA

Understanding various faults, in a SOA-based system, will help in realisation of the effect of a particular fault. But the proper impact of a fault can only be assessed through a FHA. In this section, a FHA has been performed to find out the level of criticality of software services. In addition, the entire system cannot be made redundant. Hence, analysis is done to find out the critical components from a complex system by applying safety analysis technique, i.e., FHA. Figure 10 shows a system having three components A, B and C. To perform FHA, it is required to obtained information about some specific properties like severity, occurrence distribution, likelihood, etc. While performing FHA, every component property is scanned and a component having high catastrophic severity and frequent likelihood is considered to be a critical component.

Figure 10 Hazard analysis of software services (see online version for colours)



FHA will be performed in OSATE (Download and Installation – OSATE 2.10.0 Documentation, no date) to identify abnormal conditions that may cause an error. The tool will scan each component and communicate information about all mentioned error events and error sources and an analysis report is generated that includes the list of all possible hazards. The severity and likelihood are categorised into five ranges and labels respectively according to OSATE tool (Download and Installation – OSATE 2.10.0 Documentation, no date) standards as shown in Table 1.

Table 1 Severity and likelihood classification

Severity classification	*Ranges*	*Likelihood classification*	*Labels*	*Components categorisation*
Catastrophic	1 (high)	Frequent	A (high)	Highly critical
Hazardous	2	Probable	B	Critical
Severe major	3	Remote	C	Average
Major	4	Extremely remote	D	Less critical
Minor	5 (low)	Extremely improbable	E (low)	Very less critical

4.3 The proposed fault-tolerance policy at the architecture level

In this step, a fault-tolerance policy is proposed to tolerate faults based on the FHA technique as discussed above in Section 4.2. FT policy has been proposed at an architecture level because it will prove to be very beneficial in identifying and eliminating design issues at an early stage of software development. This FT policy allows faults and their failure and propagation effects to be identified at the system level.

In our proposed FT policy, the list of used FT models is given in Table 2.

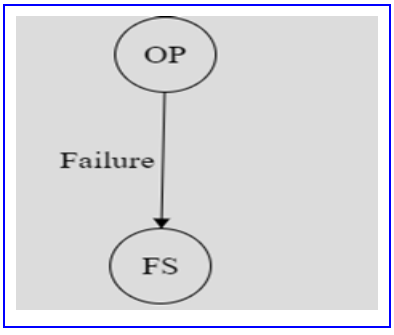
Table 2 List of FT models

FT model	*Description*
Fail-stop model	The system goes from operational to failed state on the occurrence of a failure event. No recovery mechanism.
Fail-and-recover model	The system can recover from failed state to an operational state by a recovery mechanism.
Permanent-transient-failure model	On the first failure, the system goes to failed transient state and can be returned to an operational state by applying a recovery mechanism.
Degraded-fail-stop model	On the first failure occurrence, the system goes to a degraded state but on the second failure, the system goes to a failed state.

4.3.1 Case 1: fail-stop FT model

The fail-stop FT model declares states – *operational* and *failed* and one error event *failure*. This error event triggers a transition between the two states. This model does not declare any recovery event. Hence, the component goes into a permanently failed state in case of failure occurrence event. Graphical representation is shown in Figure 11.

Figure 11 *Fail-stop* FT model – graphical representation (see online version for colours)



Note: OP: operational state and FS: failstop state.

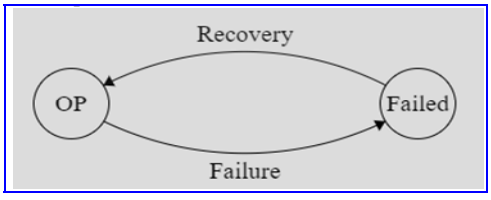
4.3.2 Case 2: fail-and-recover FT model

Fail-and-recover FT model also declares two states *operational* and *failed* and one error event *failure* in the same way as *fail-stop* model has declared. The only difference is that the model can recover from failure by using a recovery mechanism. Graphical representation is shown in Figure 12.

4.3.3 Case 3: permanent-transient-failure FT model

In this FT model on the occurrence of the first failure, the system goes to *failed transient state* and can return to an *operational state* by applying a recovery mechanism. The AADL code for this model is shown in Figure 13(a) and graphical representation is shown in Figure 13(b).

Figure 12 *Fail-and-recover* FT model – graphical representation (see online version for colours)

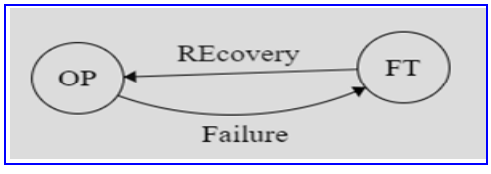


Note: OP: operational state.

Figure 13 (a) *Permanent-transient-failure* FT model AADL code (b) *Permanent-transient-failure* – graphical representation (see online version for colours)

```
error behavior PermanentTransientFailure
events
    Failure: error event ;
    Recovery: recover event;
states
    Operational: initial state;
    FailedTransient:  state;
    Failed:  state;
transitions
failtransition: Operational-[Failure]->(FailedTransient with EMV2::TransientFailureRatio, Failed with others);
    RecoveryTransition : FailedTransient-[Recovery]->Operational;
end behavior;
```

(a)



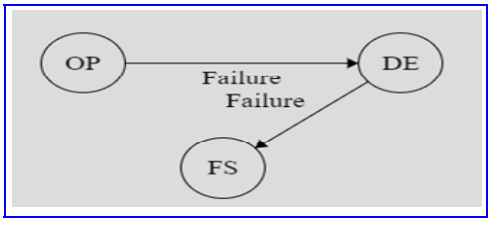
(b)

Note: OP: operational state and FT: failed transient state.

4.3.4 Case 4: degraded-fail-stop FT model

In this FT model on first failure occurrence, the system goes to a degraded state but on second failure the system goes to a failed state and cannot return to an operational state as it was in the permanent transient failure FT model. Graphical representation is shown in Figure 14.

Figure 14 *Degrade-fail-stop* FT model – graphical representation (see online version for colours)



Note: OP: operational state and DE: degraded state.

The proposed FT policy is as follows:

1 A fault-tolerance policy is proposed based on criticality analysis and hazard assessment. The motivation behind performing FHA is to find out all abnormal conditions, which may cause an error in the future. It is a comprehensive analysis tool. By using this analysis tool, a report is generated which contains information about all error sources and error events. For each component, various properties, i.e., severity, description, likelihood, etc. have to be defined on the relevant EMV2 artefacts. Figure 15 specifies the severity and likelihood occurrence for error events or error propagation as per standards given in OSATE 2.2. According to severity and likelihood, the components range is decided and based on that only, fault in the component will be handled. For example, catastrophic severe and frequent likelihood decides that the component is highly critical and faults will be tolerated by designing FT model and creating backups only of the critical components given in Figure 15.

Figure 15 Classification of components based on severity and likelihood (see online version for colours)

Severity	*Range*	*Likelihood*	*Labels*	*Components range*	*Tolerating faults*
Catastrophic	1 (high)	Frequent	A (high)	Highly critical	FT model + backup
Hazardous	2	Probable	B	Critical	Ft model
Severe major	3	Remote	C	Average	Ft model
Major	4	Extremely Remote	D	Less critical	FT model
Minor	5 (low)	Extremely improbable	E (low)	Very less critical	FT model

FT model can be FailStop, FailAndRecover, PermanentTransient Failure, and DegradedFailStop depending on application

2 Fault-tolerance is highly required in safety and business-critical applications. Safety-critical applications are the applications where the loss of life or environmental disaster has to be eliminated to perform smooth functioning. Redundancy, one of the fault-tolerance mechanisms, should be employed only for critical components. In this way, software FT has been implemented to improve the reliability of the overall system by making use of functionally equivalent critical backup components to tolerate component failure.

Figure 16 shows the proposed FT mechanism for critical components in our proposed FT policy. Whenever any error is detected in the primary component and the component fails due to that error the control is transferred to the backup component directly to ensure the continuity of the system. The proposed FT mechanism in Figure 16 is very beneficial while developing safety-critical systems, where software analysis and validation are major concerns

The summarised *SOA-specific faults* in Figure 4 are modelled in the EMA sub-clause of AADL as shown in Figure 17. The EMV2 constructs are similar to the syntax and styles as defined for AADL. Figure 16 models ‘online banking system’ module errors in the EMA.

Figure 16 Proposed FT mechanism for critical components (see online version for colours)

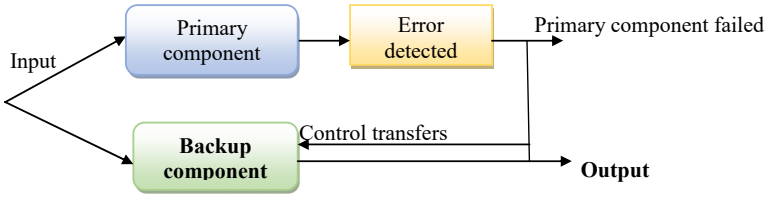


Figure 17 'Online banking system' error types (see online version for colours)

```
package Error_Library
public
annex EMV2 {**
error types
ServicePublishingFaults: type set { ServiceDescriptionError, ServiceDeploymentError};
PublishingFaults: type;
ServiceDescriptionError: type extends PublishingFaults;
ServiceDeploymentError: type extends PublishingFaults;
ServiceDiscoveryFaults: type set {ServiceError, ServiceTimingError};
DiscoveryFaults:type;
ServiceError: type extends DiscoveryFaults;
ServiceOmission: type extends ServiceError;
SequenceOmission: type extends ServiceError;
ItemOmission:type extends ServiceError;
ServiceTimingError: type extends DiscoveryFaults;
EarlyService: type extends ServiceTimingError;
DelayedService: type extends ServiceTimingError;
ServiceCompositionFaults: type set {ServiceDuplicationError, FaultyServiceCompositionError, ValueErrors};
CompositionFaults: type;
ServiceDuplicationError: type extends CompositionFaults;
AsymmetricError: type extends ServiceDuplicationError;
SymmetricError: type extends ServiceDuplicationError;
FaultyServiceCompositionError: type extends CompositionFaults;
ValueErrors: type extends CompositionFaults;
ServiceExecutionFaults:type set{ ServiceCrashed, IncorrectResult, TimingError};
ExecutionFaults: type;
ServiceCrashed: type extends ExecutionFaults;
IncorrectResult:type extends ExecutionFaults;
IncorrectInput: type extends IncorrectResult;
FaultyService:type extends IncorrectResult;
TimingError: type extends ExecutionFaults;
end types;
```

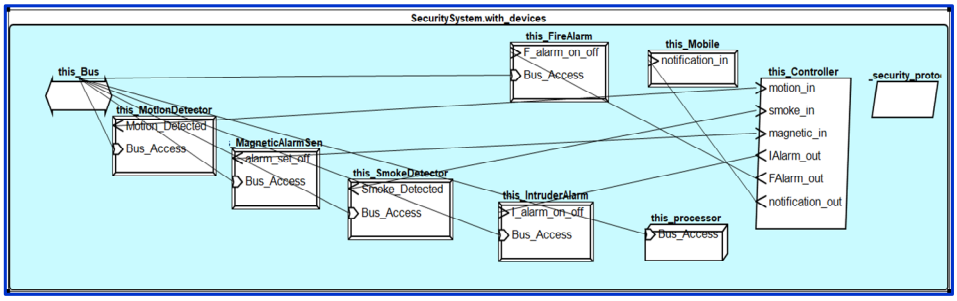
5 Case study

A case study of 'SHSS' has been taken to demonstrate the practicality of the proposed fault-tolerance policy at an architecture level depending on criticality analysis.

5.1 Architecture modelling of 'SHSS'

A 'SHSS' is designed through AADL for the demonstration of the proposed model. This case study has been taken from GitHub (no date) for performing the analysis of the proposed FT policy. The SHSS is comprised of the major components – *MotionDetector*, *MagneticAlarmSet*, *SmokeDetector*, *IntruderAlarm*, *fire alarm* and *Mobile* modelled as devices while *controller* and *securityprotocols* are modelled as processes as shown in Figure 18. All these components are used to design fault-tolerance models and error behaviour is associated with each system component to tolerate failures.

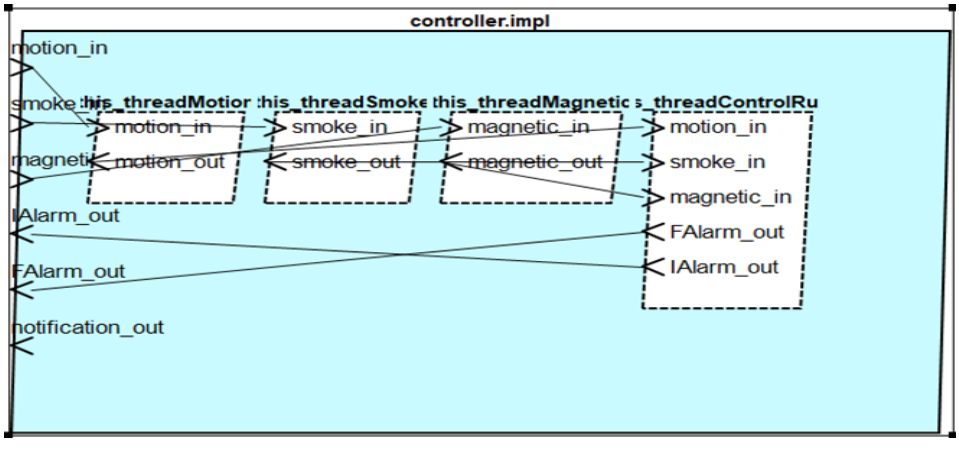
Figure 18 Architecture level service interaction – 'SHSS' (*GitHub*) (see online version for colours)



5.1.1 Fault modelling

AADL EMV2 allows one to specify error flows along with points of components interaction. In 'SHSS' service timing response plays an important role because everything is dependent on the correct timing of service. For example, a *FireAlarm* component encounters some service timing error while executing this component. In this scenario, the 'SHSS' will trigger the *FireAlarm* early, late, or not trigger. The homeowner or fire department will not be able to receive the notification of fire on time due to the occurrence of *ServiceTimingError* on the *FireAlarm* component. The overall purpose of designing 'SHSS' may diluted. Hence, a fault-tolerant 'SHSS' has been designed to tolerate the *service timing error*. Figure 19 shows the error flow in 'SHSS'.

Figure 19 Error flow – 'SHSS' (see online version for colours)



5.1.2 Applying fault-tolerance in the 'SHSS'

In order to incorporate fault-tolerance features in 'SHSS', safety analyses techniques have been used which involves architectural modelling such as FHA, FMEA, FTA and common-mode assessment (CMA) (Kushal et al., 2017). The architecture is designed and

its associated fault model is devised in open source AADL tool environment (OSATE) (Download and Installation – OSATE 2.10.0 Documentation, no date).

1 *FHA:* The likelihood of occurring *ServiceTimingError* at *F_alarm_on_off* port is taken as frequent, which means that there is a very high probability of this fault because the possibility of occurring *discovery fault* is mainly during positioning web services by service providers and while fetching service descriptions that have been made public. The severity of arising *ServiceTimingError* at *F_alarm_on_off* port is assumed as catastrophic because if an alarm does not ring or rings late, can cause loss of life. Similarly, the likelihood and severity of *ServiceTimingError* of *servicecrashed fault* is taken as frequent because of the uncertain nature of this fault. The generated FHA report from OSATE 2.2 tool is shown in Table 3.

Table 3 Generated FHA report from OSATE

Component	*Description*	*Failure*	*Severity*	*Likelihood*	*Comment*
this_MotionDetector	'Motion detector sensor failure'	'Faulty sensors'	'Hazardous'	'Probable'	'Critical'
this_MagneticAlarmSensor	'Magnetic alarm sensor failure'	'Faulty sensors'	'Hazardous'	'Probable'	'Critical'
this_SmokeDetector	'Smoke Alarm Failure'	'Faulty sensors'	'Catastrophic'	'Frequent'	'Highly critical'
this_IntruderAlarm	'Unlocked or loose doors and windows'	'Unknown faults'	'SevereMajor'	'Remote'	'Average'
this_FireAlarm	'false alarms or unwanted fire alarm activations'	'Fire alarm fail'	'Catastrophic'	'Frequent'	'Highly critical'
this_Mobile	'Early or delayed notifications'	'Not receiving notifications'	'Hazardous'	'Probable'	'Critical'

From Table 3, it is clear that the components *this_SmokeDetector*, *this_FireAlarm* is a highly critical component because of catastrophic severity and frequent likelihood. Hence, to tolerate faults occurring in these components, backup of only these components need to be created and this will be a cost-effective mechanism. Table 4 lists all components in the 'SHSS' and based on the FHA report the components are classified into the five ranges (as given in Table 1), ranging from highly critical to very less critical. According to the criticality of components, the proposed FT policy has been applied to tolerate the present faults stated in Table 4.

Table 4 Tolerating faults in 'SHSS'

	Criticality level	*FT model applied*	*FTPolicy-redundancy*
this_MotionDetector	Critical	Fail-and-recover model	Not applied
this_MagneticAlarmSensor	Critical	Permanent-transient failure	Not applied
this_SmokeDetector	Highly critical	Fail-and-recover model	Backup created
this_IntruderAlarm	Average	Permanent-transient failure	Not applied
this_FireAlarm	Highly critical	Permanent-transient failure	Backup created
this_Mobile	Critical	Fail-stop model	Not applied

5.1.3 Demonstration of FT policy on 'SHSS'

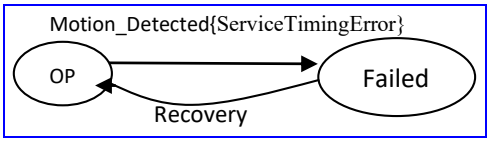
To tolerate faults of less critical components a fault-tolerance model is generated that can be associated with AADL components. In our case study of 'SHSS', the proposed FT model has been designed for all the components listed in Table 4. This model has been used to show various information related to faults present in interconnected communication networks of components. To draw an FT model following points need to be taken into account:

1 failure of a component or system

2 component behaviour in the presence of faults in terms of operational and failed states.

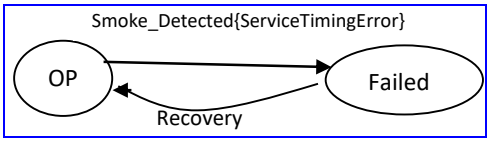
The proposed FT model is used to recover from the *failed state* and that can be further associated with any other component to tolerate faults. In this section, the faults in the components, i.e., *ServiceTimingError* have been tolerated through various FT models whereas Section 5.3 explains tolerated faults in critical components through redundancy. For example, to avoid failure of *MotionDetector* and *SmokeDetector* components, the FT model has been drawn as shown in Figure 20 and Figure 21, respectively. This generic *fail-and-recovery* FT model has been modified by taking into account an occurrence of *ServiceTimingError* at an out event port – *motion_detected* and *smoke_detected* of *MotionDetector* and *SmokeDetector* components, respectively. The fault at an out event port is tolerated by recovery transition as shown in Figure 20 and Figure 21. Motion sensors in 'SHSS' automatically turn on a light or sound an alarm whenever movement is detected in the house, and a motion sensor will activate the camera to start recording. In Figure 20, the OP state is when motion is detected on time. If the motion is not detected on time *ServiceTimingError* occurs, the system goes to the failed state. The system can be back to OP state by rolling back through the recovery mechanism. Similarly, *SmokeDetector* in 'SHSS' triggers the alarm whenever smoke enters the house and crosses the path of the light beam, light will be scattered by the smoke particles, pointing towards the sensor. In Figure 21, the OP state is when smoke is detected on time. If smoke is detected late, *ServiceTimingError* is encountered at an out event port. The system goes to failed state temporarily, the system is rolled back to the OP state through a recovery mechanism.

Figure 20 FT model – *MotionDetector*



Note: OP: operational state and Failed: failed state.

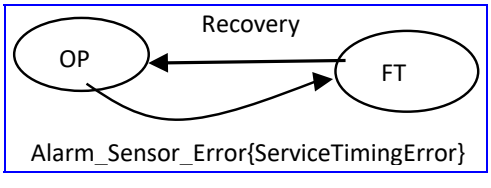
Figure 21 FT model – *SmokeDetector*



Note: OP: operational state and Failed: failed state.

To avoid failure of *MagneticAlarmSensor*, *FireAlarmSensor*, and *IntruderAlarmSensor* components, the FT model has been drawn as shown in Figure 22. This generic *permanent-transient failure* FT model has been modified by taking into account an occurrence of *ServiceTimingError* at an out event port *alarm_set-off* of *MagneticAlarmSensor* component and at in event port *F_alarm_on_off* and *I_alarm_on_off* of *FireAlarmSensor* and *IntruderAlarmSensor* components, respectively.

Figure 22 FT model – *MagneticAlarmSensor*

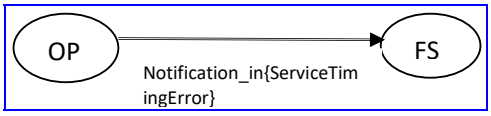


Note: FT: failed transient state.

Fire alarm systems in 'SHSS' automatically detects an event that can cause a fire. This system receives signals from the fire sensor (smoke, heat or carbon monoxide detector) and sends them to the fire alarm panel. The panel triggers the alarm. If the fault occurs, the FT model is applied on the occurrence of the first failure; the system goes to *failed transient state* and can return to an *operational state* by applying a recovery mechanism shown in Figure 22. The OP state comes when the alarm beeps on time. In the case of *ServiceTimingError* occurrence, it can be tolerated through the permanent-transient failure FT model as discussed. Similarly, the *ServiceTimingError* can be tolerated in *MagneticAlarmSensor* and *IntruderAlarmSensor* components following the FT model shown in Figure 22.

To avoid failure of the *Mobile* component, the FT model has been drawn as shown in Figure 23. This generic *fail-and-stop* FT model has been modified by taking into account an occurrence of *ServiceTimingError* at in event port *notification_in*. The *Mobile* component in 'SHSS' is used to send the notifications to an owner of the house, through data received from other components in 'SHSS'.

Figure 23 FT model – *Mobile*

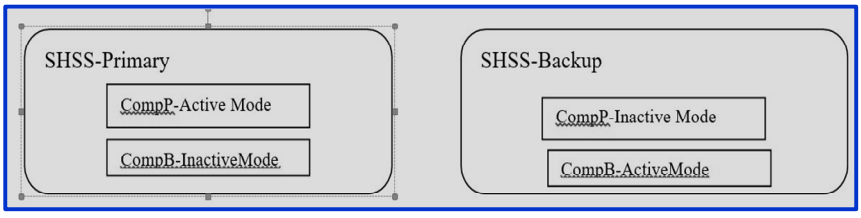


Note: FS: fail stop state.

5.2 *Designing fault-tolerant 'SHSS' through redundancy management scheme*

The purpose of designing fault-tolerant systems is that – it functions correctly even in presence of faults (Dubrova, 2013). One of the most common ways to achieve FT is by using redundancy. Redundancy provides functionalities that are not required in a fault-free environment. The redundancy allows either fault masking or fault detection, with the techniques such as location, containment, and recovery. For this, two modes, i.e., *active and inactive mode* have been taken as shown in Figure 24. The component's duplicate instance lies inside the system and mode parameter will determine which component is in (*active*) mode at a specific time. Mode transition occurs whenever there is a failure of active component. Reactivation through mode transition needs to be performed then only the failed component can resume its service (Feiler and Rugina, 2007). In this way, the backup will be designed only for critical components whose failure can affect an overall system or for a life-threatening event. For example – *SmokeDetector* and *FireAlarm* components in 'SHSS' are considered to be critical due to the critical functionalities they provide.

Figure 24 Creating primary and backup of critical components (see online version for colours)

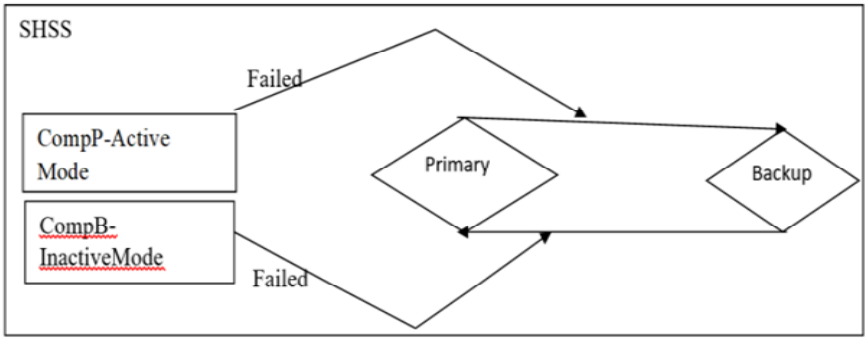


In modelling the FT system, the system is considered to be *failure recoverable*. Hence, in Figure 25, when the control transfers from *failed* state *to error-free state*, the recovery mechanism involved in case of failure, is the switching from primary component to backup component. The system can have multiple operational modes. An operational mode represents a particular configuration of a fault-tolerant system, such as operating the primary or backup of a redundant system. Figure 26 depicts an AADL architecture model (textual AADL) part and its associated error model. A *Guard_Event* property is linked with about event port involved.

ModeTransition error model is used in Figure 26 is written in detail in Figure 27; deactivation and activation of components is performed through mode transitions. The used error model declares an initial inactive state. It works on the assumption that an inactive component never fails. Whenever an error-free component is activated through a mode transition, it goes to an active *Error_Free* state and it can fail anytime during the time when the component is active. If the active failed component wants to regain its

Error_Free (active) state, it can be repaired. If a failed component is deactivated, it goes to an *Error_Free* (inactive) state.

Figure 25 Proposed recovery mechanism



In 'SHSS', the backups will be created only for critical components, i.e., *FireAlarm* and *SmokeDetector* whose failure can affect the service of the overall system or which are life-threatening events.

Figure 26 AADL code specification-recovery mechanism (see online version for colours)

```
system implementation SecuritySystem.personal
annex Error_Model {**
   Model => ModeTranition;

 Guard_Event => self[Failed] applies to failed;
**};
end SecuritySystem.personal;
system SHSS
features
Input: in data port;
Output: out data port;
end SHSS;
system implementation SHSS.general
subcomponents
CompP: system SecuritySystem.personal in modes Primary;
CompB: system SecuritySystem.personal in modes Backup;
connections
data port Input -> CompP.Input in modes Primary;
data port CompP.Output -> Output in modes Primary;
data port Input -> CompB.Input in modes Backup;
data port CompB.Output -> Output in modes Backup;
modes
Primary: initial mode;
Backup: mode;
Primary -[CompP.Failed]-> Backup;
Backup -[CompB.Failed]->Primary;
end SHSS.general;
```

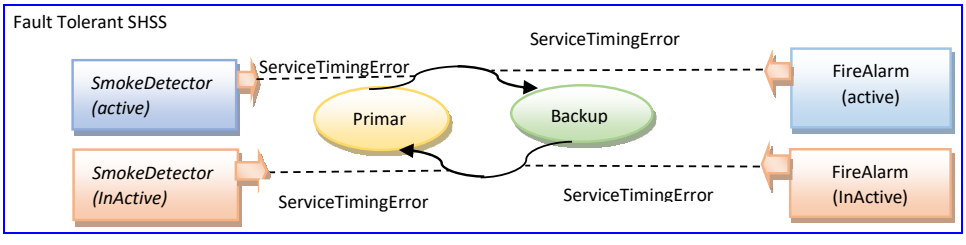
Figure 27 ModeTransition error model (see online version for colours)

```
error model ModeTransition
features
ON_ErrorFree: initial error state;
OFF_ErrorFree: initial inactive error state;
Failed: error state;
Fail: error event;
Repair: error event;
end ModeTransition;
error model implementation ModeTranition
transitions
ON_ErrorFree-[deactivate]->OFF_ErrorFree;
OFF_ErrorFree-[activate]->ON_ErrorFree;
ON_ErrorFree-[Fail]->Failed;
Failed-[out CorruptedData]->Failed;
Failed-[Repair]->ON_ErrorFree;
Failed-[deactivate]->OFF_ErrorFree;
end ModeTransition;
```

5.2.1 *Limitation of the baseline architecture, without the critical analysis component*

Whenever there is a failure of the primary devices, the control will directly transfer to the backup of these devices. Hence, there will not be any instance of the overall system getting crash and finally, the system will be called as 'fault-tolerant SHSS'. Figure 28 depicts redundant instances of critical components that lie inside 'SHSS' whose modes (primary and backup) decide which instance of component and connection is active. An event port for each of the redundant components of *SmokeDetector* and *FireAlarm* is used to report that it is in a failed state. At a time, only one component is in an active mode and only the connections to and from the active component is active in the same mode. However, if criticality analysis had not been performed in the SHSS – the complete SHSS cannot be made fault-tolerant because of redundancy management scheme. And, the crash of one complement can result in the failure of complete system.

Figure 28 Fault--tolerant 'SHSS' (see online version for colours)



6 Conclusions

In designing safety-critical systems, analysis at the architecture level is necessary to detect the occurrence of faults at the stage to stop the further propagation of faults in the

system. For this reason, there should be a clear understanding of the phase at which the fault is occurring. Failures are generally caused by faults in the components of the systems that may occur in system design or component integration. The software-supported fault-tolerance is relatively new. In this paper, an architectural level fault-tolerance policy has been proposed for an SOA-based system. The proposed policy covers most of the fault-tolerant mechanisms. The application of the proposed policy may enhance the reliability and availability of a system. It is a cost-effective solution that first identifies the severity level of services and suggests a suitable policy for that. The policy may be improved by obtaining feedback.

Acknowledgements

This paper and the research behind it would not have been possible without the exceptional support of my PhD supervisor, Dr. Ratneshwer. His enthusiasm, knowledge and exacting attention to detail have been an inspiration and kept my work on track from my first step to the final draft of this paper.

References

Bin Zheng, Z., Lyu, M.R.T. and Wang, H.M. (2015) 'Service fault tolerance for highly reliable service-oriented systems: an overview', *Science China Information Sciences*, Vol. 58, No. 5, pp.1–12, DOI: 10.1007/s11432-015-5313-y.

Buys, J., De Florio, V. and Blondia, C. (2011) 'Towards context-aware adaptive fault tolerance in SOA applications', *DEBS'11 – Proceedings of the 5th ACM International Conference on Distributed Event-Based Systems*, pp.63–74, DOI: 10.1145/2002259.2002271.

Carnegie Mellon University (no date) *Architecture Analysis and Design Language* [online] https://www.sei.cmu.edu/our-work/projects/display.cfm?customel_datapageid_4050=191439 (accessed 30 December 2020).

Delange, J. et al. (2014) *AADL Fault Modeling and Analysis Within an ARP4761 Safety Assessment (CMU/SEI-2014-TR-020)*, Software Engineering Institute, Carnegie Mellon University [online] http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=311884 (accessed 4 January 2023).

Dobson, G., Hall, S. and Sommerville, I. (2005) 'A container-based approach to fault tolerance in service-oriented architectures', *International Conference of Software Engineering (ICSE)* [online] http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:A+Container-Based+Approach+to+Fault+Tolerance+in+Service-Oriented+Architectures#0 (accessed 20 December 2020).

Download and Installation – OSATE 2.10.0 Documentation (no date) [online] https://osate.org/download-and-install.html (accessed 14 October 2021).

Dubrova, E. (2013) *Fault Tolerant Design: An Introduction*, pp.21–26, DOI: 978-1-4614-2112-2 [online] http://www.ece.nus.edu.sg (accessed 15 November 2020).

Feiler, P. and Rugina, A. (2007) *Dependability Modeling with the Architecture Analysis & Design Language (AADL) (CMU/SEI-2007-TN-043)*, Software Engineering Institute, Carnegie Mellon University [online] http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=8277 (accessed 4 January 2023).

Feiler, P.H., Gluch, D.P. and Hudak, J.J. (2006) *The Architecture Analysis & Design Language (AADL): An Introduction*, February, p.CMU/SEI-2006-TN-011 [online] http://www.sei.cmu.edu/library/abstracts/reports/06tn011.cfm (accessed 23 November 2020).

Fekih, H., Mtibaa, S. and Bouamama, S. (2019) 'The dynamic reconfiguration approach for fault-tolerance web service composition based on multi-level VCSOP', *Procedia Computer Science*, Vol. 159, pp.1527–1536, DOI: 10.1016/j.procs.2019.09.323.

Gabsi, W., Zalila, B. and Jmaiel, M. (2016) 'EMA2AOP: from the AADL error model annex to aspect language towards fault tolerant systems', *2016 IEEE/ACIS 14th International Conference on Software Engineering Research, Management and Applications, SERA 2016*, pp.155–162, DOI: 10.1109/SERA.2016.7516141.

GitHub (no date) *GitHub – mmanjun/AADLModel_HomeSecuritySystem: AADL is Architecture Analysis and Description Language, Designed and Developed by the SAE for Specification, Analysis, Automated Integration and Code Generation of Real-time Performance-critical (Timing, Saf)*.

Kushal, K.S., Nanda, M. and Jayanthi, J. (2017) 'Architecture level safety analyses for safety-critical systems', *Journal of Aeronautics & Aerospace Engineering*, Vol. 6, No. 1, pp.1–8, DOI: 10.4172/2168-9792.1000181.

Larson, B. et al. (2013) 'Illustrating the AADL error modeling annex (v.2) using a simple safety-critical medical device', *HILT 2013 – Proceedings of the ACM Conference on High Integrity Language Technology*, November, pp.65–83, DOI: 10.1145/2527269.2527271.

Lasnier, G. et al. (2011) 'An implementation of the behavior annex in the AADL-toolset OSATE2', *Proceedings – 2011 16th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2011*, May 2014, pp.332–337, DOI: 10.1109/ICECCS.2011.39.

Mahdian, F. et al. (2009) 'Modeling fault tolerant services in service-oriented architecture', *Proceedings – 2009 3rd IEEE International Symposium on Theoretical Aspects of Software Engineering, TASE 2009*, pp.319–320, DOI: 10.1109/TASE.2009.41.

Niknejad, N. et al. (2020) 'Understanding service-oriented architecture (SOA): a systematic literature review and directions for further investigation', *Information Systems*, Vol. 91, p.101491, DOI: 10.1016/j.is.2020.101491.

Procter, S. (no date) *The AADL Error Library: 4 Families of System Errors*, Software Engineering Institute [online] https://insights.sei.cmu.edu/sei_blog/2019/05/the-aadl-error-library-4-families-of-system-errors.html (accessed 15 June 2020).

Qiu, W. et al. (2014) 'Reliability-based design optimization for cloud migration', *IEEE Transactions on Services Computing*, Vol. 7, No. 2, pp.223–236, DOI: 10.1109/TSC.2013.38.

SAE, Annex X Behavior Annex (AS5506-X Draft-2.13) (no date) [online] https://www.sae.org/standards/content/as5506/2/ (accessed 27 August 2020).

Sokolsky, O. and Chernoguzov, A. (2014) *Analysis of AADL Models Using Real-Time Calculus With Applications to Wireless Architectures*, July 2008.

Sun, H., Hauptman, M. and Lutz, R. (2007) 'Integrating product-line fault tree analysis into AADL models', *Proceedings of IEEE International Symposium on High Assurance Systems Engineering*, DOI: 10.1109/HASE.2007.46.

Zhang, P., Shu, S. and Zhou, M. (2018) 'An online fault detection model and strategies based on SVM-grid in clouds', *IEEE/CAA Journal of Automatica Sinica*, Vol. 5, No. 2, pp.445–456, DOI: 10.1109/JAS.2017.7510817.

Zhang, P.Y. et al. (2021) 'A fault-tolerant model for performance optimization of a fog computing system', *IEEE Internet of Things Journal*, Vol. 4662, No. c, pp.1–1, DOI: 10.1109/jiot.2021.3088417.

Zheng, Z. et al. (2012) 'Component ranking for fault-tolerant cloud applications', *IEEE Transactions on Services Computing*, Vol. 5, No. 4, pp.540–550, DOI: 10.1109/TSC.2011.42.