Available online at www.ijpe-online.com

International Journal of Performability Engineering

Vol. 13, No. 6, October 2017, pp. 844-853
DOI: 10.23940/ijpe.17.06.p6.844853

An Analysis Tool Towards Fault Tolerance Systems based on AADL Error Model

Wenbing Zhang[a], Guohua Shen[a,*], Zhiqiu Huang[a], Zhibin Yang[a], Lei Xue[b]

[a]*Nanjing University of Aeronautics and Astronautics, No.29 Jiangjun Road, Nanjing and 211106, China*
[b]*Shanghai Aerospace Electronic Technology Institute, Shanghai 201109, China*

Abstract

Fault-tolerant embedded systems can provide the correct service with the active faults. It is important to verify the ability of fault tolerance in system design phase especially for safety-critical systems. Besides, the verification of models can reduce the cost of the system development. The Architecture Analysis and Design Language (AADL) and its Error Model Annex (EMV2) provide the ability to model a fault tolerance system. The error event, error propagation and error state machine provided by the AADL Error Model Annex can model an embedded system. However, there is a problem that whether the model satisfies the requirement of fault tolerance or not. We design a component-based algorithm to verify the ability of fault tolerance. The error and warning messages will be produced by our algorithm. Finally, a plugin based on the Osate2 tool and a case study are given.

Keywords: fault tolerance; AADL; error model annex; state machine; model analysis

(Submitted on July 25, 2017; Revised on August 30, 2017; Accepted on September 15, 2017)

(This paper was presented at the Third International Symposium on System and Software Reliability.)

1. Introduction

In the secure-critical fields like avionics, nuclear engineering, aircraft and transportation, it highly requires reliability to reduce the occurrence of failures. Within the area of system reliability, the fault tolerance is an effective and dependable method to enable the system (mostly embedded system) to still work in the correct way.

There are mainly two categories of the strategies implementing the fault tolerance: recovery and redundancy. The recovery transforms the system into an error-free state. The redundancy requires the additional resources like hardware, software, data and time to tolerant the fault. In each category, more specific and different sub-categories can be applied in the different application of the secure-critical system.

But with the growth of the scale and the complexity of the interaction of the embedded systems, the number of the components reaches an amazing level. It is challenging for developing such a large-scale embedded system meanwhile the system requires the high reliability.

The MBE (Model-Based Engineering) has been proposed as an approach that can tackle the challenges of developing robust and large-scale embedded systems. The ADLs (Architecture Design Models) are the center in the MBE, which can model the function and non-function behavior of real-world systems. More important, safety and security analysis can be done with the ADLs. The Architecture Analysis & Design Language (AADL) is one of ADLs aiming at modeling the architecture of embedded system include software and hardware part. The AADL allows designers to extend the AADL with various annexes. To be able to support fault modeling, the AADL Error Model Annex is proposed. This annex adds the

* Corresponding author.
E-mail address: ghshen@nuaa.edu.cn.

specification of the error event, error propagation and error type and error behavior state machine into the AADL component which make it possible to model the fault-tolerant embedded system.

There are some researches about how to model the fault-tolerant system using AADL. For the Error Model, some analysis tools such as FTA, FMEA and Consistency Checker have been developed. We will talk about these related work in the next section. But we notice that there are no tools to check the fault tolerance of the AADL model. In other words, how to verify that the AADL model with the error annex towards the fault-tolerant systems satisfies the requirement of the fault tolerance.

Our work is mainly to verify the error model that with any error event, the control component in the AADL will not migrate into a not-working state. If the component moves to a not-working state with an error event in the component or its sub-component, it means the component is not tolerable for a fault. Our work can help system designer to find the design problem in the model level to avoid the costly rework.

The paper is structured as follow. First, we talk about the background of our work. The background contains the concepts of the fault tolerance, AADL model and error annex. The related work is introduced in each part of the background. After that, we propose our algorithm to verify the EMV2-based model to satisfy the requirement of fault tolerance system. The next section demonstrates the approach using a DualGPS case. In the last section, a conclusion is drawn.

2. Background and related work

We show some simple concepts of the fault tolerance system and introduce some mechanisms to implement the fault tolerance in this section. Then a short overview of the AADL version 2.0 and its Error Model Annex V2 is presented after the fault tolerance.

2.1. Concepts of the Fault Tolerance

Before we introduce the concepts of the fault tolerance, we firstly show some different definitions of the fault tolerance. The definition in [5] is that it is the property that a system can continue operate properly after the failure events happened in its subcomponents. In [3], a fault-tolerant system can still deliver the correct service in the activation of faults. Reference [11] thinks of the fault tolerance as a method to achieve a dependable computing system by providing service with the specification even with faults happening or happened. These definitions mainly describe such a system that can still work rather than crush down facing with the error event. And we mainly focus on the embedded system and its reliability. The embedded system is commonly based on the module. Each module has its specific function. So, we give our definition of the fault-tolerant embedded system as follow: a fault-tolerant embedded system is one kind of embedded system that can still work functionally in the presence of active fault event.

After giving our definition of the fault-tolerant embedded system, we show some basic concepts in the following paragraphs. It can help us in the modeling phase and we will discuss it in the next section.

As mentioned in [11], error processing is the key of the implementation of the fault-tolerant system. There are two constituent phases in the field of the error processing: effective error processing and latent error processing. For each category, some theories are put up and implemented in the industry.

The most common method in the effective error processing is error recovery which transforms a system from the erroneous state into an error-free state. According to the direction of the transformation, it may make on two forms:

- Backward error recovery, where the system is transformed back to a saved state. In [12], the authors describe the detail of implementation of user-level checkpoint and migration service for UNIX processes. Reference [15] give a genetic algorithm using check point in the grid computing environment. The advantage of backward recovery is that it can handle unknown errors.
- Forward error recovery, as known as roll-forward recovery where recover the system with the transformation to a new state. It is more efficient that the backward recovery. In real-time systems with high requirement of the time constraints, forward recovery is primarily used. Some strategies have been proposed in [23,25,26].

Error compensation is another form of effective error processing. It eliminates the error by providing redundancy. It is commonly used in the field of machine tool as showed in [16,17]. In the embedded system, error compensation is based on error masking.

In term of latent error processing, error passive and reconfiguration are two methods. It is a challenge to detect the latent error and process it properly with low cost. An important role in the fault tolerance is redundancy which is a strategy for error masking. Redundancy requires additional resources to make sure that an error-free service will be delivered in any case. In term of resource types, redundancy can be divided into several categories: hardware redundancy, software redundancy, information redundancy and time redundancy. These strategies can be applied in one system simultaneously.

Hardware redundancy is perhaps the most common use of redundancy which includes replicated and supplementary hardware. The redundant hardware with the software resides on the hardware is able to tolerate both hardware and software faults. Software redundancy are the concept in the software level, such like additional programs, modules or objects. Information redundancy (data redundancy) is sometimes one kind of software redundancy. But we treat it as a new category here. It is a redundant method to ensure the safety of the critical data. Time redundancy is performing the same operation multiple times with the additional time.

These aspects of redundancy are widely used in the safety-critical fields. With model-based engineering arising, the work of modeling and verification towards the fault-tolerant system is carried out. For example, the state machine is used to implement the fault-tolerant services in [22]. A programming model is proposed to provide strong consistency and efficient fault recovery in the application of the stream processing on large clusters in [27]. Among these models targeting the embedded system, there is one category called ADL (Architecture Design Model), which has the advantage of describing both of hardware and software components. The ADLs are also adaptive for the study of the fault-tolerant systems. AADL is one of the ADLs having the ability of analysis and great extensibility. There have been some study about how to model the fault-tolerant and analysis based on the AADL. For example, the authors build a fault-tolerant AADL model with EMV2 annex for an UAV system in [18]. Reference [4] shows some detailed AADL example model of the redundant flight guidance systems (FGS) using AADL Error Model. Reference [7] has developed a code generation tool which can generate aspect-oriented code from the AADL Error Model towards fault-tolerant systems.

But the work based on the AADL and other annexes mostly focuses on how to model the fault-tolerance system. Although, there are some tools to analysis the model's dependability in [19], the quantitative risk in [13]. There is a lack of tools to analysis the model's fault tolerance attribute based on the AADL model. So we put up an algorithm to check the AADL model whether it is fault-tolerant and develop a plugin integrated in the Osate2 which is an open-source platform for AADL. Before we demonstrate our approach, we will shortly introduce the AADL and its Error Model Annex which is capable to model the fault-tolerant embedded systems.

2.2. Overview of the AADL Language

The Architecture Analysis & Design Language (AADL) [21], as the name suggests, supports analysis and design for systems. It is derived from the MetaH, which was developed by the Advanced Technology Center of Honywell. MetaH is also an architecture description language having the syntactical roots in Ada. So the style of syntax of the AADL is similar to Ada. The AADL has its emphasis on the embedded domain, having the elements being abstract for the hardware and software. Several AADL-related tools and platforms have been developed by communities to support verification, analyzing and code generation of the AADL. The list of tools can be found in [1].

Not like the Unified Modeling Language, the most elements in the AADL have the well-defined semantic. The AADL standard provides three categories of the elements abstracting the components in the embedded system.

- Software components: The software components are the abstraction of the software part of the embedded system, like process, thread, thread group, data and subprogram.
- Hardware components: The hardware components are used to model the architecture and the connection of the hardware part of the embedded system. They are also the execution platform of the software. The AADL provides the specification of binding to bind a process to a specific processor. There are 4 types of the hardware component: processor, memory, bus and device.
- System components: The system component is a hybrid component representing the whole system or a subsystem. It only has one type component, system. The system component is commonly used to glue software components and hardware components together.

Components in AADL can be specified with the properties. Besides the standard property library provided by the committee, the designers can also provide their own property library by the extension mechanism. The AADL is capable for modeling the complex embedded system. We only present an overview. More detail of AADL can be found in [2].

2.3. Overview of the Error Model Annex

The Error Model Annex is one of the annexes for the AADL. The mechanism of the annex enhances the ability of describing and analyzing of the AADL. The Error Model Annex version 2 (EMV2) is proposed in 2013 to enhance the ability in fault modeling. It is described in documents [6,20].

The EMV2 can model the fault in the system into three levels of abstraction:

- Component Error Behavior: Each component can specify one or more events of error, recovery and repair in the annex sub-clause. These events can trigger the transitions in the state machine which is contained in the same component.
- Composite Error Behavior: With the change of subcomponent error states the parent component may be into a new error state. This is called compositional error behavior. In the secure-critical system, the error behavior of some sensors which are applied triplication strategy with voting is compositional. The annex provides serval state composite logic operations: *or*, *and*, *ormore*, *orless*. The operation *or* means that the expression will be true if one of the operators is true. The operation *and* means that the expression will be true if both of two operators are true. The expression with *ormore* will be true only if at least n operators are true. The operation *orless* is the opposite of the *ormore*. The expression with the *orless* will be false if more than n operators are true.
- Error Propagation: Error propagation is used to model the impact of the typed fault on components. Outgoing and incoming error propagation of one error type can be specified on the port or binding feature. Path propagation can be specified on an in port and an out port. In term of the path of error propagation, it is determined by component connections and deployment bindings which are specified in the component's features rather than in annex clauses.

There are a lot research focusing on the safety and security analysis based on the AADL and Error Model. For example, there are FTA analysis in [10, 24], FMEA analysis in [8, 9], dependability analysis in [18]. But there is no research towards analyzing the fault tolerance of the AADL and Error Model. After having built the model towards a fault-tolerant system, knowing how the model is fault-tolerant is a problem. The wrong design in model level will lead the wrong system. So we develop a plugin integrated in the Osate2 to check the fault tolerance of the model. We will demonstrate our algorithm and our plugin in the next section.

3. The approach

To check the fault tolerance of the model, we first traversal the AADL model to find the components which have the composite behavior. And then take these components as our input, respectively. For each component the error message about the path of transitions with error event will be printed. We demonstrate our approach from three aspects.

3.1. The component error behavior

In the real embedded system, the fault happens in the component will affect other components or its parent component. Our approach can check the state of the target component with any error event within the subcomponent. Therefore, we firstly study how the component behaves with an error event inside it.

In the error model, the error event triggers transition to a state. In our approach, we firstly assume a single error event with a specific type happened inside one of the subcomponents. It is noticed that in the Osate2 tool the behavior events in a component error behavior can be returned through the method *ComponentErrorBehavior.getEvents()*. The error event is a type of error behavior event and it can generate events of different types. We enumerate every type of every error event in every turn to check the state of the control component. After choosing an error event, we iterate all the transitions whose source state is marked with the "*Statekind: Working*" to check whether the condition comes true with the error event. These states which are reached with the error event will be marked with "*readed*" tag and then added to a stack. After the states are added to the stack, we will execute the transition whose condition is empty and source state is the state taken from the stack. Meanwhile, the state machine for the out propagations will be checked because the state trigger by the error event can trigger the out propagation to another component.

Besides, the component where the error event occurs, we take the components which receive the out propagation into consideration. As the propagation between two components are determined by the AADL architecture, we use the connection through the ports to find out the component we want to check. What we do is to iterate all the connection and then match the type of the ports and the type of propagations specified the ports. For these components which receive the out propagation, we need to execute the transitions with the incoming propagation. The process is similar with the process of the transition with the error event because the incoming propagation can be the condition in the transition too. Also some out propagations may be propagated to other components. We use the breadth-first algorithm to visit the components with the help of a stack. To avoid the endless loop, the visited incoming propagations will be marked. We do not mark the component with tag since one component can have more than one incoming propagations. The process is in Algorithm 1.

Algorithm 1 The algorithm of Component Error Behavior

Input: ComponentInstance component
Output: The queue states

```
1.  // There is only one trigger in one time
2.  trigger = component.getTrigger()
3.  Queue states
4.  Queue q
5.  States.put(component.getInitState())
6.  // using the depth-first algorithm
7.  While states.empty() == False:
8.      state = states.pop()
9.      for each transition in component.getTransition(state)
10.         // if the condition is null or the condition can be triggered by the trigger already save
11.         if Trigger(transition.getCondtion(), trigger) == True:
12.             state ts = transition.getTargetState()
13.             if ts.isVisit() == False:
14.                 states.put(ts)
15.                 // We deal with the outpropgation triggered by the state ts,
16.                 // all the components recivce the propagation will be put in the Queue q
17.                 DealwithOutPropagation(component, ts, q)
18.
19. // using the procedure to see whether the component has the mechanism and what the states the component can recover
    to. For the reason of page, we won't put up the detailed algorithm
20. Recover(component, states)
21. Return states
```

Algorithm 2 The algorithm of main component checker

Input: ComponentInstance parent
Output: The warning and error messages

```
1.  List warningMessages
2.  List errorMessages
3.  For subcomponent in parent.getAllSubcomponent()
4.      for ev in subcomponent.getAllErrorEvent()
5.          parent.resetState() // reset each component to its original state
6.          Queue q
7.          subcomponent.setTrigger(ev)
8.          q.push(subcomponent)
9.          while(q.empty() == False)
10.             subcomponent = q.pop()
11.             if subcomponent.visitWithTrigger() == False
12.                 componentErrorBehavior(subcomponent, q)
13.
14.              compositeErrorBehavior(parent, warningMessages, errorMessages)
15.
16. print warningMessages
17. print errorMessages
```

3.2. The composite error behavior

The compositional error behavior specification contains a mapping relation from error states of sub-components into system-level error states. Therefore, the compositional error behavior state machine can be affected by the state in sub-components. Usually the composite error behavior can be used to model the fault tolerance strategies towards the fault tolerance system. What we need to do is to verify the target state in the composite error behavior state machine.

First we visit all the transitions in the error behavior state machine. For each transition, we firstly filter the transitions whose source state is not working. Then the conditions of the transitions remained will be checked. This is a complex process. There are 7 types of condition expressions: *AndExpression*, *CondtionElement*, *OrExpression*, *OrlessExpression*, *OrlessExpression*, *SAndExpression* and *SOrExpression*. The structure of the whole condition is a tree with the concrete condition expression. For example, the whole condition may be an *AndExpression*. It may contain an *OrExpression* and a *ConditionElement*. We implement a method for each type of the condition expressions to return the Boolean value indicating the condition comes true or false. The method for *ConditionElement* is more complex than others. The reason is that the *ConditionElement* is supported to contain the error event, the propagation or the sub-component state. When encountered with the *ConditionElement*, we need to visit the corresponding elements. As the value of the whole condition, we call the value method of the top condition expression. The value method will be executed in a top-down way and the logic value will be returned in a down-top way.

After checking the condition in the transitions, we can get all the target states in the successful transitions. If one target state represents the work state, the component can still work even with the error event. We will produce a warning because this is an unusual behavior for a fault tolerance design; we will produce an error when the states in not-working exist in the target state set. In this situation, the component moves into a not-working state with the error event. So the component is not fault tolerable. But there are exceptions if the repair or the recovery strategy is set. We will demonstrate the situation in the next section. And the algorithm of the composite error behavior is similar to the component error behavior.

3.3. The recovery event

The mechanism of recovery can recover the system from erroneous state when error events are detected. The error model provides the recover event to represent the occurrence of the recovery process. Many fault tolerance systems have the recovery design. Therefore, our algorithm takes the recover event into account.

In the phase of the component error behavior, the recover event can be triggered by the component itself when it moves into a specific state. We need to add the step of detecting the recover event and get the target states from the states which migrate with the error event. Then we read the "*StateKind*" property to check whether it is a working state. An error will be produced if the target state is not-working.

Besides the recover event triggered by the component itself, it also can be triggered by the incoming event port. The control component may send the recover event to other components through the ports between components. In the composite error behavior, the component may move into some error states and then send the recover event through the specific port. We add the step to check the state of the control component and other sub-components. An error will be produced when any component is still in not-working state.

We implemented our approach with the help of the Osate2 [14]. We present the framework of our algorithm in the Algorithm 2.

Because our approach is component-based, we take those components with the composite error behavior as our input. Then we take every error event as a turn. In one turn, we save the states of each sub-component reached with the error event. Then we use the procedure *compositeErrorBehavior* to check the state of the parent component. In the procedure *compositeErrorBehavior*, the recover event is checked for the recovery mechanism of the fault tolerance. All the error messages and warning messages will be printed.

4. CASE STUDY

So far, we have constructed our approach verifying the ability of the system's fault tolerance and an algorithm has been given in Sect. 4. In this section, we show an overview of the development process of our Osate2 plugin and present a model of the DualGPS as a study case to demonstrate our approach.

4.1. FT Eclipse plugin

Our plugin is based on Osate2 [21] which is an official platform in the field of AADL. It enables modeler to build the AADL model graphically. The advantage of Osage2 is its capacity of analysis and expandability. Some development library written by JAVA are available for developers. The process of the development are demonstrated as 3 steps.

The first step: new a plugin project

The second step: traverse the instance of AADL component.

We traverse all the components in the system instance with the help of the package org.osate.aadl2.modelsupport. modeltraversal provided by the Osate2. The package contains various methods of traversing the AADL instance, including post-order and pre-order traverse.

The third step: We get the error model sub-clause from in the component instance by using the classes in the package *org.osate.xtext.aadl2.errormodel.errorModel*. The concepts of the error model are represented as the Java interfaces. And each interface has an implementation, correspondingly. Then we use the algorithm to verify the whole model.

The fourth step: We print the error massages to a report stored in a file. A message contains the name of the component, the name of error event, the content of the transition.

The tool can be run by right-clicking the instance as showed in Figure 1. The file circled by the red is automatically produced by our tool.

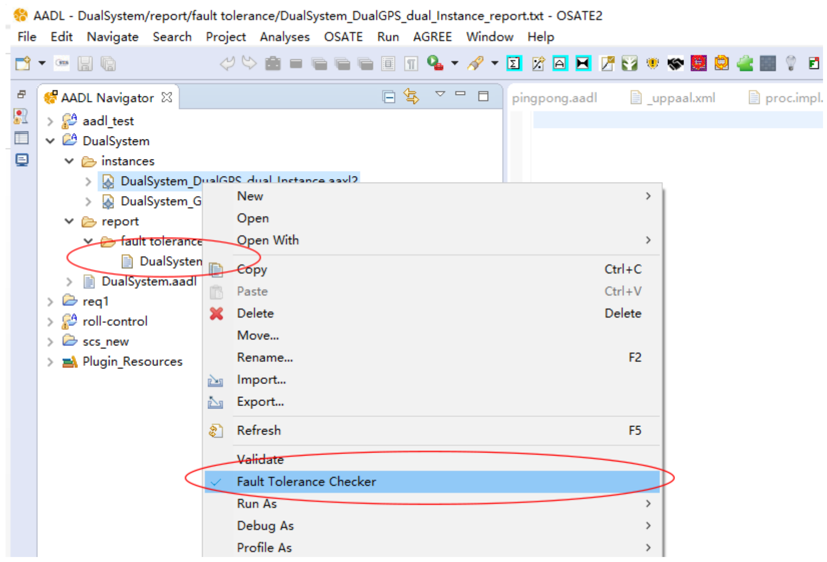


Figure 1. The screenshot of our plugin

4.2. DualGPS Example

We use the DualGPS model as our target model to be checked because it is a common module but critical-security in the field of the aviation, self-driving car and so on. This example contains a fault tolerance strategy named redundancy. We firstly use the correct design as the input of our plugin. Then we remove the part of the redundancy to check whether our tool produces the proper report.

As the Figure 2 shows, the system DualGPS contains three subcomponents, two devices and a vote system. The two devices in the Figure 3(b) are instances of the specification of the device GPS. The GPS device uses the FailStop error behavior from the Error Library. The FailStop error behavior consists of two states: Operational and FailStop. We add “StateKind: working” and “StaeKind: not-working” to the Operational and FailStop, respectively. The transition from the Operational to the FailStop can be triggered by the error event named Fault. After the FailStop were reached, an error propagation will be propagated to the vote system through the connection.

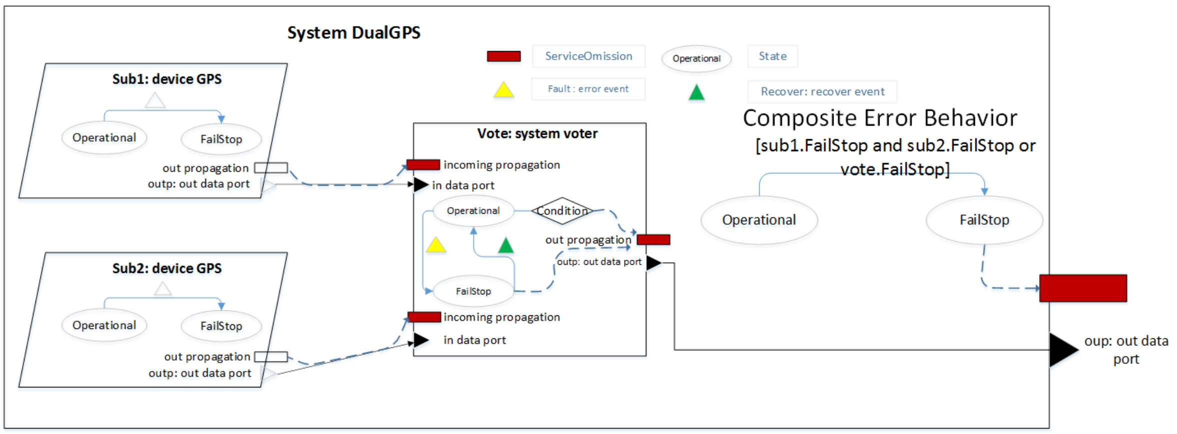
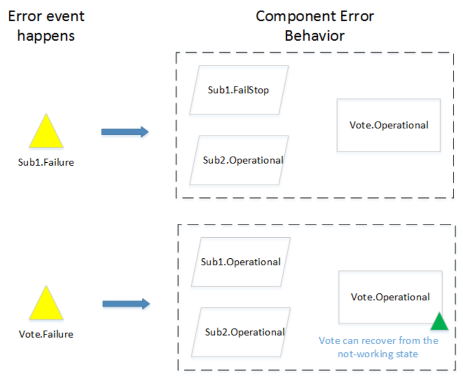


Figure 2. The graphic view of AADL model of DualGPS system

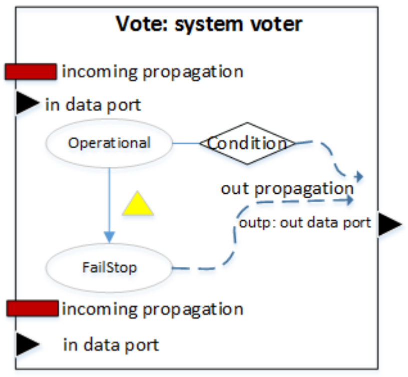
The Vote system is also a sub-component of the DualGPS system. It receives the input data from two GPS devices and send to the DualGPS with the data voted by a strategy. The fault may be produced by the Vote system itself or be propagated from one of the GPS device.

The DualSystem contains a composite error behavior mapping the states of sub-components to the states of itself. The change of the states of the GPS device will affect the state of the DualSystem.

Table 1. shows the result of running the fault tolerance checker for the DualSystem. There are three error events in the result table. There is no error message in the category of each error event. That means that whichever error event happens, the DualSystem won't move into error state. In others words, the DualSystem is a fault tolerance model with high safety.



(a)



(b)

Figure 3. (a) The states of each component with the event (b) The graphic view of not fault-tolerant model

Then we make a little change for the DualSystem. We remove the recover transition in the Vote system. The structure of the Vote system is presented in the Figure 3(b). Then we run our plugin again. The result is showed in the Table. 2. An error message is produced that the error event of the Vote system causes the fail of the DualSystem.

Table 1. The result of Fault-tolerant model

Error Event	***Control Component***	***Transition into not-working state***
Sub1.Failure	DualGPS	None
Sub2.Failure	DualGPS	None
Vote.Failure	DualGPS	None

Table 2. The result of unfaulty-tolerant model

Error Event	***Control Component***	***Transition into not-working state***
Sub1.Failure	DualGPS	None
Sub2.Failure	DualGPS	None
Vote.Failure	DualGPS	[sub1.FailStop and sub2.FailStop or vote.FailStop]->FailStop;

5. Conclusions

In this paper, we put up an algorithm and develop a tool to verify whether the AADL model with EMV2 annex satisfies the requirement of fault tolerance. The proposed method can help engineers to find the irrationality of the system design in the model level. We development a plugin in Osate2 to run our algorithm on the instance of AADL model. The algorithm can check the state of the control component under every event happened inside. The algorithm will print the error messages which contain the error events and the transitions. Our method takes the state machines, error propagations and error events into consideration and prints the transitions that do not satisfied with the requirement of the fault tolerance. In the future, we consider combining the AADL Behavior Model with the error model to make a more specific analysis towards the fault tolerance system to check the correctness of the design of the model.

Acknowledgements

This work was supported by the graduate student innovation base (laboratory) open fund project of Nanjing University of Aeronautics and Astronautics, kfjj20171606 and the National High Technology Research and Development Program (863 Program) of China (2015AA105303), the National Defense Basic Scientific Research Project under Grant of China (JCKY2016203B011), the National Natural Science Foundation of China (61502231), Natural Science Foundation of Jiangsu Province (BK20150753), the Project of the State Key Laboratory of Software Development Environment of China (SKLSDE-2015KF-04), the Avionics Science Foundation of China (2015ZC52027).

References

1. “AADL_tools,” *AadlWiki*, Available at https://wiki.sei.cmu.edu/aadl/index.php/AADL_tools, Last accessed on Sept 22, 2017
2. “AADL Wiki,” *AadlWiki*, Available at https://wiki.sei.cmu.edu/aadl/index.php/Main_Page, Last accessed on Sept 22, 2017
3. A. Avizienis, J. Laprie and B. Randell, “Fundamental Concepts of Dependability” University of Newcastle upon Tyne, Computing Science, 2001.
4. J. Delange and P. Feiler, “Architecture Fault Modeling with The AADL Error-model Annex,” in *Software Engineering and Advanced Applications* (SEAA), 2014 40th EUROMICRO Conference on, 2014, pp. 361-368.
5. “Fault_tolerance,” *Wikipedia*, Available at https://en.wikipedia.org/wiki/Fault_tolerance, Last accessed on Sept 22, 2017
6. P. Feiler. Architecture Analysis and Design Language (AADL) Annex Volume 3: Annex E: Error Model V2 Annex. Number SAE AS5506/3 (Draft) in SAE Aerospace Standard. SAE International, 2013.
7. W. Gabsi, B. Zalila and M. Jmaiel, “EMA2AOP: From the AADL Error Model Annex to Aspect Language towards Fault Tolerant Systems,” in *Proceedings of the 14th International Conference on Software Engineering Research, Management and Applications* (SERA), pp. 155-162, 2016.
8. M. Hecht, A. Lam, R. Howes, and C. Vogl, “Automated Generation of Failure Modes and Effects Analyses from AADL Architectural and Error Models,” AEROSPACE CORP EL SEGUNDO CA 2010.
9. M. Hecht, A. Lam, C. Vogl, and C. Dimpfl, “A Tool Set for Generation of Failure Modes and Effects Analyses from AADL Models,” in Presentation at Systems and Software Technology Conference, 2012.
10. A. Joshi, S. Vestal and P. Binns, “Automatic Generation of Static Fault Trees from AADL Models,” in *DSN Workshop on Architecting Dependable Systems*, 2007.
11. J. Laprie, “Dependable Computing and Fault-tolerance,” Digest of Papers FTCS-15, pp. 2-11, 1985.
12. M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, “Checkpoint and Migration of UNIX Processes in The Condor Distributed Processing System,” *Technical Report 1997.*
13. Y. Liu, G. Shen, Z. Huang, and Z. Yang, “Quantitative Risk Analysis of Safety–critical Embedded Systems,” *Software Quality Journal*, vol. 40, pp. 1-25, 2016.
14. “Osate 2,” *AadlWiki*, Available at https://wiki.sei.cmu.edu/aadl/index.php/Osate_2#Introduction, Last accessed on Sept 22, 2017
15. S. B. Priya, M. Prakash and K. K. Dhawan, “Fault Tolerance-genetic Algorithm for Grid Task Scheduling Using Check Point,” in *Proceedings of the Sixth International Conference on Grid and Cooperative Computing*, pp. 676-680, 2007.

16. R. Ramesh, M. A. Mannan and A. N. Poo, "Error Compensation in Machine Tools—A Review: Part I: Geometric, Cutting-force Induced and Fixture-dependent Errors," *International Journal of Machine Tools and Manufacture*, vol. 40, pp. 1235-1256, 2000.
17. R. Ramesh, M. A. Mannan and A. N. Poo, "Error Compensation in Machine Tools—A Review: Part II: Thermal Errors," *International Journal of Machine Tools and Manufacture*, vol. 40, pp. 1257-1284, 2000.
18. H. Reza, R. Marsh and M. Askelson, "A Fault Tolerant Architecture Using AADLs and Error Model Annex for Unmanned Aircraft Systems (UAS)," *Software Engineering Research and Practice*, pp. 180-184, 2010.
19. A. Rugina, K. Kanoun and M. Kaâniche, "A System Dependability Modeling Framework Using AADL and GSPNs," *Architecting Dependable Systems IV*, pp. 14-38, 2007.
20. SAE International, AADL Error Model Annex, (Standards Document AS5506/1, 2006., 2006
21. SAE International, AS5506 – Architecture Analysis and Design Language (AADL), 2012
22. F. B. Schneider, "Implementing Fault-tolerant Services Using The State Machine Approach: A Tutorial," *ACM Computing Surveys* (CSUR), vol. 22, pp. 299-319, 1990.
23. R. E Smith, Richard E., and Maria Gini. "Reliable Real-time Robot Operation Employing Intelligent Forward Recovery," *Journal of Field Robotics 3.3* (1986): 281-300.
24. H. Sun, M. Hauptman and R. Lutz, "Integrating Product-line Fault Tree Analysis into AADL Models," in High Assurance Systems Engineering Symposium, 2007. HASE'07. 10th IEEE, 2007, pp. 15-22.
25. J. Xu, B. Randell, "Roll-forward Error Recovery in Embedded Real-time Systems," *Parallel and Distributed Systems*, 1996. Proceedings., 1996 International Conference on. IEEE, 1996.
26. J. Xu, B. Randell and A. Romanovsky, "Fault Tolerance in Concurrent Object-oriented Software through Coordinated Error Recovery," Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on. IEEE, 1995.
27. M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters," HotCloud, vol. 12, p. 10-10, 2012.