Available online at www.ijpe-online.com

International Journal of Performability Engineering

vol. 13, no. 7, November 2017, pp. 1057-1062
DOI: 10.23940/ijpe.17.07.p7.10571062

A Research for Aerospace Complex Software System Runtime Fault Detection

Chenjing Yan[*], Wei Zhang, Xiaochuan Jing, Hui Ge, Xiaoyin Wang

China Aerospace Academy of Systems Science and Engineering, Beijing and 100000, China

Abstract

Aerospace complex software system is the keypoint of aerospace industry informatization. The complexity and scale of aerospace complex software system is growing with the increase of system requirements. Therefore, the possibility of runtime failures is also increasing. The runtime failures may lead to some serious problems of the aerospace software system and may cause great damage. To reduce the loss of software failures and to ensure the normal operation of aerospace complex software system, this paper focuses on runtime fault detection based on runtime verification. Runtime verification aims to monitor a running system and check whether executions of the monitored system satisfies or violates a given correctness property. This paper proposes a method to realize runtime fault detection and solve the runtime failure problem.

Keywords: aerospace complex software system; runtime fault detection; runtime verification; fault detection process; runtime failures

(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

Nowadays, the aerospace industry is in a period of rapid development and the aerospace mission is becoming more and more intensive. The high density of space mission has raised higher requirements for the development of efficiency and quality of software system. On the other hand, aerospace mission involves lots of tasks and applications. As the system scale and intelligent degree increasing, a tendency that software systems are becoming intensive with high complexity has been showed. Because the complexity of aerospace software requirements increases, the complexity of software itself increases, which leads to the possibility of runtime failures of the aerospace software system. Once the core aerospace software system fails, it will cause great loss to the national economy, people's safety and even national security. Hence, it is very important and urgent to realize fault localization and self-diagnosis of aerospace software.

At present, scholars have developed some methods of software system automatic fault detection, which are divided into static methods and dynamic methods. Static methods detect the possible fault in the target program by analyzing the dependency, type constraints and other information [6]. While dynamic methods detect the program faults by testing system program, tracking program execution traces and tracking program coverage information. The efficient detection of software faults provides the prerequisite and guarantees the self-repair and self-adaptation of software. These methods have solved the fault detection problem in some ways, but the effects are not very satisfactory. What's more, the existing methods rarely consider software runtime state information when detecting faults. As a result, how to detect the faults of aerospace software based on system's runtime state and information has become a research hotspot in recent years. This paper focuses on overcoming current fault detection technical deficiencies and proposing an aerospace software system faults detecting method based on runtime verification. Runtime verification is the supplement of traditional program correctness assurance technology, such as model checking and testing. Model checking and testing focuses on verifying the correctness of all

* Corresponding author.
E-mail address: 18813095861@163.com.

execution paths in a system, while runtime verification pays attention to current execution path. Based on these advantages, we introduce runtime verification to our proposed method.

2. Runtime verification

Verification contains lots of techniques; all these techniques can be used to judge whether a system satisfies its specification. Runtime verification is an emerging lightweight program verification technique. During runtime verification, the monitor is generated from the system requirements [11]. In runtime verification, a correctness property φ is typically automatically translated into a monitor. The monitor is used to check whether a runtime execution or a set of execution records can satisfy the property φ. For method based on runtime execution, we call it online monitoring and for method based on execution records, we call it offline monitoring. After defining a correctness property φ, $L(\varphi)$ is introduced which is a set of effective executions given by property φ. Then the runtime verification can be summarized as a method to check whether the execution ω is an element of $L(\varphi)$. By analyzing the definition and introduction of runtime verification, we can conclude that runtime verification deals with the word problem in mathematical justification, i.e., the target problem whether a given word is included in some language [3].

Monitor checks whether the operation process meets the demand of system by monitoring and observing program execution. Runtime verification is the effective supplement to the traditional software verification and validation technology, such as model checking and testing. Runtime verification deals with those verification techniques that allow checking whether an execution of a system under monitored satisfies or violates a given correctness property. It can not only effectively detect abnormal behavior in system operation, but also make it possible to effectively repair the system when the correctness deviation is detected.

Runtime verification technique combines the formal verification techniques with the actual running results of monitored system. Runtime verification monitors the operating behavior of the system to ensure that the system runs in line with the user's requirements. The technical framework of runtime verification is shown in Figure 1. During runtime verification, the monitor is generated from the system requirements. The monitor checks whether the operating process meets the demand of a system by observing program execution.

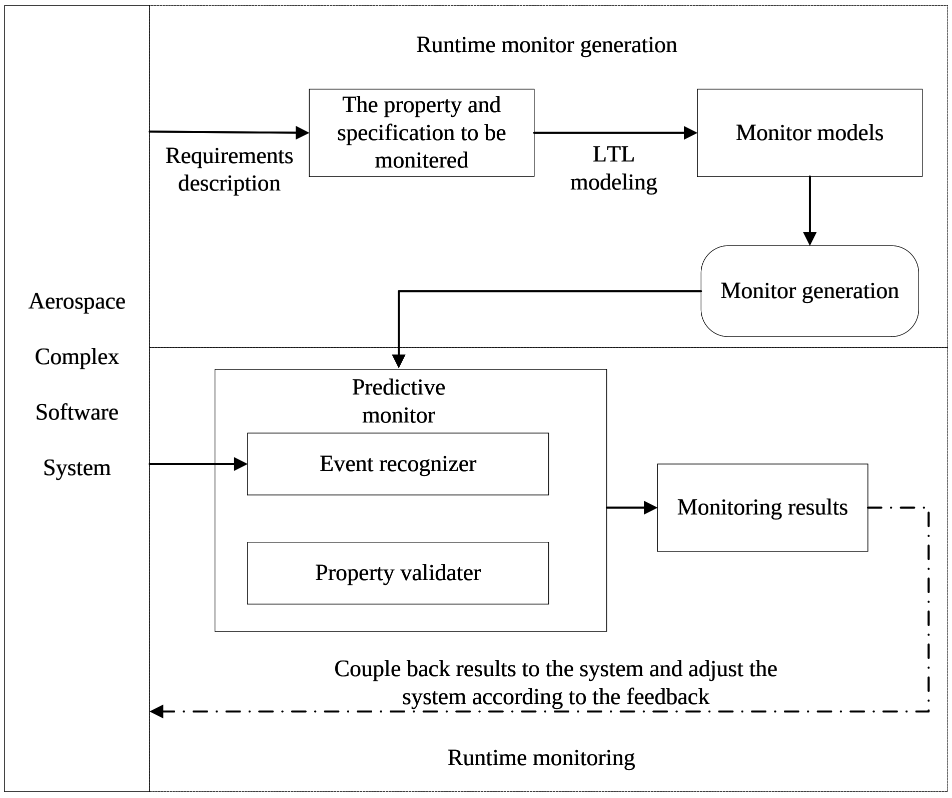
Runtime monitor generation
The property and specification to be monitered
Monitor models
Requirements description
LTL modeling
Monitor generation
Aerospace Complex Software System
Predictive monitor
Event recognizer
Property validater
Monitoring results
Couple back results to the system and adjust the system according to the feedback
Runtime monitoring

Figure 1. Runtime verification technical framework.

The runtime verification technology framework is mainly divided into two parts: runtime monitor generation and runtime monitoring. Firstly, we need to construct a runtime monitor. After analyzing system requirements, we transform key requirements and key properties of the system into a formal specification for monitor through specification description language. On the basis of formal specification, we use the advanced theory and algorithm to generated the runtime monitor from high level specification automatically [13]. Secondly, the running software system needs to be monitored and verified. The event recognizer in runtime monitor is used to recognize the corresponding events and conditions. Monitor takes the results of event recognizer as input and calculates the next state based on input and current state. Moreover, monitor checks whether running process violates the property specification or not and stores the recorded paths and verification results. Meanwhile, the verification results return to the aerospace software system and the software system will adjust itself according to this feedback to avoid runtime failure.

Runtime verification only monitors the running situation of current path whether is in the line with user's established properties. Because only one path of execution need to be monitored, this method costs relatively smaller than model checking and theory proving. Runtime verification deals with the output part of the system, which does not focus on the input part of the system. The concept map of runtime verification is shown in Figure 2.

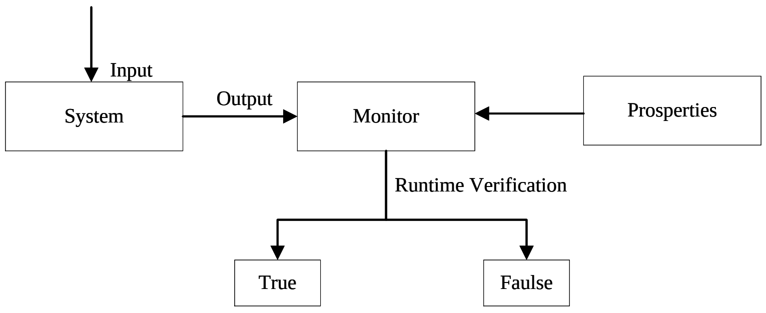

Figure 2. Runtime verification concept map

After operating the target system, we check whether the operating state of system satisfies its property specification by monitoring it, namely, checking whether the operating state of the system satisfies its property specification or violates certain correctness properties. Runtime verification only detects the violation or satisfaction of the correctness properties; therefore, once any violation is found the monitor will report.

3. Runtime fault detection based on runtime verification

This paper proposes a fault detection method based on runtime verification, which is a lightweight verification technology. The goal of this method is to check whether the actual operation of software system violates the specified properties and to detect the fault of software system. Linear Temporal Logic (LTL) formula is used to describe the property specifications and generate the corresponding monitor [3]. Monitor is usually generated from the system requirements and is used to check the operation process whether meet the demand of the system by monitoring and observing program execution. Runtime verification technology cannot only effectively detect abnormal behavior during system operation, but also makes it possible to effectively repair systems when the correctness deviation is detected. On the other hand, the proposed method uses online monitoring to monitor the operating system and checks the operation of software system in a progressive way without checking all operating states of the software system once in a time. The fault detection process can be shown in Figure 3.

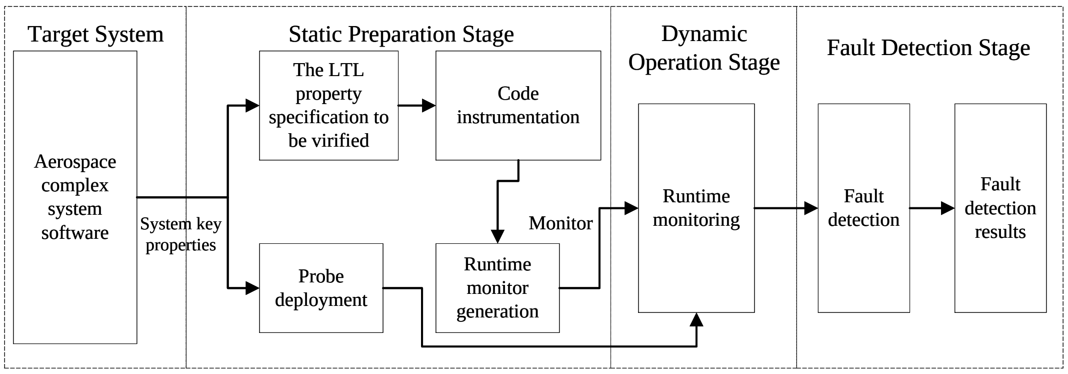

Figure 3. Fault detection based on runtime verification.

The fault detection method proposed in this paper can be divided into three stages: the static preparation stage, the dynamic operation stage and fault detection stage.

3.1. Static preparation stage

Before the operation of software system, according to high-level requirements description of target software system, we determine the property specifications. And then based on the property specifications we implement codes and deploy probes into target software system. In this stage, we need to construct a monitor according to system specifications.

- Define the property specifications

In the framework of software oriented runtime, verification methods, events and conditions are divided into atomic events and conditions, compound events and conditions. Atomic events and conditions are abstracted from variables and methods in the target system [14]. While compound events and conditions have no connection with the variables and methods in the target system, they are the combination of atomic events and conditions. This method uses the atomic event definition language PEDL and the compound event definition language MEDL to define software system requirements as the property specification.

Firstly, to get PEDL specification, the bottomed atomic event definition language PEDL is used to describe the key properties in the target software system and the basic events and conditions that are composed of these properties. Secondly, we use high-level compound event definition language MEDL to describe the compound events and conditions, which are composed of basic events and conditions. And then we can get the MEDL specification [10]. Finally, we define the LTL property specification to be verified which linking the high-level test requirements of software system to the bottomed execution information.

- Code instrumentation

PEDL specification and MEDL specification are compiled to obtain event recognizer that can collect the status information of target runtime software system and get the instrumentation position of the event recognizer. Before software system program is compiled, the code of event recognizer to target system program needs to be inserted by code instrumentation tools [5]. That is, the state collecting instructions are inserted in front of the specified statements and key code fragments to collect changing data of the monitored variables during the execution of target software system.

- Runtime monitor generation

Linear Temporal Logic (LTL) is a basis of verification technology. For one hand, the traditional methods, such as design-time verification and model checking use LTL to solve verification problem. For the other hand, LTL has also been used for the emerging runtime verification [12]. During the operation of target system, a finite operating trace should always be monitored. However, the traditional LTL method is defined on the infinite trace and has no connection with fairness and predictability. Hence, we introduce LTL$_3$ as a lineartime temporal logic, which shares the syntax with LTL but deviates in its semantics for finite traces [8]. Based on LTL$_3$, monitor can get the LTL property semantic according to a finite prefix. And we use three truth values: true, false, and inconclusive, to express the LTL$_3$ semantic [1]. The LTL$_3$ formula corresponding to target software system is determined according to the system properties. And LTL$_3$ formula is converted into a finite state machine, which is considered as the monitor, using the automata theory.

Having defined the semantics of LTL$_3$, we develop and discuss a monitor generation technique to construct a deterministic finite-state machine for an LTL property. The construction steps are as followed.

(1) Convert the LTL formula to Nondeterministic Büchi Automata.

Firstly, convert the LTL formula to the Alternating Büchi Automata(ABA), $A_\varphi = (Q_\varphi, \Sigma_\varphi, \delta_\varphi, I_\varphi, F_\varphi)$,using the theory of correctness detection and the formation of syntax tree. Secondly, convert the ABA to Nondeterministic Generalized Büchi Automata(NGBA), $G_A = (Q', \Sigma, \delta', I, \Gamma)$. Finally, convert the NGBA to Nondeterministic Büchi Automata(NBA), $B_G = (Q \times \{0,...,r\}, \Sigma, \delta', I \times \{0\}, Q \times \{r\})$, $B_G = (Q \times \{0,...,r\}, \Sigma, \sigma', I \times \{0\}, Q \times \{r\})$, according to the following rule:

$\delta'((q, j)) = \{(\alpha, (q', j')) \mid (\alpha, q') \in \delta(q))且 j' = next(j, (q, \alpha, q'))\}$, the next function is:

$$next(j,t)=\begin{cases}\max\{j\le i\le r \mid \forall j<k\le i, t\in T_k\} & j\ne r\\ \max\{0\le i\le r \mid \forall 0<k\le i, t\in T_k\} & j=r\end{cases} \tag{1}$$

(2) Construct Nondeterministic Finite Automata(NFA) and Deterministic Finite Automata(DFA).

Because not every Nondeterministic Büchi Automata can be converted to a deterministic one, we use the following steps to get a deterministic finite-state machine, which is the DFA [4]. Firstly, we concert the NBA to NFA, $\hat{A}=(Q,\sum,\delta,I,F)$, by some optimization operations, such as judging whether the NBA is empty and converting the NBA to a finite one. Then get the DFA, $\tilde{A}=(Q',\sum,\delta',I',F')$,according to some optimization rules.

(3) On the basis of deterministic finite automata, we take the Cartesian product of NFA and DFA and mark additional output symbols at each state of the automata.

Through all these operations, we can get the Finite State Machine(FSM) to be a runtime monitor [2]. Note that the resulting monitor evaluates LTL properties as predictively as possible, because once it can decide whether LTL properties will remain satisfied or unsatisfied, the monitor will provide this information immediately. The monitor can report a violation of a given property as early as possible.

(4) Probes deployment

The key variables and key functions of the software system are determined according to the property specifications and they are identified as the detection nodes where the probes to be deployed on [7]. These probes are used to track and record the execution paths of aerospace software system.

3.2. Dynamic operation stage

During the process of system operation, the instrumentation codes collect the changing information of the monitored variables. The monitor verifies the operating state of property specifications based on the changing information, and the probes record the execution path of software system.

Event recognizer collects the changing data of the monitored variables and sends the changing information to runtime monitor. Runtime monitor verifies the changing data of the monitored variables and decides whether the changing information satisfies or violates the property specification [9]. If the runtime monitor determines that the execution of software system conforms to the property specification, the output verification result is 0. While, if the runtime monitor determines that the execution of the software system does not comply with the property specification, the output verification result is 1. If the runtime monitor can’t determine whether the execution is consistent with the property specification, the output verification result is not valid.

In the process of software system execution, if the key variables or key functions where are the probe deployed on is operated, the probe output result is 1. Otherwise, the probe output is 0. After operating the software system, the vector of the probe value is used as the execution path record of software system. During this stage, we execute system for several times and record the execution paths and verification results.

3.3. Fault detection stage

According to the software execution paths and monitor verification results, software system fault detection is realized by statistical analysis.

Assuming that this paper delivers the software system execution for M times, and M' is the number of effective executions and verification results. Based on effective verification results and execution paths, we can get the software failure analysis matrix Q.

According to the software failure analysis matrix, the number of software failures occurred during one execution of software system can be calculated using statistical analysis method. Based on the statistical results, one can detect the fault in the target aerospace software system.

4. Conclusions

In recent years, with the continuous development of Lunar exploration, Mars exploration and other deep space exploration, how to ensure the high reliability and security of aerospace system and software is becoming more urgent. And the complexity of aerospace software system and the possibility of system failure have also increased. In order to improve aerospace software system to adapt to the changing external environment and timely to respond to runtime system failures, this paper focuses on the aerospace complex software system runtime fault detection technology. This paper proposes a runtime verification framework and a runtime monitor generation method. Based on this research, we propose a runtime fault detection method to recognize the aerospace software system failure as soon as possible. In the future, we will pay our attention on this field continuously and carry out in-depth research on tool development and contrast tests to promote the continuous development of aerospace software system and to ensure the security of aerospace software system.

References

1. F. Barbon, P. Traverso, M. Pistore, and M. Trainotti, “Runtime Monitoring of Instances and Classes of Web Service Compositions”, *In International Conference on Web Services* (ICWS), IEEE Computer Society, pp. 63–71, 2006
2. L. Baresi, S. Guinea, and L. Pasquale, “Towards a Unified Framework for The Monitoring and Recovery of BPEL Processes”, In *Workshop on Testing, Analysis, and Verification of Web Services and Applications* (TAV-WEB), ACM, pp. 15–19, 2008
3. A. Bauer, M. Leucker, and C. Schallhart, “Runtime Verification for LTL and PTLTL”, *Technical Report*, TUM-I0, 724, 2007
4. A. Bauer, M. Leucker, and C. Schallhart, “Comparing LTL Semantics for Runtime Verification”, *Logic and Computation*, pp. 651-674, 2010
5. E. Bodden, “A Lightweight LTL Runtime Verification Tool for Java”, In *OOPSLA Companion*, ACM, pp. 306–307, 2004
6. M. Broy, “Software Technology Formal Methods and Scientific Foundations”, *Information and Software Technology*, vol. 41, pp. 947-950, 1999
7. B. D’Angelo, S. Sankaranarayanan, C. S´anchez, “LOLA: Runtime Monitoring of Synchronous Systems”, In *International Symposium on Temporal Representation and Reasoning* (TIME), pp. 166–174, 2005
8. P. Gastin, and D. Oddoux, “Fast LTL to Büchi Automata Translation”, *Lecture Notes in Computer Science*, pp. 53–58, 2012
9. M. Geilen, “On the Construction of Monitors for Temporal Logic Properties”, *Electronic Notes in Theoretical Computer Science* (ENTCS) 55, 2, 2001
10. Bengtsson J., Larsen K. G., Larsson F., Pettersson P., “UPPAAL: A Tool Suite for The Automatic Verification of Real-time Systems”, *Lecture Notes in Computer Science*, vol. 1066. Springer-Verlag, pp. 232–243, 1996
11. L. Osterweil, “Software Processes Are Software Too”, In *proceedings of the 9th International Conference on Software Engineering*, Los Alamitos: IEEE Computer Society press, pp. 2-13, 1987
12. A. Pnueli, “The Temporal Logic of Programs”, In *Symposium on the Foundations of Computer Science* (FOCS), IEEE Computer Society Press, Providence, Rhode Island, pp. 46–57, 1977
13. V. Stolz and F. Huch, “Runtime Verification of Concurrent Haskell Programs”, In *Proceedings of the Fourth Workshop on Runtime Verification*, to appear in ENTCS, Elsevier Science Publishers, 2004
14. M. Y. Vardi and P. Wolper, “An Automata-theoretic Approach to Automatic Program Verification”, In *Logic in Computer Science*, A. Meyer, (ed.), pp. 332–345, 1986