A Survey of Health Management System for On-The-Fly Repairing of Concurrency Errors in Airborne Software

Junho Lee[1], Seongyun Go[2], Eu-Teum Choi[3] and Seongjin Lee[4,*]

[1]*Dept. of Aerospace Software Engineering, Gyeongsang National University, Jinju, Republic of Korea*
[2]*Dept. of Aerospace and Software Engineering, Gyeongsang National University Jinju, Republic of Korea*
[3]*Convergence Research Center for Materials and Mechanical Systems, Jinju, Republic of Korea*
[4]*Dept. of AI Convergence Engineering, Gyeongsang National University, Jinju, Republic of Korea*

Abstract

Concurrency errors are known for their difficulty of debugging and reproducing prior to execution. Undetected concurrency errors could result in nondeterministic executions that deviate from the programmer's intent and significantly undermining the reliability of the program. To prevent functional failure in airborne software, on-the-fly repairing of concurrency errors is crucial. This paper surveys the Health Management System for on-the fly repairing of concurrency errors in airborne software and motivates the future work in concurrent airborne software.

Keywords

Concurrency error, reliability, airborne software, on-the-fly repairing, Health Management System,

1. Introduction

Airborne software is the embedded software that controls, manages, and monitors the state of airborne system[1, 2]. With the rapid advances in avionics, the importance of software in avionics has been increasing. In the 1960s, the proportion of software in F-4 was only 4%. Whereas, in the case of F-35 produced in 2007, the proportion of software had risen to 90%[3]. It indicates that software has played a crucial role in avionics system. As the software made up in avionics system has dramatically risen, the scale and complexity of airborne software have also grown. As a result, debugging of airborne software has become more challenging and the risk of system failure has increased due to potential errors and faults[4, 5, 6, 7].

In aircraft, when accidents occur due to the system failure, they could result in catastrophic loss of life and property. This serious issue can be observed in the accident that occurred during the inaugural test flight of the F-22 Raptor in 1992[4]. The failure of control software made pilot unable to control the induced oscillation, leading to a crash. Other examples can be observed in the case of Boeing 737 MAX accidents[8]. Boeing's new aircraft model, the Boeing 737 MAX, experienced two crashes in October 2018 and March 2019, resulting in the loss of 189 lives and

ISE 2023: 2nd International Workshop on Intelligent Software Engineering, December 4, 2023, Seoul
*Corresponding author.
dlwnsgh0901@gnu.ac.kr (J. Lee); tjddbs6696@gmail.com (S. Go); etchoi@gnu.ac.kr (E. Choi); insight@gnu.ac.kr (S. Lee)

157 lives, respectively. The main cause of the accidents was determined to be a malfunction of Boeing's newly introduced technology, MCAS (Maneuvering Characteristics Augmentation System). MCAS is a technology designed to prevent a crash by lowering the aircraft's nose when a stall is anticipated. In the two instances of flight, due to the malfunctions of MCAS, the aircraft's nose pitched abnormally downward. This made the pilots unable to respond appropriately, leading to the aircraft crashes.

Concurrency errors are typically one of the most critical error types occurring in concurrent programs, and they must be eliminated as they can compromise the intended execution results of the program[9, 10, 11, 12, 13, 14]. Concurrency errors are difficult to debug and reproduce, so they are not easily eliminated during the development phase. To ensure the correct execution of airborne software, these errors must be eliminated during run-time. The software health management system in airborne software diagnoses and treats the errors occurring during the execution, ensuring the proper functioning of the program. Diagnosis is performed by monitoring the state changes of specific values or variables in a program. If an error is diagnosed, appropriate recovery actions are employed to ensure the proper execution of the program. Since 2010, there have been four studies in airborne software that utilize health management systems to eliminate concurrent errors during execution. Three researches[6, 7, 4] diagnose and repair atomicity violations among concurrent errors, while only one research[5] focuses on diagnosing and repairing order violations within the same context. Ha et al.[6][6] and Tchamgoue et al.[7] detect atomicity violations by comparing diagnosis algorithms with actual execution information, while Choi et al.[4] repairs atomicity violations by comparing pre-collected correct execution information with actual execution information. For on-the-fly repairing of order violation in airborne software, Kim et al.[5] diagnoses order violations by comparing the user-defined function call sequences with the information acquired during execution.

The remainder of this paper is organized as follows. Section 2 gives the research background discussing concurrency errors and health management systems for on-the-fly repairing them in airborne software. In section 3, related works using health management systems to repair concurrency errors in airborne software are introduced. It also discusses some research questions of related work in section 4. Finally, section 5 concludes the research.

2. Background

This section introduces concurrency errors that are hard to debug and remove during the development phase. It also explains the health management system for diagnosing and on-the-fly repairing errors in airborne software during the operational phase.

2.1. Concurrency Error

Concurrency errors occur when two or more different threads access shared resources without proper synchronization, resulting in outcomes different from the programmer's intent[11, 4, 5, 6, 7]. Types of concurrency errors include atomicity violation, order violaiton, and deadlock[11].

Atomicity violations occur in concurrent programs when at least one write access occur without proper synchronization[11, 15, 16, 17], violating the atomicity of the program's execution in the affected section that should be performed atomically. Fig. 1 depicts two threads accessing a

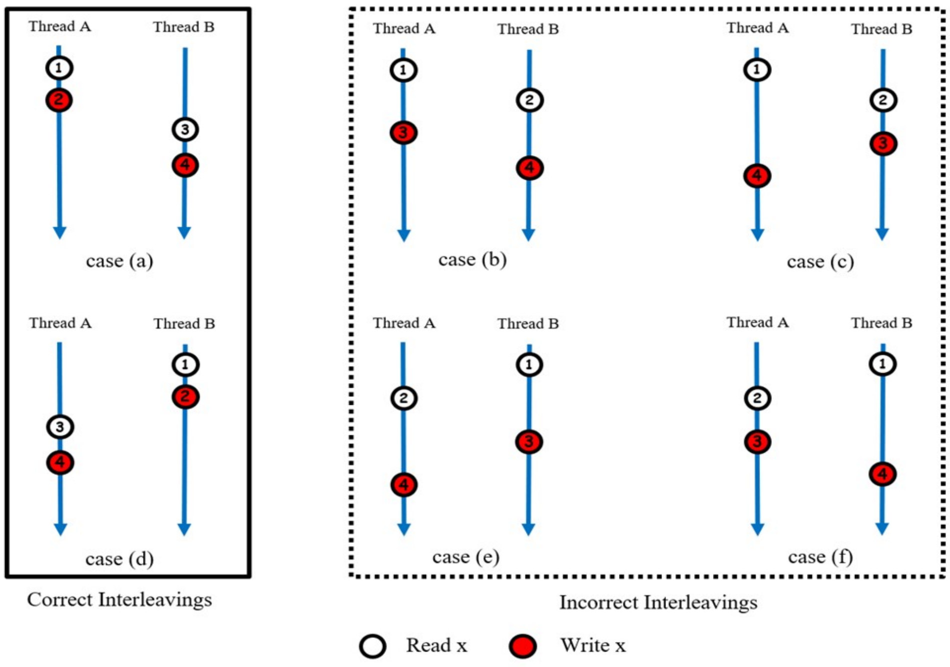


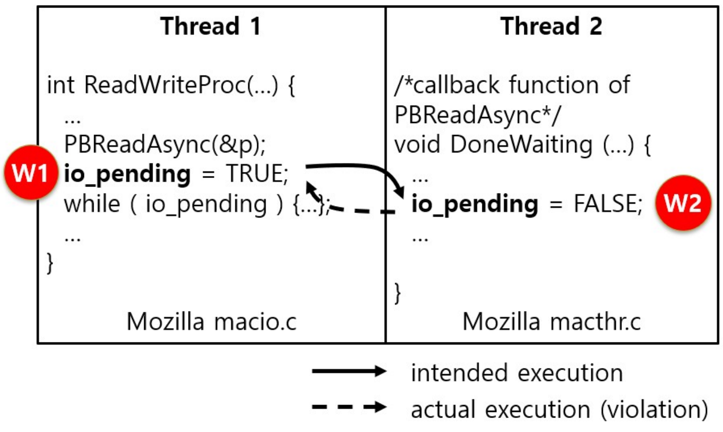
Figure 1: Possible interleavings of two threads performing x++ operation respectively

shared variable sequentially, performing read and write operations respectively without proper synchronization. In the Fig. 1, white circles represent the read access to the shared variable x, while red circles represent write accesses to the shared variable x. Programmers typically expect that the result will increase by 2 when the x++ operation is performed twice. In cases a and d, the value of x is incremented by 2 because read and write accesses to the shared variable are performed atomically in the x++ operation. However, in cases b, c, e, and f, read and write accesses to the shared variable are not performed atomically, resulting in an increment of only 1 even if the x++ operation is performed twice. Such non-deterministic behavior makes it difficult to guarantee the intended execution of the program.

Order violation is occurred when other threads disrupt the intended sequence of shared variable access defined by the developer[11, 10, 5]. Fig. 2 provides an example of order violation that occurred in Mozilla. In this example, two threads access and perform write operations on a shared variable called `io_pending`. The developer's expectation is that thread 1 initializes `io_pending` to TRUE, and after a certain period, thread 2 will change it to FALSE. However, due to an incorrect interleaving, thread 2 could mistakenly initializes `io_pending` to FALSE first, and then thread 1 attempts to change it to TRUE. In this scenario, an error can occur, resulting in an infinite execution of the code inside a while loop.

Deadlock is a type of concurrency error in which a process is unable to proceed when two threads each possess a resource that the other thread requires and they request each other's resources.

The purpose of the health management system (HMS) for airborne software is to prevent



```
int ReadWriteProc(...) {
  ...
  PBReadAsync(&p);
  io_pending = TRUE;
  while ( io_pending ) {...};
  ...
}
```

```
/*callback function of
PBReadAsync*/
void DoneWaiting (...) {
  ...
  io_pending = FALSE;
  ...

}
```

Figure 2: Example of an order violation in Mozilla

system failures caused by faults or errors[1, 4, 5, 6, 7]. Errors can occur during operation due to faults that were not eliminated during the development phase. HMS monitors, diagnoses, and treats these errors at the module level, partition level, and process level. For error diagnosis, HMS continuously monitors specific variables or state changes. When an error is diagnosed, HMS employs recovery techniques to eliminate the error. Recovery techniques can be categorized into Forward Recovery and Backward Recovery, depending on the timing of dealing with errors. Forward Recovery is relatively straightforward to implement but may not be able to repair all errors. Whereas, Backward Recovery involves returning to the last checkpoint where no errors occurred to correct the fault. This method can address most errors but comes with significant time and space overheads when rolling back to a checkpoint.

2.2. Health Management System of Airborne Software

Since 2010, research on utilizing health management systems of airborne software has been actively pursued to diagnose and repair errors [4, 5, 6, 7]. These Researches focus on repairing concurrency errors[4, 6, 7], one of the most severe types of software errors, which occur in concurrent programs. Ha et al.[6], Tchamgoue et al.[7], Choi et al.[4] diagnose and repair atomicity violations. To diagnose atomicity violations, Ha et al.[6] and Tchamgoue et al.[7] incorporate probe code through actual run-time diagnostic algorithms. Choi et al.[4] compares previously executed correct execution information with actual run-time information. Kim et al.[5] diagnoses and repairs order violation errors by comparing the user-defined function call sequence and information acquired during execution.

3. Related Works

Errors occurring during the operational phase due to fautls that were not eliminated during the development phase can threaten the normal execution of airborne software. Given that eliminating concurrency errors by considering all possible interleavings of a program during

work	Diagnosis	Treatment
Ha et al.	atomicity violation	lock, unlock
Tchamgoue et al.	atomicity violation	wait/set_event
Choi et al.	atomicity violation	wait, signal
Kim et al.	order violation	wait, signal

Table 1
HMS for repairing concurrency errors in airborne software

the development phase is impossible, it becomes critically important to address these errors during execution[9, 10, 12, 13, 14]. Since 2010, research in the field of airborne software has been underway to repair atomicity violations and order violations using health management systems.

Table 1 lists research studies in the field of airborne software that utilize health management systems to repair concurrency errors. There are three studies focusing on repairing atomicity violations. To diagnose atomicity violations, Ha et al.[6] and Tchamgoue et al.[7] incorporate detection protocols, while Choi et al.[4] diagnoses atomicity violations by comparing information obtained from pre-execution with information from actual run-time. All three studies include synchronization techniques to remedy atomicity violations.

In the cases of Ha et al.[6] and Tchamgoue et al.[7], they diagnose atomicity violations by inserting detection protocols and comparing the actual execution results. The detection protocol first determines the concurrency suitability of threads using labeling techniques. Each thread is assigned a unique number, and information about shared variable access and the execution sequence relationship among threads is stored in a data structure called 'label.' The labeling technique is then used to check whether threads executing in parallel are protected by synchronization techniques to diagnose atomicity violations. When an atomicity violation is diagnosed, these studies control the thread flow by inserting POSIX lock/unlock [6] or using APEX's wait/set event [7]before and after shared variable access for threads without proper synchronization.

Choi et al.[4] diagnoses atomicity violations by comparing Anticipated Invariant (AI)[14] based BSet() and RPre(). BSet() represents the correct execution order information obtained through pre-execution of the program, while RPre() represents the actual execution information. Choi et al.[4] considers it an atomicity violation if RPre is not included in the previously collected correct execution, BSet, and it repairs atomicity violations by delaying the specific thread accessing shared variables by performing APEX's wait and signal.

To diagnose order violations, Kim et al.[5] compares the user-defined function call sequence with information acquired during execution. When an order violation is detected, wait and signal calls are used to remove it.

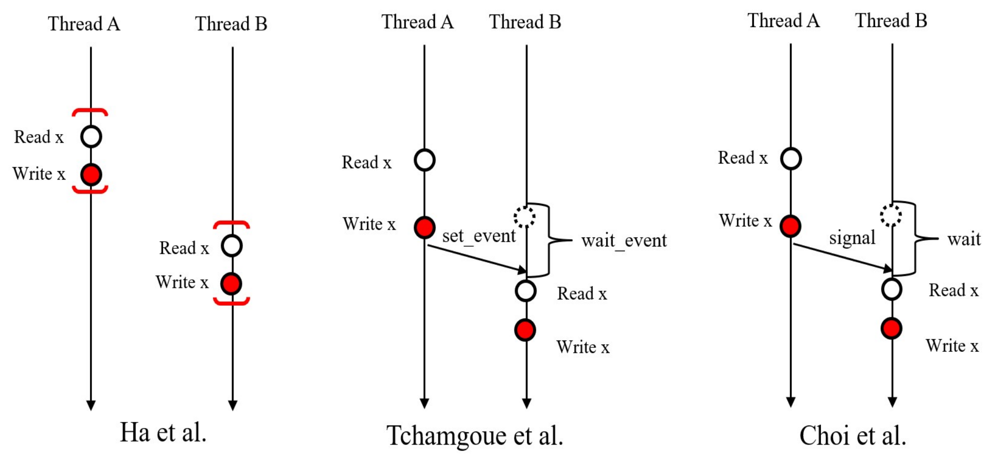


Figure 3: Techniques for repairing of atomicity violations

4. Research Questions

4.1. RQ1: Is it possible to repair atomicity violations regardless of the number of shared variables involving in errors?

Atomicity violation can be categorized into a single-variable atomicity violation and multi-variable atomicity violation depending on the number of shared variables involving in errors[11]. In the case of a single-variable atomicity violation, it occurs when two or more threads access a shared variable concurrently without proper synchronization. Whereas, multi-variable atomicity violation occurs when two or more threads access two or more shared variables concurrently without proper synchronization. To diagnose and repair multi-variable atomicity violations, correlations among shared variables should be considered[11]. However, three existing studies only focus on a single-variable atomicity violation, not considering the correlations among the shared variables. Therefore, it is limited to diagnose and repair multi-variable atomicty violations using existing techniques.

4.2. RQ2: Is it possible to repair atomicty violations in signal-driven program?

When a signal occurs in a program, the program is terminated or preempted by a user-defined signal handler[12]. Due to the characteristics of signals having higher priority than threads, they preempt the execution flow of threads upon occurrence. In a situation where a signal is raised while a thread is accessing a shared variable and sharing it with a signal handler, there is a potential for atomicity violation. In a scenario where a thread increments the value of a shared variable x using the x++ instruction, and a signal handler also increments the value of x using x++ instruction, if a signal is raised between the read and write access of the thread, atomicity violation may occur. The value of x will be incremented by only 1 when the x++ operation is performed twice. If atomicity violation is diagnosed, related works control threads inserting lock/unlock[6], APEX set/wait_event[7], POSIX wait/signal[4]. However, using these mechanisms in a signal handler, deadlock can occur. Due to the inherent nature of signal handlers, once invoked, they do not return to the thread until their execution is complete. If, however, control is transferred to a signal handler while a thread has acquired a lock, the signal handler is simply waiting for the thread to release the lock, and the thread is waiting for the signal handler to return, potentially resulting in a deadlock. Therefore, there is a need for future research to investigate methods that block the access of a signal handler, which shares a variable with a thread, when the thread is accessing that variable.

5. Conclusion

This paper discussed one of the most serious type of software errors known as concurrency errors. To ensure the correct execution of the program, it is essential to eliminate undetected concurrency errors. The Health Management System of Airborne software is designed to diagnose and repair these errors during execution. While three studies have focused on repairing atomicity violations, one study has addressed to repair order violation in airborne software. The research question indicates that existing works for repairing atomicity violations primarily concentrate on a single-variable atomicity violation, and a data race among threads. Therefore, there is a need for future research to expand the scope of repairs to include multi-variable atomicity violations and those caused by signal handlers.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No. 2021R1A2C1014163).

References

[1] A. E. E. C. (AEEC), Avionics Application Software Standard Interface – ARINC Specification 653 – Part 1, ARINC Inc, 2015.

[2] J. Knight, The glass cockpit, Computer 40 (2007) 92–95.

[3] D. G. Firesmith, P. Capell, D. Falkenthal, C. B. Hammons, L. DeWitt, T. Merendino, The method framework for engineering system architectures, CRC Press, 2008.

[4] E.-t. Choi, T.-h. Kim, Y.-K. Jun, S. Lee, M. Han, On-the-fly repairing of atomicity violations in arinc 653 software, Applied Sciences 12 (2022) 2014.

[5] T.-H. Kim, E.-T. Choi, Y.-K. Jun, An efficient on-the-fly repairing system of order violation errors for health management of airborne software, The Korean Society for Aeronautical and Space Sciences 12 (2020).

[6] O.-K. Ha, G. M. Tchamgoue, J.-B. Suh, Y.-K. Jun, On-the-fly healing of race conditions in arinc-653 flight software, in: 29th Digital Avionics Systems Conference, IEEE, 2010, pp. 5–A.

[7] G. M. Tchamgoue, O.-K. Ha, K.-H. Kim, Y.-K. Jun, A framework for on-the-fly race healing in arinc-653 applications, International Journal of Hybrid Information Technology, SERSC 4 (2011) 1–12.

[8] D. KOENIG, New software glitch found in boeing's troubled 737 max jet (2011).
[9] Y. Lin, S. S. Kulkarni, Automatic repair for multi-threaded programs with deadlock/livelock using maximum satisfiability, in: Proceedings of the 2014 International Symposium on Software Testing and Analysis, 2014, pp. 237–247.
[10] B. Lucia, L. Ceze, Cooperative empirical failure avoidance for multithreaded programs, ACM SIGPLAN Notices 48 (2013) 39–50.
[11] S. Lu, S. Park, E. Seo, Y. Zhou, Learning from mistakes: a comprehensive study on real world concurrency bug characteristics, in: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems, 2008, pp. 329–339.
[12] G. M. Tchamgoue, K. H. Kim, Y.-K. Jun, Eventhealer: Bypassing data races in event-driven programs, Journal of Systems and Software 118 (2016) 208–220.
[13] L. Zhang, C. Wang, Runtime prevention of concurrency related type-state violations in multithreaded applications, in: Proceedings of the 2014 International Symposium on Software Testing and Analysis, 2014, pp. 1–12.
[14] M. Zhang, Y. Wu, S. Lu, S. Qi, J. Ren, W. Zheng, A lightweight system for detecting and tolerating concurrency bugs, IEEE Transactions on Software Engineering 42 (2016) 899–917.
[15] G. Jin, W. Zhang, L. B. Deng, Dongdong, S. Lu, Automated {Concurrency-Bug} fixing, in: 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), 2012, pp. 221–236.
[16] C. Li, R. Chen, B. Wang, T. Yu, D. Gao, M. Yang, Precise and efficient atomicity violation detection for interrupt-driven programs via staged path pruning, in: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, 2022, pp. 506–518.
[17] A. Muzahid, N. Otsuki, J. Torrellas, Atomtracker: A comprehensive approach to atomic region inference and violation detection, in: 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, IEEE, 2010, pp. 287–297.