International Journal of Software Engineering and Its Applications
Vol. 10, No. 1 (2016), pp. 201-212
http://dx.doi.org/10.14257/ijseia.2016.10.1.19

AR653: Dynamic Detection Tool for Atomicity Races in ARINC 653 Applications

Eu-Teum Choi[1], Ok-Kyoon Ha[2] and Yong-Kee Jun[1*]

[1]*Department of Informatics, Gyeongsang National University*
[2]*Engineering Research Institute, Gyeongsang National University*
{etchoi, jassmin, jun}@gnu.ac.kr

Abstract

Atomicity races in ARINC 653 applications are a kind of concurrency bugs which causes nondeterministic behaviors by parallel processes. The defects must be detected to ensure the reliability of the applications, because they may lead to unpredictable results to the programmer. This paper presents a tool, called AR653, to dynamically detect atomicity races for an execution of the application. The tool monitors only minimal information, such as processes, semaphores, and read/write accesses to shared resources, and analyzes the relation of synchronizations to report atomicity races throgh a locking decipline of semaphores. We compared the accuracy of AR653 with CodeSonar using synthetic programs on a simulation system for integrated modular avionics. The emperiacal results show that our tool correctly reports atomicity races in cases of using shared pointers as well as in cases of using shared variables, while CodeSonar only locates atomicity races in cases of unsing shared variables.

Keywords: *ARINC 653 operating systems, atomicity races, dynamic detection, synchronization, avionics*

1. Introduction

The ARINC 653 standard [1-6] for integrated modular avionics [6-10] defines four intra-partition communication services: buffer, blackboard, semaphore and event. The semaphore service creates an atomic execution region in the application and enables to control access to a shared resource by concurrent processes. However, atomicity races may occur in ARINC 653 applications without explicit synchronization such as semaphore or using incorrect synchronization, when more than two processes access a shared resource with at least one or more write accesses included. Such atomicity races must be detected because the reliability of the applications cannot be guaranteed.

It is important to detect atomicity races [10-13] in the applications for the safety critical of avionics systems. CodeSonar [14] is a representative tool which is widely used to identify the concurrency bugs, such as data races and deadlocks, as well as sequential errors based on the static analysis techniques [15-17] using source codes. The static analysis tool is sound, but imprecise because it produces a lot of false positives through evaluation all of possible executions including impractical execution paths which are never reached in the actual execution of the applications. This requires the programmer to analyze the results reported by CodeSonar again.

This paper presents a tool, called AR653 that consists of AR653 Monitor and AR653 Detector to detect atomicity races during an execution of ARINC 653 application. The AR653 Monitor collects information such as processes, semaphores, and accesses to shared resources, and AR653 Detector analyzes the relation of synchronizations to report atomicity races through a locking discipline of semaphores. We implemented AR653 on

* Corresponding Author

ISSN: 1738-9984 IJSEIA

top of PIN binary instrument [18-19] software framework and evaluated the accuracy of the tool on a simulation system for integrated modular avionics (SIMA) by using ARINC 653 synthetic programs. The empirical results show that our tool correctly reports atomicity races in cases of using shared pointers as well as in cases of using shared variables, while CodeSonar only locates atomicity races in cases of using shared variables.

2. Background

2.1. Intra-Partition Communication for ARINC 653 Standards

Avionic systems have been changing from Federated Systems to Integrated Modular Avionics (IMA) which purposes to reduce the weight of air-born systems and its power consumption by integrated management of the system. The ARINC 653 Specification has been developed as a standardized interface definition of real-time operating system to simplify the development of IMA. The ARINC 653 standard for integrated modular avionics specifies the semaphore service for intra-partition communication. The semaphore service is a synchronization object commonly used to create the atomic execution region and to control process accesses to the shared partition resources. Figure 1 shows the source code that can cause atomicity races due to the incorrect use of semaphore operations in the ARINC 653 operating system. Figure 1 (a) is an example of the source code that can cause atomicity races because the programmer incorrectly used the semaphore and the shared resources within the atomic regions are not protected. Figure 1 (b) shows an example of the source codes include the atomicity races due to the usage of different semaphores.

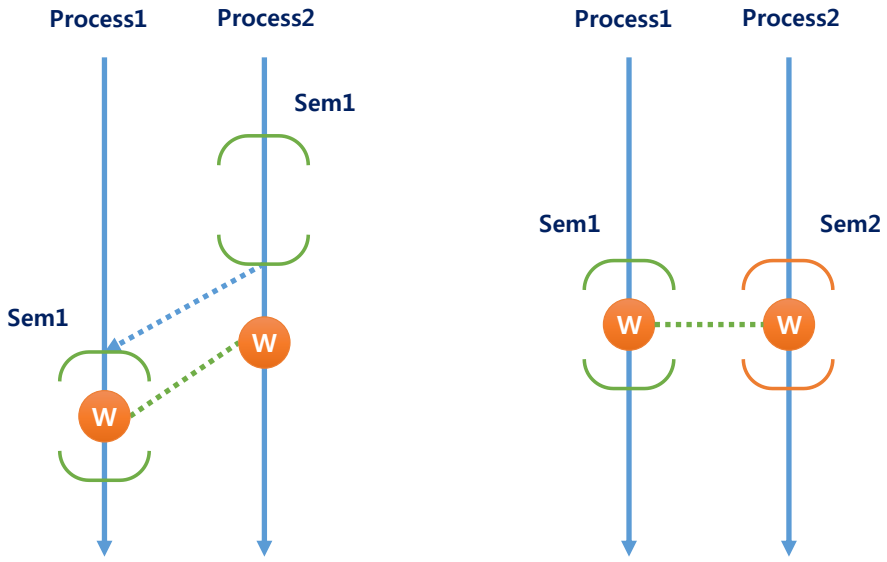
```
01:  void process1() {
02:      RETURN_CODE_TYPE rc;
03:      SEMAPHORE_ID_TYPE id;
04:
05:      GET_SEMAPHORE_ID("SEM1", &id, &rc);
06:
07:      while(1) {
08:          WAIT_SEMAPHORE(id, INFINITE_TIME_VALUE, &rc);
09:          counter += 1;
10:          if (counter == 100) {
11:              counter = 0;
12:          }
13:          SIGNAL_SEMAPHORE(id, &rc);
14:      }
15:  }
16:
17:  void process2() {
18:      RETURN_CODE_TYPE rc;
19:      SEMAPHORE_ID_TYPE id;
20:
21:      GET_SEMAPHORE_ID("SEM1", &id, &rc);
22:       while(1) {
23:          WAIT_SEMAPHORE(id, INFINITE_TIME_VALUE, &rc);
24
25:          SIGNAL_SEMAPHORE(id, &rc);
26:          counter += 1;
27:          if (counter == 100) {
28:              counter = 0;
29:          }
30:      }
31:  }
```

(a) Creation of incorrect Atomic Region

```
01:  void process1() {
02:      RETURN_CODE_TYPE rc;
03:      SEMAPHORE_ID_TYPE id;
04:
05:      GET_SEMAPHORE_ID("SEM1", &id, &rc);
06:
07:      while(1) {
08:          WAIT_SEMAPHORE(id, INFINITE_TIME_VALUE, &rc);
09:          counter += 1;
10:          if (counter == 100) {
11:              counter = 0;
12:          }
13:          SIGNAL_SEMAPHORE(id, &rc);
14:      }
15:  }
16:
17:  void process2() {
18:      RETURN_CODE_TYPE rc;
19:      SEMAPHORE_ID_TYPE id;
20:
21:      GET_SEMAPHORE_ID("SEM2", &id, &rc);
22:       while(1) {
23:          WAIT_SEMAPHORE(id, INFINITE_TIME_VALUE, &rc);
24
25:          counter += 1;
26:          if (counter == 100) {
27:              counter = 0;
28:          }
29:          SIGNAL_SEMAPHORE(id, &rc);
30:      }
31:  }
```

(b) Use of different Semaphores

Figure 1. Source Codes of ARINC 653 Applications Using Semaphores



(a) A shared resource is not protected by a semaphore

(b) Shared resources are protected by the atomic regions that use different semaphore

Figure 2. Examples of Atomicity Races

The ARINC 653 standard interface provides semaphore services including CREATE_SEMAPHORE, WAIT_ SEMAPHORE, SIGNAL_SEMAPHORE, GET_ SEMAPHORE_ID, and GET_SEMAPHORE_STATUS. The CREATE_ SEMAPHORE service is used to create semaphore. The process intended to use shared resources through the created semaphore executes WAIT_SEMAPHORE, and then executes SIGNAL_SEMAPHORE to allow other processes to access shared resources. GET_SEMAPHORE_ID is used to bring the current semaphore's ID, while GET_SEMAPHORE_STATUS is employed to know the status of the current semaphore.

2.2. Atomicity Races in ARINC 653 Applications

Atomic execution region refers to a code block where a program is executed as one atom by explicit synchronization. If the atomic operation is guaranteed, shared variables (read/amend) within an atomic region should not be fixed by other processes. Figure 2 represents the execution of the program shown in Figure 1. If shared resources are not protected by atomic regions as seen in Figure 2 (a) or they are protected by the atomic region that are different from each other such as Figure 2 (b), accesses to the shared resources may be conflicted by execution of other processes, and consequently the atomic operation cannot be guaranteed. Likewise, atomicity races may occur in ARINC 653 applications without explicit synchronization such as semaphore or using incorrect synchronization, when more than two processes access to a shared resource with at least one write access included. The atomicity races must be detected for debugging, because they may lead to unpredictable results. However, it is well known that the defects are hard to handle through the traditional debugging methods, such as break points or watch points since they are hard to reproduce in an execution of the program and are no errors in source codes.

2.3. Detection Tool for Atomicity Races

The integrated development environment for ARINC 653 applications provides information such as semaphores, messages, queues, and tasks, which are needed to configure an application and detect errors in a safe and efficient way. However, it is difficult to use these information and to check exist atomicity races in the applications, because detecting atomicity races requires to understand parallel executions of processes and to predict their nondeterministic behaviors. Therefore, in general, a range of automatic detection tools based on sophisticated techniques [10-27] is employed to locate atomicity races which exist in ARINC 653 applications. A representative tool for detecting atomicity races in the avionics application is CodeSonar. It analyzes the source code and binary code to detect concurrency errors such as data races and deadlocks and other serious programming errors. However, the static tool produces a lot of false positives because it evaluates all of possible executions including impractical execution paths which are never reached in the actual execution of a program.

3. Dynamic Detection of Atomicity Races

3.1. AR653

To dynamically detect atomicity races in ARINC 653 applications, the programmer must monitor and collect the information such as processes, semaphores and accesses to shared resources. We present a dynamic detector, called AR653 that locates atomicity races during an execution of the application. Figure 3 shows the overall architecture of AR653. AR653 consists of two modules: AR653 Monitor and AR653 Detector. During an execution of an application, the AR653 Monitor module collects execution information of the application, and the AR653 Detector module analyzes the relation of synchronizations.

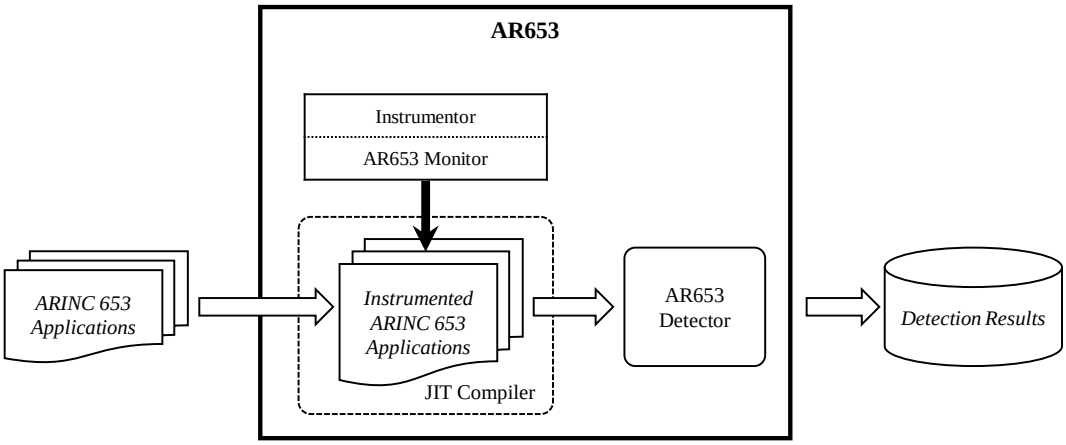


Figure 3. The Overall Architecture of AR653

With AR653, The programmer enters the viable binary code for ARINC 653 applications. Then, Dynamic Binary Instrumentor inserts AR653 Monitor, which monitors information needed to detect atomicity races, into the entered binary code. The AR653 monitors information such as processes, semaphores, and accesses to shared resources during an execution of the ARINC 653 applications. AR653 Detector analyzes the monitored information, checks if atomicity races exist, and reports the results.

3.2. AR653 Monitor

AR653 Monitor, which is inserted into the binary codes of ARINC 653 applications, collects processes, semaphores, and memory accesses during an execution of the

application. Figure 4 shows the architecture of AR653 Monitor which consists of three components: ProcessMonitor, SemaphoreMonitor, and AccessMonitor.

The ProcessMonitor component monitors information related to process function calls generated in an execution of the application. Generally, a process is created in ARINC 653 application by CREATE_PROCESS API. If the function is called with parameters including process name, it delivers the process name to create a new process and is assigned a process ID as an active process by the kernel of OS. The ProcessMonitor maintains these process information, such as the process IDs and the process names, to monitor the acting processes, and it removes the information when the processes expire.

The SemaphoreMonitor component is to monitor the occurrence of semaphores considering CREATE_SEMAPHORE, WAIT_SEMAPHORE, and SIGNAL_SEMAPHORE APIs. CREATE_ SEMAPHORE is a function to create a new semaphore object, which delivers the semaphore command through parameters and brings the semaphore ID assigned from the kernel. This enables to obtain the ID used in the application and the semaphore command. If WAIT_SEMAPHORE and SIGNAL_SEMAPHORE, the functions to control synchronization, are called, the semaphore ID can be obtained through the parameters of these functions.

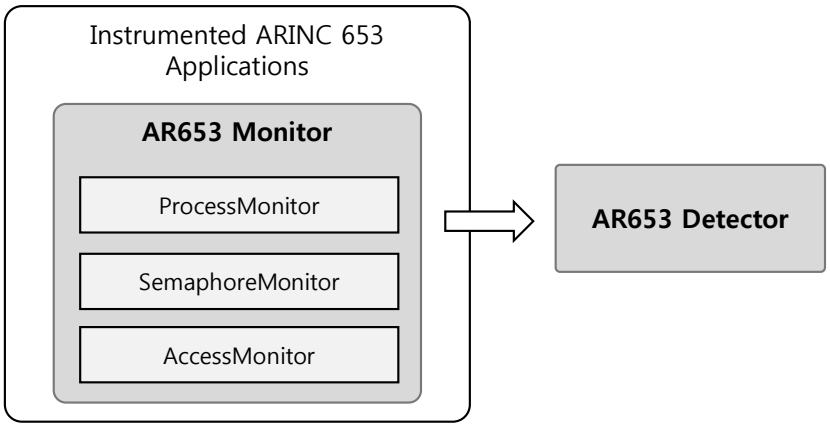


Figure 4. The Architecture of AR653 Monitor

The AccessMonitor component monitors the accesses to shared resources, such as read/write of shared memories. When an access event happens, this component acquires the type of access event, process ID, address of the shared variables, the name of function, and line number. The monitored and collected information by the AR653 Monitor is used to analyze the synchronization of an application execution for AR653 Detector module.

3.3. AR653 Detector

AR653 Detector reports atomicity races using an on-the-fly analysis technique, called lock-set based analysis. The Detector module checks a locking discipline which stipulates that any two different processes access a shared resource with a common semaphore. Also the Detector maintains the access histories for each shared resource and ratiocinates the protective relations to check if atomicity races exist. Figure 5 shows the architecture of AR653 Detector.

For detecting atomicity races, the AR653 Detector maintains a candidate set of semaphores C_x that is held by all processes during a program execution for a shared resource x. Therefore, it locates an atomicity race whenever any two accesses on different processes access a shared resource with at least one write, and the two accesses are not protected by a common semaphore. Given two processes P_i and P_j that access to a shared resource x is occur an atomicity race if they satisfy following condition:

$$(P_i = Write \vee P_j = Write) \wedge (P_i \neq P_j \wedge Cx = \emptyset) \quad (1)$$

where C_x maintains a set of semaphores by intersecting itself with the set of semaphores held by the current process. AR653 detects an atomicity race whenever a pair of conflicting accesses to a shared resource are satisfied the above condition.

AR653 Detector receives information such as processes, semaphores and accesses to shared variables monitored by AR653 Monitor during an execution of the application. The information acquired by Process Monitor is kept in Process Information from creation to extinction. Semaphore Information maintains the semaphore status from creation to extinction through the information acquired by Semaphore Monitor, so that the semaphore is used to check if atomicity races exist. Access History keeps the history of accesses to shared resources, and the history is used to check if atomicity races exist by comparing with the status of semaphore.

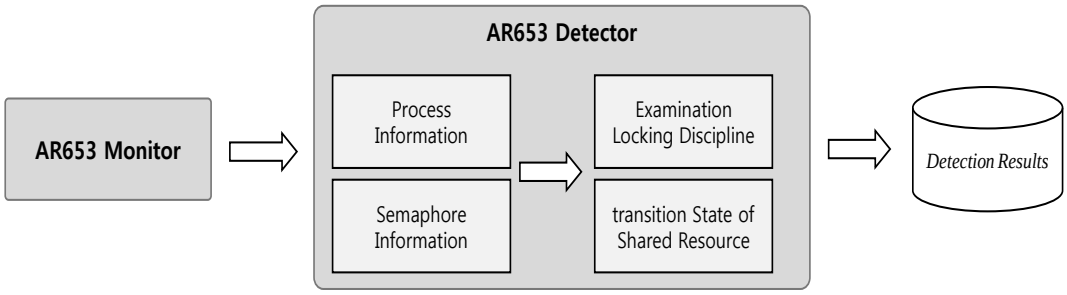


Figure 5. The Architecture of AR653 Detector

AR653 Detector checks the violation of a locking discipline and changing information through Process Information, Semaphore Information and Access History, which change during an application execution. The locking discipline basis that a shared variable, which two or more processes access, must be protected with a common lock. Also, AR653 Detector keeps the access history of a shared variable and ratiocinates its protective relation to check if atomicity races exist.

4. Experimentation

4.1. Methodology

To evaluate the accuracy of AR653, we used synthetic programs considering either involving atomicity race or not. We developed the synthetic programs considering two criteria such as using shared variable and using pointers of shared resources. To pair comparison, we employed CodeSonar and analyzed the synthesis to compare the results of the static tool with the results by AR653.

Figure 6 graphically shows the execution of synthetic programs using the shared variables. A shared integer variable is declared before the application is created and operated. Write events to the shared variable are generated in the process 1 and process 2. If the shared variable is not properly synchronized by semaphore, an atomicity race may occur. In the SV-N01 and SV-N02, the shared variable is properly synchronized by semaphore, resulting in no atomicity race. In the SV-R01, an atomicity race may occur because semaphore is not used for the shared variable of the two processes. In the SV-R02, an atomicity race may occur because the shared variable of the process 1 is protected with semaphore, but that of the process 2 is not protected.

Figure 7 graphically shows the execution of synthetic programs using the pointer for shared resources. These synthetic programs are similar to the synthetic programs for the shared resources.

The implementation and experimentation were carried on a single board computer with Linux Kernel, a real time operating system. ARINC 653 synthetic programs are compiled

with GNU C Compiler Version 4.6.2. The complied program was operated under a simulation system for integrated modular avionics (SIMA), and each synthetic program ran on the standalone mode.

4.2. Results

The results of detecting atomicity races in the synthetic programs by both AR653 and CodeSonar appear in Table 1. AR653 precisely locates the existence of atomicity races for four synthesis, SV-R01, SV-R02, SP-R01, and SP-R02, while CodeSonar reports atomicity races only for SV-R01 and SV-R02. Naturally, all of tools precisely detect no atomicity races for SV-N01 and SP-N01 which use a common semaphore for an explicit synchronization. However, AR653 reports false positives in case of SV-N02 and SP-N02 because these programs use implicit synchronization by two different semaphores. For optimized execution, these kind of programs should be modified to use only one of two semaphores due to the fact that it may incur to other serious faults, such as deadlocks. In the table, CodeSonar also reports a false positive for SV-N02 similar to AR653, while it does not report a false positive for SP-N02 since the static analysis tool cannot trace pointer variables in the programs.

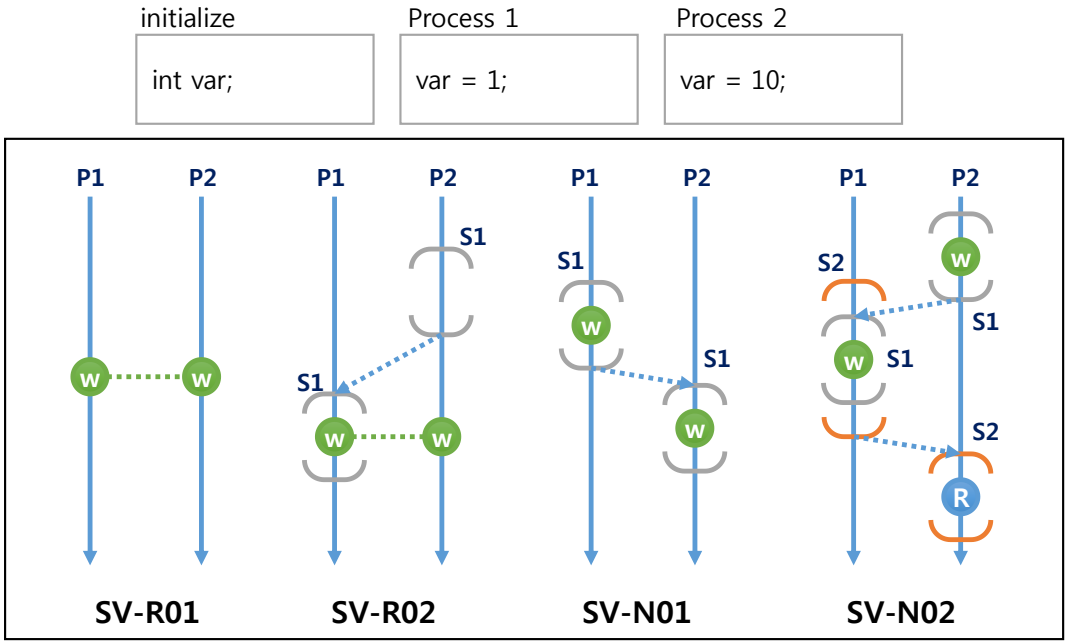


Figure 6. The Graph of Synthetic ARINC 653 Applications Using the Shared Variables

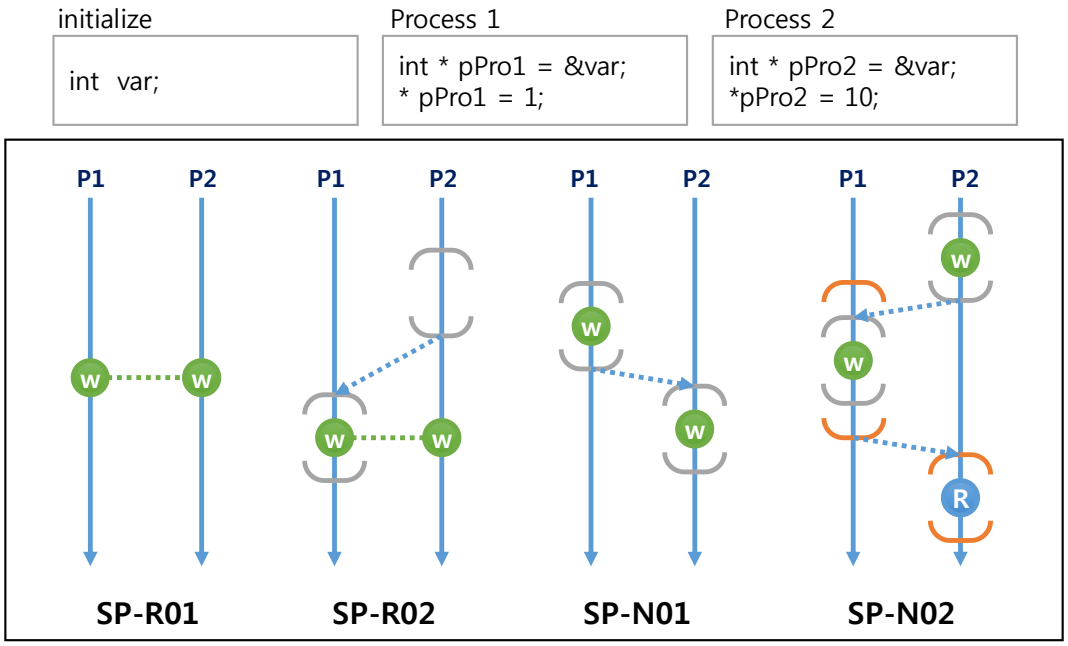


Figure 7. The Graph of Synthetic ARINC 653 Applications Using the Shared Pointers

Table 1. The Detection Result of Synthetic ARINC 653 Applications

Type	Race	Number	Program	Detection Results	
				AR653	CodeSonar
Shared Variable	○	01	SV-R01	○	○
	○	02	SV-R02	○	○
	×	01	SV-N01	×	×
	×	02	SV-N02	○	○
Shared Pointer	○	01	SP-R01	○	×
	○	02	SP-R02	○	×
	×	01	SP-N01	×	×
	×	02	SP-N02	○	×

5. Conclusion

In the ARINC 653applications for integrated modular avionics (IMA), incorrect use of semaphore must be detected because it can cause atomicity races, one of concurrency errors and harm the reliability of the system. This paper presented a tool for atomicity race detection, called AR653 that consists of AR653 Monitor and AR653 Detector. The AR653 Monitor collects monitored information such as processes, semaphores, and accesses to shared resources, while AR653 Detector reports atomicity races by checking a locking discipline which stipulates that any two different processes access a shared resource with a common semaphore.

To evaluate the precision of AR653, we compared it with CodeSonar. The comparison results using synthetic programs show that AR653 precisely detects atomicity races in ARINC 653 applications using shared variables and shared pointers, while CodeSonar using the static analysis is imprecise for the applications using shared pointers due to the

fact that it cannot trace pointer variables. Therefore, AR653 is a useful tool to debug atomicity races in applications for avionics. However, it produces high overhead because the tool consumes a lot of resources of the system during an application execution. Future work includes improving our detection tool to remove false positive problem and to reduce the heavy run-time overhead. Future works would be needed to focus on reducing false positives reported by locking discipline, and adopting the AR653 Detector to Happens-before or Hybrid detection technique. Also more works would be necessary on reducing overhead during detection so that it would be used in the actual airplanes for detecting atomicity races, leading to realization of fault tolerance.

Acknowledgments

This work was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2014R1A1A2060082), and also was supported by Development Fund Foundation, Gyeongsang National University, 2015.

This paper is a revised and expanded version of a paper entitled "Detecting Atomicity Races in ARINC 653 Applications" presented at GDC 2015 conference, November 25, 2015, Jeju, Korea.

References

[1] Airlines electronic engineering committee (AEEC), "Avionics Application Software Standard Interface - ARINC Specification 653 – part 1. (Supplement 2- Required Services)", ARINC inc., **(2006)**.
[2] S. Santos, J. Rufino, T. Schoofs, C. Tatibana and J. Windsor, "A portable ARINC 653 standard interface", Proceedings of the 27th Digital Avionics Systems Conference, St. Paul, MN, **(2008)** October 26-30.
[3] S. H. VanderLeest, "ARINC 653 hypervisor", Proceedings of the 29th Digital Avionics Systems Conference, Salt Lake city, UT, **(2010)** October 3-7.
[4] S. Han, and H. Jin, "Full virtualization based ARINC 653 partitioning", Proceedings of the 30th Digital Avionics Systems Conference, Seattle, WA, **(2011)** October 16-20.
[5] J. Rufino, S. Filipe, M. Coutinho, S. Santos and J. Windsor, "ARINC 653 interface in RTEMS", Proceedings of the Data Systems In Aerospace, Napoli, Italy, **(2007)** June.
[6] P. J. Prisaznuk, "ARINC 653 role in Integrated Modular Avionics (IMA)", Proceedings of the 27th Digital Avionics Systems Conference, St. Paul, MN, **(2008)** October 26-30.
[7] P. J. Prisaznuk, "Integrated modular avionics", Proceedings of the IEEE 1992 National, Dayton, OH , **(1992)** May 18-22.
[8] C. B. Watkins and R. Walter, "Transitioning from federated avionics architectures to Integrated Modular Avionics", Proceedings of the 26th Digital Avionics Systems Conference, Dallas, TX, **(2007)** October 21-25.
[9] Y.-H. Lee, D. Kim, M. Younis and J. Zhou, "Scheduling tool and algorithm for integrated modular avionics systems", Proceedings of the 19th Digital Avionics Systems Conference, Philadelphia, PA, **(2000)** October 7-13.
[10] T. Schoofs, S. Santos, C. Tatibana and J. Anjos, "An integrated modular avionics development environment", Proceedings of the 28th Digital Avionics Systems Conference, Orlando, FL, **(2009)** October 23-29.
[11] Netzer, Robert H. B. and Miller, Barton P., "What are race conditions?: Some issues and formalizations", Proceeding of Letters on Programming Languages and Systems, New York, USA, **(1992)** March.
[12] E. Farchi, Y. Nir, and S. Ur, "Concurrent bug patterns and how to test them", Proceeding of the 17th Parallel and Distributed Processing Symposium, **(2003)** April 22-26.
[13] U. Banerjee, B. Bliss, Z. Ma and P. Petersen, "A theory of data race detection", Proceedings of International Symposium on Software Testing and Analysis 2006, Portland, ME, **(2006)** July 17-20.
[14] GrammaTech, CodeSonar http://www.grammatech.com/codesonar, **(2015)**.
[15] D. Callahan and J. Sublok, "Static analysis of low-level synchronization", Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging, New York, USA, **(1988)** November 01.
[16] C. E. McDowell, "A practical algorithm for static analysis of parallel programs", Journal of Parallel and Distributed Computing, vol. 6, no. 3, **(1989)**, pp. 515-536.

[17] D. Engler and Ken Ashcraft, "RacerX: effective, static detection of race conditions and deadlocks", Proceedings of the 19th ACM Symposium on Operating Systems Principles, New York, USA, **(2003)** October 19-22.
[18] H. Patil, C. Pereira, M. Stallcup, G. Lueck and Cownie, "Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs", Proceedings of the 8th annual IEEE/ACM International Symposium on Code Generation and Optimization, Toronto, Canada, **(2010)**, April 24-28.
[19] A. R. Bernat and B. P. Miller, "Anywhere, any-time binary instrumentation", Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools, Szeged, Hungary, **(2011)**, September 05-09.
[20] E. Pozniansky and A. Schuster, "Efficient on-the-fly data race detection in multithreaded C++ programs", Proceeding of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, USA, **(2003)**, June 11-13.
[21] A. Jannesari and W. F. Tichy, "On-the-fly race detection in multi-threaded programs", Proceedings of the 6th workshop on Parallel and Distributed Systems: testing, analysis, and debugging, Seattle, USA, **(2008)**, July 20-24.
[22] O.-K. Ha, G. M. Tchamgoue, J.-B. Suh and Y.-K. Jun, "On-the-fly Healing of Race Conditions in ARINC 653 Flight Software", Proceedings of the 29th Digital Avionics Systems Conference, Salt Lake City, UT, **(2010)** October 3-7.
[23] G. M. Tchamgoue, O.-K. Ha, K.-H. Kim and Y.-K. Jun, "A Framework for On-the-fly Race Healing in ARINC 653 Applications", International Journal of Hybrid Information Technology, vol. 4, no. 2, **(2011)**, pp.1-12.
[24] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs", ACM Transactions on Computer Systems, vol. 15, **(1997)**, pp. 391-411.
[25] T. Elmas, S. Qadeer and S. Tasiran, "Goldilocks: a race and transaction-aware java runtime", Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, San Diego, USA, **(2007)**, June 11-13.
[26] O.-K. Ha and Y.-K. Jun, "An Efficient Algorithm for On-the-Fly Data Race Detection Using an Epoch-Based Technique", Scientific Programming, vol. 2015, **(2009)**, pp. 1-14.
[27] C. Flanagan and S. N. Freund, "FastTrack: efficient and precise dynamic race detection", Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, Dublin, Ireland, **(2009)**, June 15-21.

Authors

Mr. Eu-Teum Choi received the BS degree in Department of Applied Life Chemistry, and the MS degree in Informatics from Gyeongsang National University (GNU), South Korea. He is currently enrolled PhD degree from the department of Informatics in GNU and works Aero Master Corporation, South Korea. He worked as an engineer of IT department in Korea industry for two years. His research interests include distributed programming and its debugging, embedded system programs for avionics, and dependable systems.

Dr. Ok-Kyoon Ha received the BS degree in Computer Science under the Bachelor's Degree Examination Law for Self-Education from National Institute for Lifelong Education, and the MS and PhD degree in the department of Informatics from Gyeongsang National University (GNU), South Korea. He is now a research professor in GNU. He worked as the manager of IT department in Korea industry for several years. His research interests including parallel/distributed programming, software testing and debugging, embedded system programs, dependable software, and software development activities for avionics. Dr. Ha is a member of Korean Institute of Information Technology (KIIT), Korea Institute of Information Scientist and Engineers (KIISE), and Institute of Embedded Engineering of Korea (IEMEK).

Dr. Yong-Kee Jun received the BS degree in Computer Engineering from Kyungpook National University, and the MS and PhD degree in Computer Science from Seoul National University. He is now a full professor in the Department of Informatics, Gyeongsang National University, where he had served as the first director of GNU Research Institute of Computer and Information Communication (RICIC), and as the first operating director of GNU Virtual College. He is now the head of GNU Computer Science Division and the director of the GNU Embedded Software Center for Avionics (GESCA), a national IT Research Center (ITRC) in South Korea. As a scholar, he has produced both domestic and international publications developed by some professional interests including parallel/distributed computing, embedded systems, and systems software. Prof. Jun is a member of Association for Computing Machinery (ACM) and IEEE Computer Society.