Research Report

False-negative-free Interrupt Race Condition Detection Using Multiple Static Code Analysis Methods

Yutaka Inamori and Nobuyuki Yamada

Report received on Apr. 11, 2012

ABSTRACT Interrupt handlers are used in vehicle control programs for high responsiveness but are a possible cause of data races. The present paper describes a detection method for interrupt race conditions that produces no false negatives and a smaller number of false positives. The proposed method is characterized by a mechanism whereby the masses of false positives are sifted through using five types of static code analysis methods that are free from false negatives. One of these methods identifies possible interrupt states for every statement using abstract interpretation and determines the possibility of interruption in accessing a shared memory. Another of these methods uses the model checker SPIN to verify the non-existence of an execution path creating a race condition.

KEYWORDS Static Code Analysis, Interrupt Race, Data Race, Model Checking, Automotive Software, Code Inspection

1. Introduction

Interrupt handlers are used in automotive software for the purpose of ensuring high responsiveness. However, they may cause data races (interrupt races) that are difficult to detect by software tests. Therefore, interrupt races are to be prevented from the standpoint of high reliability of automotive software. An interrupt race is a rare behavior that occurs only when an interrupt handler is called within a specific narrow slice of time. This is why interrupt races are rarely found using test technologies and must be found by other means, such as design review or code inspection, in order to assure high reliability.

However, in recent years, the larger and more complicated automotive software becomes, the longer it takes to check the absence of interrupt races. The process of the checking has room for improvement and we focused on the process of code inspection, in which, after code implementation, a number of people look into whether a source code has flaws that cause interrupt races. This takes a great deal of time because it is difficult to fully examine the control flow of a program, including the interrupt handlers.

Therefore, we developed a detection method for interrupt race conditions in order to automate to a large degree the flaw inspection process of interrupt races. The following are the requirements of the method:

- Detection of interrupt races without omissions in order to leave no flaws on interrupt races in the test process.
- Reduction of false positives[*1] in order to reduce the number of worker-hours required for manual flaw inspection.
- Minimization of manual works, for example the insertion of assertion codes into original source code.

In the present study, we developed a method in which the masses of false positives are sifted through multiple stages of static code analysis methods that detect races without producing false negatives. Control flow analysis, pointer analysis, abstract interpretation, and model checking are adapted as analysis methods in the order of increasing analysis time in order to shorten the total analysis time.

The remainder of the present paper is organized as follows. Section 2 defines the problem to be solved in the present study, and Section 3 describes research related to data race detection with a focus on static code analyses. Section 4 outlines the proposed approach, and Section 5 explains the details of the proposed approach. Section 6 introduces a newly developed system that is used to demonstrate the

[*1] A false positive means that part of a program without flaws is detected incorrectly, and so does not influence the quality of the program.

usefulness of applying the proposed approach to automotive software production.

2. Definition of the Problem

This section describes the object of analysis of the proposed method. In addition, the definition of the problem to be solved is presented, and related terminologies are explained.

2. 1 Object of Analysis

The object of analysis is a C program that is executed on a unit processor without an OS and is composed of a main program and interrupt handlers. The main program and interrupt handlers send and receive data using shared memories.

2. 2 Definition of Interrupt Race

An interrupt race is a situation in which a main program and an interrupt handler (or two interrupt handlers) cause a data race. In such a case, both A and B below occur (**Fig. 1**):

A While a low-priority function accesses a shared memory, an interrupt occurs and a high-priority function accesses the same shared memory for a short time.
B At least one of the above three accesses is a write access.

In A, low-priority function refers to a function that is called by the interrupted task, and high-priority function refers to a function that is called by the interrupting task. Here, short time is defines as the interval between the start time and the finish time of one function. The same shared memory means that the access memories of the above three accord with one another bit-by-bit.

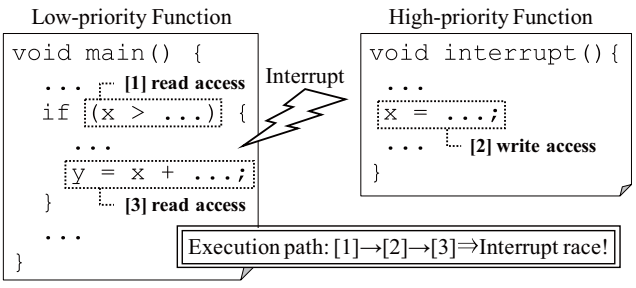

Fig. 1 Example of interrupt race.

In the example of Fig. 1, the value of x, which is first read in branch condition expression [1], is overwritten by sentence [2] in the high-priority function. In this case, sentence [3] reads a value different from that in branch condition expression [1]. If this behavior is unintended, the program is to be corrected.

2. 3 Location of Interrupt Race

A location of an interrupt race or a possible interrupt race is defined as a triple of the following three sentences:

- A sentence in which the first access to a shared memory is performed in a low-priority function ([1] in Fig. 1).
- A sentence in which an access to the same shared memory is performed in the high-priority function ([2] in Fig. 1).
- A sentence in which the second access to the shared memory is performed in the low-priority function ([3] in Fig. 1).

Each sentence is represented by the name of the function to which it belongs, a line number, and the name of the task (the main function or an interrupt handler) that calls the function directly or indirectly.

2. 4 Detection of Interrupt Races

In the present study, the proposed method requires the locations of the interrupt races to be detected with no omissions. In other words, if a program includes flaws that cause interrupt races, all of the locations of interrupt races should be found. At the same time, the number of false positives (mistakenly detecting the locations at which interrupt races do not occur) should be as small as possible.

3. Related Research

Including interrupt races, research on data races consists of dynamic analysis (accompanying the execution of a program or simulation) and static analysis (using static code analysis). Dynamic analysis[(1)] focuses on finding as many flaws as possible, so that less attention is paid to false negatives.

In contrast, a number of static analyses can detect data races with no omissions, although the number of

false positives tends to increase. Research into static analysis includes nesC[2] and LOCKSMITH.[3] nesC is an extension of C that can detect flaws in which accesses to a shared memory are performed outside the atomic region. LOCKSMITH verifies whether the accesses to a shared memory are protected in a consistent manner by using type inference. However, although these two methods verify the existence of the specific access protection (atomic regions or locks), these methods do not analyze the other access protection schemes (for example, access control by mode variables), which leads to numerous false-positives.

4. Overview of the Proposed Interrupt Race Detection Method

In order to detect interrupt races with no omissions, the following approach was adopted. First, the locations of possible interrupt races are enumerated exhaustively and a list of candidate locations of possible interrupt races is composed. Then, each location is judged from various standpoints to determine whether it is possible to trigger an interrupt race. Locations that are judged to be race-free are eliminated from the list, and the final list shows the locations of possible interrupt races.

In order to exhaustively enumerate the locations of possible interrupt races, it is necessary only to enumerate the global variables and static variables that each task (a main program or an interrupt handler) accesses and to count all of the pairs of tasks that both access the same memory. As a result, a list called an itemized list to be judged is generated automatically. Each item on the list contains information about the location of a possible interrupt race, types of memory accesses (read/write), and the name of a shared variable (**Fig. 2**).

For each enumerated item, whether an interrupt race can or may occur is judged in a step-by-step manner with the following judgment criteria (1 to 5 below). In order to prevent false-negatives, all of the judgments are to determine whether an interrupt race can or may occur and narrow down items that may have interrupt races. In order to shorten the analysis time, the judgments are arranged in order from shortest to longest.

1. **Judgment of access pattern**: Do the type and number of accesses satisfy the condition for interrupt race?
2. **Judgment of simultaneity of accesses**: Do the two accesses ([1] and [3] in Fig. 1) in a low-priority function exist on the same control flow?
3. **Judgment of interrupt state**: Are the two accesses ([1] and [3]) in a low-priority function protected by a DI (Disable Interrupts) instruction?
4. **Judgment of agreement of access memory**: Do both a low-priority function and a high-priority function access the same memory by the bit?
5. **Judgment of existence of execution path**: Does an execution path (through [1], [2], and [3]) that causes interrupt races exist?

The 4th judgment uses pointer analysis, and the 5th judgment uses model checking, both of which are placed in the latter part of the judgments because of their longer analysis time. In the next section, each judgment will be explained in detail.

5. Details of the Proposed Interrupt Race Detection Method

5. 1 Judgment of Access Pattern

Whether the types and numbers of accesses fit the condition of the interrupt race is judged, i.e., two accesses in a low-priority function and one access in a high-priority function. The judgment can be performed with the information in the itemized list.

Item No.	Shared variable	Low-priority function							High-priority function					Judgment results
		Function name	Access [1]		Access [3]		Caller		Function name	Access [2]		Caller		No race/ Possible race
			Line No.	Access type	Line No.	Access type	Process name	Priority		Line No.	Access type	Process name	Priority	
1	x	fun1	110	read	113	read	main	0	sub1	15	write	int1	3	?
2	y	sub1	53	write	54	write	int1	3	sub2	22	read	int2	5	?
…	…	…	…		…		…	…	…	…		…	…	…

Fig. 2 Itemized list to be judged.

5. 2 Judgment on Simultaneity of Accesses

In order to judge whether two accesses ([1] and [3] in Fig. 1) to a shared memory occur while a low-priority function is executed once, whether the two statements in which the two accesses are performed are on the same control flow is examined automatically. Based on the control flow diagram generated from C source code, reachable statements from each statement and to each statement are analyzed, and whether the reachable statements from statement [1] include statement [2], or vice versa, is judged.

5. 3 Judgment on Interrupt State

5. 3. 1 Abstract Concept

Judgment on the interrupt state is performed based on whether the interrupt state disables all of the paths between the two accesses ([1] and [3] in Fig. 1). If the judgment is true, accesses [1] and [3] are never interrupted, i.e., interrupt races never occur.

The following points are to be considered in realizing the judgment:

a. The interrupt state changes according to the execution path for the case in which an interrupt instruction, DI (Disable Interrupts) or EI (Enable Interrupts), exists in a branching statement (if-statement or switch-statement) (left-hand side of **Fig. 3**).
b. The interrupt state in a loop statement is dependent on not only the interrupt state of the former statement of the loop statement, but also on the interrupt instructions in the loop statement (center of Fig. 3).
c. The interrupt state is determined not only by interrupt instruments on the same function, but also by those on caller functions and callee functions (right-hand side of Fig. 3).

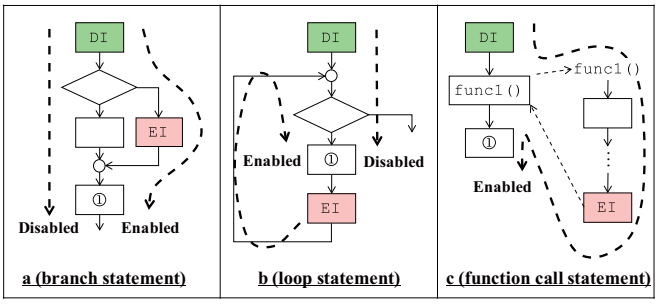

Fig. 3 Relationship between control flow and interrupt state.

The method of searching interrupt instruments on the upstream execution paths is anticipated to be very complicated. We used abstract interpretation and developed a method of identifying the interrupt states of all of the statements in a C program. Abstract interpretation [4] is a static code analysis methods that abstracts operations and operands that show the execution states of a program and extracts beneficial information from the execution results. The present approach defines abstract operations and abstract domains on interrupt states of pre-execution and post-execution of a statement.

It is necessary to analyze the interrupt state while carefully considering caller-callee relationships.[5] In the present study, we developed a method by which to analyze each function, which prevents the algorithm from becoming too complex. This method is show below.

5. 3. 2 Abstract Domains and Operations Expressing the Interrupt State

We define the abstract domain S_{ifg}, which shows an interrupt state at a certain point, as

$$S_{ifg} = (low_{ifg}, high_{ifg}, inh_{ifg}), \cdots\cdots\cdots\cdots\cdots (1)$$

where low_{ifg} and $high_{ifg}$ show the range of possible interrupt states, each of which is set to *e* (*enabled state*), *d* (*disabled state*), or *nil* (*undefined*). For example, $low_{ifg} = e$ and $high_{ifg} = d$ indicate that an interrupt state of a statement becomes enabled or disabled. Moreover, inh_{ifg} is set to be *true* or *false*. If inh_{ifg} is set to be *true*, then the interrupt state of a statement is dependent on the interrupt states of the function call statements that call the function, including the statement. In the interrupt state of the pre-execution of the first statement in a function, $inh_{ifg} = true$ without fail. For the cases in which $inh_{ifg} = true$, low_{ifg}, and $high_{ifg}$ may not express the actual upper range and must be calculated using an application function (4), which is described later herein, in order to add information about the interrupt states of the caller functions. The calculation of the analysis

guaranteed to stop in finite steps, because the abstract domain S_{ifg} forms a complete lattice (**Fig. 4**) and consists of finite elements.

The interrupt state at pre-execution of a statement (hereinafter pre-execution state) is calculated as follows. If an assign statement does not include a function call, its pre-execution state is equivalent to its post-execution state, and if a statement is DI (EI), its post-execution state never fails to change into d (e). The interrupt state at the confluence in a branching statement or a loop statement is calculated using the following OR-operation:

$$\begin{aligned}&(low_1, high_1, inh_1)\ OR_{ifg}\ (low_2, high_2, inh_2)\\&= (\min_{ifg}(low_1, low_2), \max_{ifg}(high_1, high_2), inh_1 \vee inh_2)\end{aligned} \quad \cdots\cdots\cdots\cdots\cdots (2)$$

The operation $\min_{ifg}$ ($\max_{ifg}$) calculates the minimum value based on the premise of the relationship $e < d$.

If a statement includes a function call, its post-execution state is calculated in reference to the state of the callee function, because the post-execution state of a statement with a function call is equivalent to the post-execution state of the final statement in the callee function. If $inh_{ifg} = true$ in the referred state, then the following application-function is used:

$$\begin{aligned}&(low_1, high_1, true) \lhd_{ifg} (low_2, high_2, inh_2)\\&= (low_1, high_1, false)\ OR_{ifg}\ (low_2, high_2, inh_2)\end{aligned} \quad \cdots\cdots\cdots\cdots\cdots (3)$$

which means that the referred state inherits the pre-execution state of itself (the function call statement). The first term of the left-hand side of Eq. (3) shows the post-execution state of the last statement in the callee function, and the second term of the left-hand side of Eq. (3) is the pre-execution state of itself (the function call statement). In the case of a function call with a function pointer, the function call is regarded as a branching and confluence of multiple functions and its post-execution state is calculated using the OR-operation of Eq. (2) on the post-execution states of the last statements in all possible callee functions.

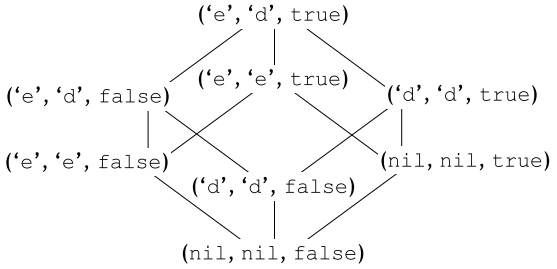

Fig. 4 Abstract domain in the interrupt state.

The pre-execution state of a statement is taken over from the post-execution state of its previous statement.

In the analysis of the interrupt permission level, which specifies which interrupt handlers are enabled, it is necessary only to perform calculation in a similar manner, except that the range of possible interrupt permission level is set using low_{ipl} and $high_{ipl}$.

5. 3. 3 Judgment of Interrupt State

After calculating the pre-execution state and post-execution state of each statement in every function, the interrupt state of the target location is analyzed. Using the following condition, whether the two accesses ([1] and [3] in Fig. 1) in a low-priority function are interrupted is judged:

- $low_{ifg} = high_{ifg} = d$ on the pre-execution state of [1] and no EI exists between [1] and [3].

In this regard, however, when $inh_{ifg} = true$ on the pre-execution state of [1], the actual pre-execution state must be preliminarily calculated with the application operation (3).

5. 4 Judgment on Accordance of Access Memory

Whether three accesses ([1], [2], and [3] in Fig. 1) to shared memories are in bit-by-bit accordance with one another is judged. In order to analyze possible access memories and bitwise offsets for all C language expressions (add-subtract expression of a pointer variable, cast expression of pointer type, and so on), we constructed a memory model that holds bitwise information on variables and developed a pointer analysis method that can treat bitwise access information. The details are omitted due to space limitations.

5. 5 Judgment on the Existence of the Execution Path

Whether an execution path that causes an interrupt race exists is judged by model checking. An execution

path that causes an interrupt race passes through [1], [2], and [3] in Fig. 1. The path includes the occurrence of a high-priority process and the disruption of a low-priority process.

In the present study, the SPIN [6] model checker is used to model the behaviors of the interrupt handlers. SPIN is good at modeling the behaviors of concurrent processes and is applicable for verifying the occurrence of interrupt races in all cases of interrupt timing. After program slicing is performed in order to enhance scalability, the sliced C program is automatically translated into a promela (model description language for SPIN) code. Then, an assertion statement is inserted in order to verify the execution path of the interrupt race.

The following explains a method of modeling multiple interrupts and program slicing.

5. 5. 1 Modeling of Multiple Interrupts

The outline of the method for modeling multiple interrupts in a promela code is given as follows. Each interrupt handler is modeled as a process with the "provided clause", which refers to the following execution condition:

- Interrupt is permitted (activation condition) or a process itself is currently executing (continual condition).

An interrupt state and execution state of each process are modeled by the global variable in the promela. The contents of an interrupt handler are modeled as an infinite loop, and the execution state is set to "currently executing" at the top of the infinite loop and "finished" at the bottom of the infinite loop. These settings prevent an interrupt process from stopping midstream and returning to an interrupted process. A stack is laid on to model the behaviors of multiple interrupts. At the top of the infinite loop on each process, the information on the process itself is pushed onto the stack, and at the bottom, the information is popped off the stack. These operations enable modeling of the behavior of multiple interrupts.

5. 5. 2 Program Slicing for Execution Path Analysis

If a C source code with several tens of thousands of lines is translated into a promela code, SPIN cannot search all of the state space of the code and is aborted. Therefore, the proposed method reduces the amount of code by means of program slicing in terms of execution path analysis.

What is required for the program slicing is to cut off of the statements which are out of relation to the execution condition of the three accesses. The following shows the basic algorithm that we have developed (**Fig. 5**):

Step 1: Three accesses ([1], [2], and [3] in Fig. 1) and interrupt instructions (DI, EI, and so on) are defined as necessary codes.
Step 2: All of the branch conditions that are inevitably passed from the top of the function to necessary codes are added to necessary codes.
Step 3: For the branch conditions that are added in Step 2, all of the variables are extracted, and all of the assign statements to these variables are added to necessary codes.
Step 4: For the assign statements that are obtained in

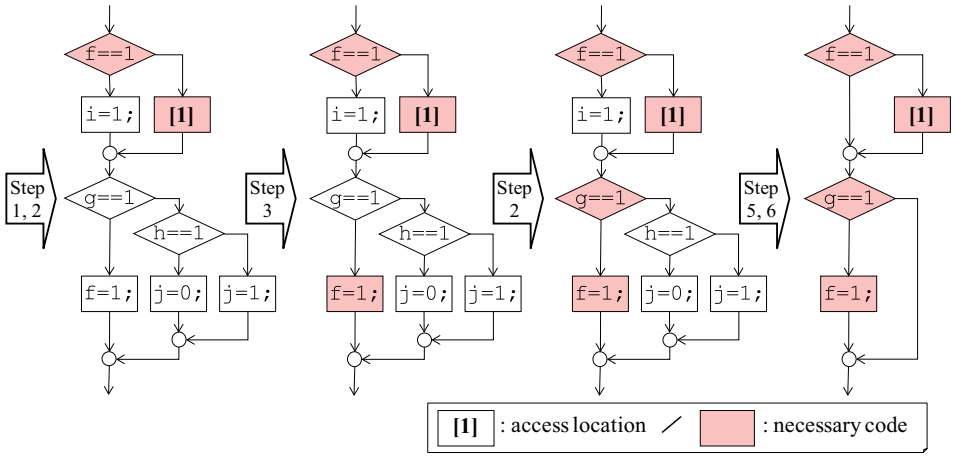

Fig. 5 Program slicing for execution path analysis.

Step 3, all of the variables shown on the right-hand side of the statements are extracted, and all of the assign statements to these variables are added to necessary codes.

Step 5: If necessary codes are added in Step 3 or Step 4, return to Step 2, otherwise go to Step 6.

Step 6: All codes except the necessary codes are cut off.

The basic algorithm does not work well because it causes the chain of necessary codes in Steps 3 and 4. Therefore, we improved the basic algorithm, which has the difference in Step 2 (the improved algorithm contains Step 2' instead of Step 2):

Step 2': Out of all of the branch conditions that are inevitably passed from the top of the function to the necessary codes, the branch conditions that satisfy the specific condition are added to the necessary codes.

By dropping part of the branching conditions from the necessary codes, the chain of necessary codes in Steps 3 and 4 is kept small. However, since the dropped branching conditions are modeled as non-deterministic, this generates impossible execution paths (over-approximation) and causes false-positives. The specific condition we adopt here is that "a branching condition has only bit variables". Since most bit variables are assigned a constant (0 or 1), the chain of necessary codes is reduced. Furthermore, the number of states in model checking is reduced because the promela code has only bit variables.

6. Development and Evaluation of the Proposed System

Figure 6 shows a prototype system of interrupt race detection. The preprocessing part generates parse trees, control flow graphs, and a memory model. The judging part performs judgment using various types of static code analyses. Both parts are implemented with Ruby. For model checking, the part that generates codes for SPIN and analyses the results from SPIN is also implemented with Ruby and calls SPIN automatically. We use the database generated by the Understand, Scientific Toolworks' code analysis tool, which contains information about lexemes, symbols, and types of references.

This system was developed for application to practical applications and was targeted on the case in which a C source code for a customer product, the scale of which is several tens of thousands of lines, and has a main process and four interrupt processes. We set the development goals for this system so that the system is false-positive-free and the rate of automation (percentage of automatically judged race-free items among all items) is equal to or greater than 90%.

The largest problem to solve was the scalability of the judgment on the existence of the execution path. Since it is impossible to analyze a large code using SPIN, the key issue was to reduce the number of states. In addition to the slicing method shown in Section 5.5.2, we developed algorithms to cut off the interrupt processes and interrupt instruments that are irrelevant to execution paths of interrupt race (not shown herein). As a result, a source code could be reduced to from one-twentieth to one-fiftieth the original C source code, and the number of abnormal terminations could be reduced to 15% among all of the items that were needed to perform model checking.

Moreover, in order to reduce the number of model checking operations, we increased the precision of the former judgments, especially the judgment on the interrupt state. For example, an early algorithm of the judgment was as follows:

- $low_{ifg} = high_{ifg} = d$ in the pre-execution state of [1] and $low_{ifg} = high_{ifg} = d$ in all of the statements between [1] and [3].

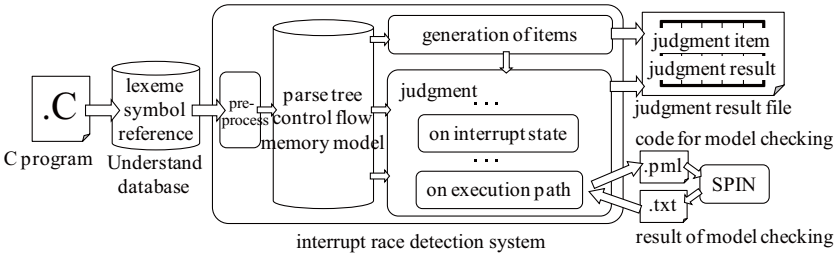

Fig. 6 System structure.

Along with the algorithm presented in Section 5.3.3, the above algorithm is also intuitive. Thus, the precisions of the two algorithms are thought to be equal. However, as a result of the detailed analysis of the two algorithms, false-positives are found to exist, as shown in **Fig. 7**. The interrupt states that are identified in the proposed method using abstract interpretation are the possible states in consideration of all of the execution paths. On the other hand, in the case of examining partial paths (paths from [1] to [3] in Fig. 7) the algorithm in Section 5.3.3, which scans interrupt instruments between [1] and [3], turned out to be more precise. As a result of the other improvements on the judgment, the number of false-positives approached zero near the locations at which interrupt instructions prevent interrupt races. Thus, we succeeded in reducing the number of model checking operations.

The following are the results of the case study:

- False-negatives: none
- Rate of automation: 94.3% (better than the target)
- Manual works: only to alter the part of the code that is dependent on a microcomputer or a compiler

In relation to the rate of automation, approximately 90% of the causes of false-positives were over-approximation in model checking and abnormal termination of model checking caused by insufficient memory.[*2] In order to prevent over-approximation, the improvement of Step 2' in Section 5.5.2 (a selection method of branching conditions) or the introduction of Counter Example-Guided Abstraction Refinement (CEGAR) [7] can be considered, but as the number of variables in a model increases, the number of states of a model increases dramatically. When the proposed method is applied to cases larger than the above case, the rate of abnormal terminations is expected to increase. Therefore, there are still challenges to reducing a state space, for example, by model improvement, problem partitioning, and so on. For reference, example data on state numbers in model checking are presented in **Table 1**.

Thus, with the result being sufficient, in the future, the system will be evaluated using several other cases.

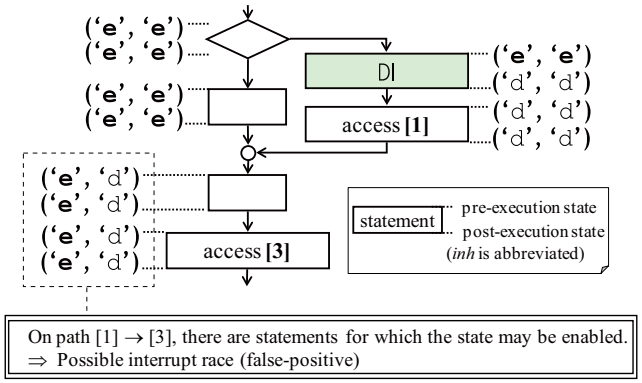

Fig. 7 Example of a false-positive in the judgment of the interrupt state before improvement.

[*2] The remaining causes were of insufficient precision of pointer analysis.

7. Conclusions and Future Research

The present paper introduced a method for the detection of defects that may cause interrupt races by applying static code analysis methods for C source codes. The proposed method judges whether accesses to a shared memory in a low-priority function are protected by means of identifying the interrupt state through abstract interpretation. Then, for non-protected accesses the existence of an execution path that causes an interrupt race is verified by model checking. This method is applicable for C programs of approximately 100,000 lines, and its practicality is ascertained.

As automotive software becomes larger, the number of worker-hours required to ensure high reliability, by software inspection, software testing, and so on, increases. In such a situation, the proposed method will contribute to shortening of the development period. In the future, we will attempt to solve the problems associated with incorrect timing (including interrupt race) through new design and analysis methods.

Through the courtesy and with the permission of Information Processing Society of Japan this paper is translated in English and reprinted in full from IPSJ Symposium Series Vol.2010, No.10 (2010), pp.113-118, Inamori, Y. and Yamada, N., "*Seiteki Kodo Kaiseki ni yoru Kenshutsumore no nai Warikomikansho Kenshutsu Shuho no Kaihatsu*" (in Japanese).

Table 1 State numbers in model checking.

Example No.	Line No. of model	No. of bit variables	State No.
1	2,559 lines	1	90,000
2	2,572 lines	5	90,000
3	2,659 lines	7	1 million
4	2,653 lines	8	2 million

References

(1) Higashi, M., Yamamoto, T., et al., "An Effective Method to Control Interrupt Handler for Data Race Detection", *Proceedings of AST'10* (2010), pp.79-86, ACM.

(2) Gay, D., Levis, P., et al., "The nesC Language: A Holistic Approach to Networked Embedded Systems", *ACM SIGPLAN Notices* (*Proceedings of PLDI'03*), Vol.38 No.5 (2003), pp.1-11.

(3) Pratikakis, P., Foster, J., et al., "LOCKSMITH: Context-sensitive Correlation Analysis for Race Detection", *ACM SIGPLAN Notices* (*Proceedings of PLDI'06*), Vol.41 No.6 (2006), pp.320-331.

(4) Cousot, P. and Cousot, R., "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints", *Proceedings of the 4th ACM POPL* (1977), pp.238-252, ACM.

(5) Sekiguchi, T., "A Practical Pointer Analysis for C Language", *Computer Software*, Vol.21, No.6 (2004), pp.34-49, JSSST.

(6) Holzmann, G. J., *The Spin Model Checker: Primer and Reference Manual* (2004), Addison-Wesley.

(7) Clarke, E., Grumberg, O., et al., "Counterexample-guided Abstraction Refinement", *Proceedings of CAV'00*, Vol.1855 of LNCS (2000), pp.154-169, Springer.

Yutaka Inamori

Research Fields:
- Reliability of Automotive Software
- Automatic Detection of Software Flaws
- Software Verification with Model Checking

Academic Society:
- Information Processing Society of Japan

Award:
- IPSJ Yamashita SIG Research Award, 2012

Nobuyuki Yamada*

Research Field:
- Working on Streamlining of Tool Environment for Effective Development of ECU (Electronic Control Unit) Software

*AISIN SEIKI CO., LTD.

