

Verification of Flag Usage Patterns by Static Analysis Techniques

Amey Zare and Advaita Datar
Tata Consultancy Services Ltd.

Mitsuhiko Kikuchi, Satoshi Ichikawa, and Miwako Hasegawa
Nissan Motor Co., Ltd.

Shigenori Tsunekado
Nissan Technology Co., Ltd.

ABSTRACT

A flag is a global boolean variable used to achieve synchronization between various tasks of an embedded system. An application implementing flags performs actions or events based on the value of the flags. If flag variables are not implemented properly, certain synchronization related issues can arise which can lead to unexpected behavior or failure of the underlying system. In this paper, we present an automated verification technique to identify and verify flag usage patterns at an early stage of code development. We propose a two-step approach which consists of: a. identification of all potential flag variables and b. verification of flag usage patterns against predefined set of rules. The results of our experiment demonstrate that the proposed approach reduces the cost and complexity of the flag review process by almost 70%.

CITATION: Zare, A., Datar, A., Kikuchi, M., Ichikawa, S. et al., "Verification of Flag Usage Patterns by Static Analysis Techniques," *SAE Int. J. Passeng. Cars – Electron. Electr. Syst.* 7(1):2014, doi:10.4271/2014-01-0180.

INTRODUCTION

The use of embedded software in safety-critical systems [1, 2] has prompted the evolution of Verification & Validation (V&V) techniques [3], mainly due to risks involved in the software's use. Additionally, embedded applications are also expected to comply with certain safety standards (like ISO 26262) to avoid or control failures and mitigate their effects [4]. V&V techniques check the correctness of intended functionalities and ensure that the system operates/behaves as per expectations. To keep up with the increase in complexities and accelerate the time to market, software systems are usually implemented by multiple developers in distinct modules or sections of code. A basic requirement of such code environments is the use of an effective medium for communication between various sections of the code [5]. This requirement is implemented by either an external medium/system or by the software code implementation which is a part of the underlying system. Flag variables form the base of one such technique which involves use of variables for communication and synchronization between various parts of the code [6].

A flag variable, *Flag* has two distinct values throughout its lifetime, in general 1 and 0, which represent whether the flag is *set* (*Flag*=1) or *clear* (*Flag*=0). Generally a flag is *set* or *cleared* in a particular section of code and the state of this flag is checked in another section before performing certain operations.

We refer to these sections of the code as "Synchronizing Units" that could be part of single task/interrupt service routine (ISR) or part of multiple tasks/ISRs or a combination of both. We explain a typical implementation of flags by using the example shown in [Figure 1](#). In this example, *mainFunc* is *setting* the *Flag* to 1 only when *Oper1* is performed successfully. The same flag is then *compared* in *func01* for taking a decision as to which operation needs to be performed next. Hence, *Flag* acts as a mean of communication between *mainFunc* and *func01*.

In modern day embedded systems, flag variables are used exhaustively for synchronizing various code sections responsible for performing critical functionalities [2, 5]. This makes the verification of flag usage patterns an important aspect for ensuring the safety and reliability of embedded systems [4]. But given the current trends of the increasing size and complexity of software implementations, manual code review isn't a reliable or feasible approach for the verification of flag variables. Further, effective testing of concurrent and multi-thread applications is complicated due to non-determinism in their execution. Hence identification of defects, which may arise due to inconsistencies in flag implementation, in a systematic and efficient manner, is not possible in the current state of art [7, 8].

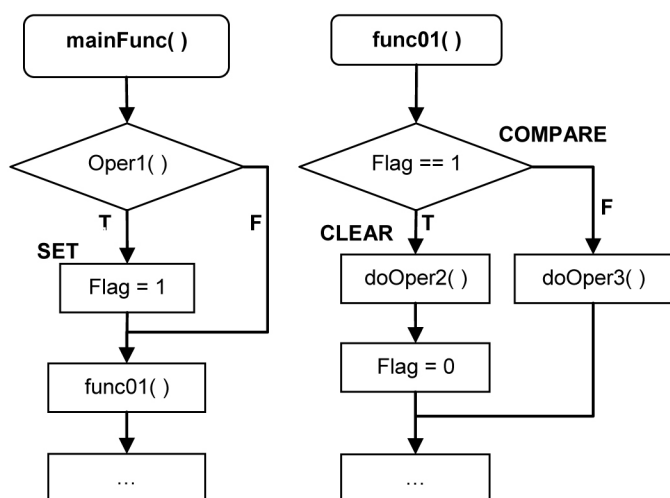


Figure 1. Demonstration of how flags are used for communication between various parts of code

Example 1 illustrates some typical defects related to incorrect usage of flags, which are as follows:

- I. Flag is not set/cleared/compared appropriately, which may lead to livelock or deadlock. For example, *stateFlag* is never cleared after change in state at line#36 which means that *Operation02* will never be performed and hence system operation may not proceed further.
- II. Same flag is incorrectly used to represent multiple conditions or states. For example, *Task2* expects *Task1* to set *syncFlag* when *Operation03* is completed, but *Task3* is also setting the same flag when *Operation05* is complete. Such implementation may cause problems when flags are used to synchronize sequential operations.
- III. Flag is written with incorrect value. After a *compare* operation, a flag may get written with incorrect value, typically with the value same as *compare* value. For example, it is expected that *Operation04* be performed whenever *loopFlag* is in *clear* state, but due to the incorrect *clear* operation at line#16 the loop body will not get executed once *Operation04* is performed.
- IV. Flag is having ambiguous values at *compare* operation. In a typical implementation, a flag's state at its *compare* operation can be used to guarantee the execution of a particular operation. For example at line#55 it is expected that *overFlag* is set whenever *Cond02* is true, and *Operation06* is performed. However, due to the unexpected *clear* at line#53, the execution of *Operation06* is not guaranteed.

In this paper, we propose an automated technique for verification of flag variables and their usage patterns, as illustrated in [Figure 2](#). The presented technique is a two-step approach, which makes use of static analysis, to first identify all potential flag variables and later verify all operations performed on the identified flag variables to detect inconsistencies in flag implementation

Example 1. Example illustrating possible defects due to incorrect usage of flags

<pre> 1. char syncFlag; 2. int stateFlag, loopFlag, overFlag; 3. 4. void Task1() 5. { 6. loopFlag = 1; 7. 8. if(Operation03()) 9. syncFlag = 1; 10. ... 11. while(IGN_ON()) 12. { 13. if(loopFlag==0) 14. { 15. //Do Operation 04 16. loopFlag=0; 17. } 18. ... 19. if(loopFlag==1) 20. { 21. //Loop Body 22. ... 23. if(Cond01()) 24. loopFlag=0; 25. } 26. ... 27. } 28. }</pre>	<pre> 29. void Task2() 30. { 31. if(syncFlag==0) 32. { 33. //Do Operation 01 34. ... 35. if(State_OFF()) 36. //stateFlag not cleared 37. ... 38. } 39. if(State_ON()) 40. stateFlag=1; 41. ... 42. if(stateFlag==0) 43. //Do Operation 02 44. } 45. void Task3() 46. { 47. if(Cond02()) 48. overFlag = 1; 49. ... 50. if(Operation05()) 51. { 52. syncFlag = 1; 53. overFlag = 0; 54. } 55. if(overFlag == 1) 56. //Do Operation 06 57. }</pre>
---	--

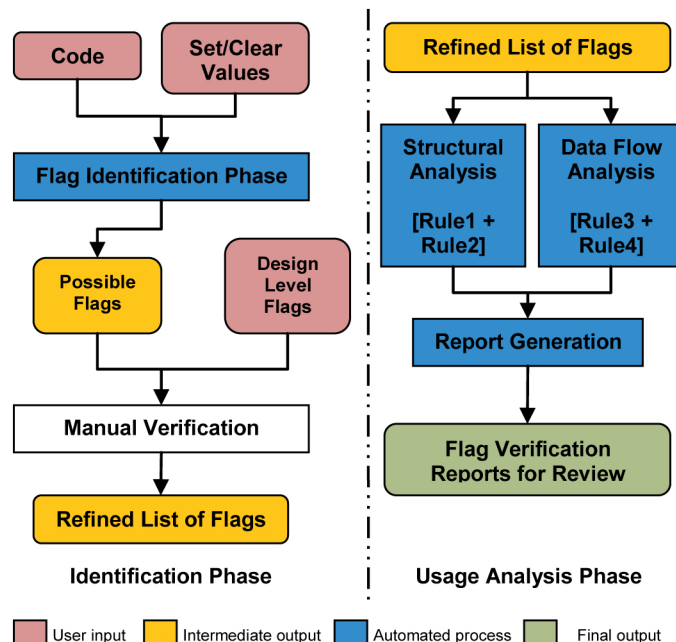


Figure 2. Overview of the presented flag verification technique

In the first step, known as the *Flag Identification Phase*, potential flag variables are identified based on the type of operations performed on the variables used in the application. The list of identified potential flags is then output to the user for the first level of manual verification. This makes the presented technique flexible as the user can either remove extra identified flags or add flags which are not identified automatically. This refined list of flags is then passed on to the next phase for further analysis.

In the second step, known as the *Flag Usage Analysis Phase*, the access points of the variables present in the refined list of identified flags are analyzed and are judged based on a pre-defined set of rules. These rules have been defined based on the listed defects (*refer I - IV*) we observed on various real world embedded applications. We use the following rules for verifying flag implementation:

- **Rule 1 - All operations:** A flag must be *set*, *cleared*, and *compared* at least once in its lifetime. The flag *stateFlag* in [Example 1](#) violates this rule as it is never *cleared* in its lifetime.
- **Rule 2 - Write in multiple synchronizing units:** A flag should not be written with the same value in multiple synchronizing units. The flag *syncFlag* in [Example 1](#) violates this rule because it is *set* in both *Task1* and *Task3*.
- **Rule 3 - Write after compare:** If a flag is modified inside its *comparison* block, it should be written with the value opposite to the *comparison* value. The *compare* operation of *loopFlag* at line#13 in [Example 1](#) violates this rule as *loopFlag* is written with the same value (0 at line#16) with which it is being compared.
- **Rule 4 - Conditional overwrite:** A flag must hold only one value, either *set* or *clear*, before getting *compared*. The *compare* operation of *overFlag* at line#55 in [Example 1](#) violates this rule as both *set* (from line#48) and *clear* (from line#53) values can reach this *compare* operation.

ISO 26262-6 section 7.4.5 [4] expects that the software architecture should describe aspects such as interface of software components, control flow and concurrency of processes, data flow between software components etc. The proposed technique considers design level flag information through user inputs and generates review useful information, which can be used to verify the implementation against design and improve review efficiency. Hence, the technique enables verification of implementation against design using static analysis, as per ISO 26262-6 section 8.4.5 [4].

In the presented paper, the section “Classification of Flags” describes the classification of the common types of flags used by developers. The section “Flag Identification Phase”, describes the criteria and process of flag identification in detail. In the section “Flag Usage Analysis Phase” we describe the rules used to judge flag access points and present findings from the experiments conducted on a real-world embedded code in the section “Experiments and Results”.

CLASSIFICATION OF FLAGS

The main objective of the proposed verification technique is to detect potential flag variables and check their access points for possible erroneous scenarios. The technique is capable of identifying two categories of flags - data type based flags, and scope based flags.

Data Type Based Flags

Data type based flags can be categorized further into the following sub-categories:

1. **Integral or Boolean Flags:** Global or static variables of primitive types that are assigned and compared with the values of *set* or *clear* throughout their lifetime are considered as integral flags. In [Example 2](#), *intFlag* is an integral flag.
2. **Bit-field Flags:** Members of a user defined data type (structure, union, and so on) can be used as one bit variable which represent a flag. These types of flags are known as bit-field flags. In [Example 2](#), *firstFlag* is a bit-field flag.
3. **Masked Flags:** Global or static variables in which each bit of a variable represents an independent flag are known as masked flags. The *set*, *clear* and *compare* operations on masked flags are performed using the bit-wise operators such as *&*, *|*, *~*. In [Example 2](#), *mask* is a masked flag. In the presented verification technique, the evaluation of judgments for each bit of an identified masked flag is carried out separately.

Scope Based Flags

Scope based flags can be categorized further into the following sub-categories:

1. **Pure Global Flag (PGF):** A PGF is a flag used for synchronization between multiple sections of the same task. In [Example 1](#), *stateFlag* is a PGF as it is used only in *Task2*.
2. **Shared Flag (SHF):** A SHF is a flag which is shared and used by multiple synchronization units (multiple tasks or ISRs or a combination of both) as a medium for communication. In [Example 1](#), *syncFlag* is a SHF because it is used in *Task1*, *Task2* and *Task3*.

FLAG IDENTIFICATION PHASE

The flag identification phase includes analysis of the Abstract Syntax Tree (AST) [9] of the entire application. AST corresponding to the application is obtained using customized front-end. Once AST is generated, it is traversed and potential flag variables are identified in parallel, based on the values with which the variables are written or compared. To ensure detection of all possible flags, implemented *set* and *clear* values are taken as input from the user.

Once the traversal of AST ends, all identified integral, bit-field and masked flags are given as an intermediate output to the user. Special handling is done for masked flags in the flag identification phase as these flags can be assigned and compared with values other than the *set* and *clear* values specified by user.

The variables which satisfy the criteria mentioned below are identified as potential flag variables:

Flag Identification Criteria

- Flag should be a global or static variable.
- Flag can be shared between different synchronizing units.
- Flag cannot be a constant variable as it is supposed to be modified by *set* and *clear* operations.
- Variables having float or double data type are not considered as flags.
- Flag should not be used in any arithmetic operation.
- A variable will be identified as a flag only if at least one of the operations *set*, *compare* or *clear* is performed on it.
- Flag can be assigned or compared with other flags only. It should not interact with non-flag variables.
- Flag should have only two values throughout the application i.e. one for *set* and other for *clear*.

The criteria stated above are applicable for the identification of integral and bit-field flags. Masked flags are identified based on the following additional criteria:

Masked Flag Identification Criteria

- A masked flag variable is similar to an integral flag variable except that the variable can have any integral value. This variable can have more than two values based on the bits which have been *set* or *cleared*.
- Masked flags will be identified only if the *set* and *clear* values are 1 and 0.
- The masked flag variable should be used in bit-wise operations such as *AND*, *OR*, *NOT*, *shift*.
- The masked flag variable should not be used in *XOR bit operation*.

Process for the Identification of Potential Flags

Example 2 best describes how flag identification process is carried out. Below is a description of what will happen with the variables used in this example:

- mask:** This will be detected as a masked flag as it participates in a bitwise *compare* operation at line#20. Since *mask* is a masked flag, assigning a value other than *set* (1) and *clear* (0) is allowed.
- intFlag:** This will be detected as an integral flag. It is interacting with *mask* (which itself is a masked flag) at line#19, and hence *intFlag* will be a valid flag.
- firstFlag:** This will also be detected as a flag as it is *set* at line#14 and *cleared* at line#24.
- nonVar:** This will not be detected as a flag as it is being assigned with a value other than *set* and *clear* at line#17.
- nonFlag:** This will not be detected as a flag as it is interacting with a non-flag variable *nonVar* at line#26.

Example 2. Example illustrating the types of flags and the flag identification process

1. unsigned char mask;	13. nonVar = 1;
2. struct myFlags	14. firstFlag = 1;
3. {	15. ...
4. char flag1 :1;	16. if(Cond())
5. ...	17. nonVar = 5;
6. }flag;	18. ...
7. int intFlag, nonVar, nonFlag;	19. intFlag = mask;
8. #define firstFlag	20. if((mask & 0x01) == 0x01)
9. void Task1()	21. {
10. {	22. mask = 126;
11. mask = 1;	23. intFlag = 0;
12. ...	24. firstFlag = 0;
	25. }
	26. nonFlag=nonVar;
	27. }

Flag Usage Analysis Phase

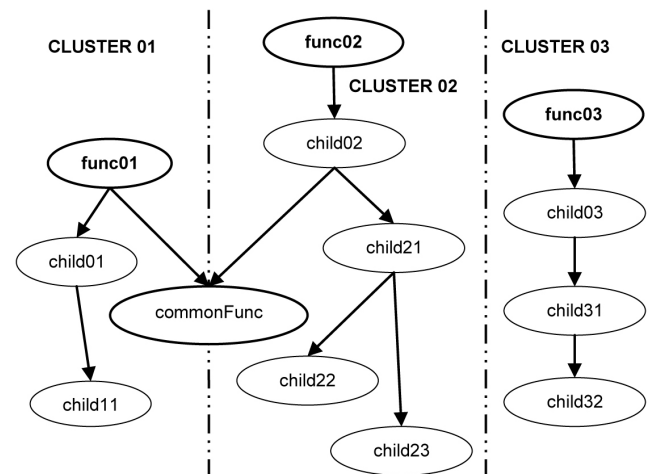


Figure 3. An illustrative example of the clustering process

In this phase, the access points of the variables present in the refined list of identified flags are analyzed in a task-wise manner. The complete application is partitioned into multiple independent tasks/ISRs with the help of clustering [10]. Clustering is a process of creating smaller and independent vertical slices of a large code base (Refer Figure 3 for more details). To ensure that all of the code which accesses flags is verified, ISRs are also treated as independent tasks. This is because flags can also be used as a communication medium between multiple ISRs or between multiple tasks and ISRs. Clustering preserves the system behavior as the results of analyzing the complete application are the same as merged results of individual tasks. This is possible because the analysis results that we are interested in are intra-task i.e. which do not require the effect of context switching to be taken into account.

Once clustering is completed and independent tasks/ISRs have been identified, flag access points in each task/ISR are examined for consistency and judged based on a pre-defined set of rules. These rules are applied for each identified flag simultaneously in a single iteration of the AST corresponding to