Unit Testing of Flash Memory Device Driver through a SAT-based Model Checker

Moonzoo Kim and Yunho Kim
CS Dept. Korea Advanced Institute of Science and Technology
Daejeon, South Korea
moonzoo@cs.kaist.ac.kr, vita500@gmail.com

Hotae Kim
Samsung Electronics
Suwon, South Korea
hotae.kim@samsung.com

Abstract

Flash memory has become virtually indispensable in most mobile devices. In order for mobile devices to successfully provide services to users, it is essential that flash memory be controlled correctly through the device driver software. However, as is typical for embedded software, conventional testing methods often fail to detect hidden flaws in the complex device driver software. This deficiency incurs significant development and operation overhead to the manufacturers.

Model checking techniques have been proposed to compensate weaknesses of conventional testing methods through exhaustive analyses. These techniques, however, require significant manual efforts to create an abstract target model and, thus, are not widely applied in industry. In this project, we applied model checking technique based on Boolean satisfiability (SAT) solver. One advantage of SAT-based model checking is that a target C code can be analyzed directly without an abstract model, thereby enabling fully-automated and bit-level accurate verification. *In this project, we have applied CBMC, a SAT-based software model checker, for unit testing of the Samsung OneNAND device driver. Through this project, we detected several bugs that had not been discovered previously.*

1 Introduction

Among the various available storage platforms, flash memory has become the most popular choice for mobile devices owing to its good characteristics such as low power consumption and strong resistance to physical shock. Thus, in order for mobile devices to successfully provide services to users, it is essential that the device driver of the flash memory operates correctly. However, as is typical of embedded software, conventional testing methods often fail to detect hidden bugs in the complex device driver software. This deficiency incurs significant overhead to the manufacturers.

Conventional testing has limitations in terms of checking whether a target software satisfies a given requirement specification, since testing does not provide complete coverage; it is very hard to systematically generate and test all possible scenarios/environments/configurations for the target software. As a result, subtle bugs are hardly detected by testing and cause significant overhead after the target software is deployed. In addition, even after detecting a violation, debugging requires significant human effort to trace the scenario leading to the violation step-by-step and thereby identify the cause of the violation. These limitations were manifest in the development of flash software for Samsung OneNAND$^{\mathrm{TM}}$ flash memory [1]. For example, a multi-sector read function was added to flash software to optimize the reading speed (see Sect. 5.3). However, this function caused numerous errors in spite of extensive testing and debugging efforts, to the extent that the developers seriously considered removing the feature.

Model checking techniques [10] have been proposed to compensate the aforementioned weaknesses of conventional testing methods by automatically exploring all state spaces of an abstract model of the target software. This is equivalent to testing the target model in all possible scenarios. In addition, if a violation is detected, a model checker generates a concrete counter example through which a bug can be conveniently identified.

However, model checking techniques are not widely applied in industry since there exists a *gap* between the target software and its abstracted model. To apply model checking, significant additional manual efforts are required to create an abstract target model, which is not affordable for most industrial software projects. On the other hand, software model checkers with automated abstraction capability can analyze a C code directly by automatically creating an abstract model out of the target C program. However, automatic abstraction based model checkers often provide inaccurate analysis results due to severe abstraction on arrays or pointers. Thus, these weaknesses of model checkers hinder adoption of model checking techniques as main-stream verification & validation (V&V) methods.

In this project, we applied a SAT-based model checker, CBMC [8], to find subtle bugs from the Samsung OneNAND device driver. CBMC directly analyzes C code without an abstract model and provides an accurate bit-level analysis. Thus, the aforementioned problems associated with model checkers are overcome by means of the advanced computational power of a modern SAT solver through intelligent heuristics (see Section 3). Through this project, we have demonstrated that a model checker can be used as an automated and productive unit-testing tool for an exhaustive analysis, which increases the reliability of embedded C codes as well as the productivity of software development in an industry setting.

2 Overview of the OneNAND Flash Device Driver

2.1 Overview of the Device Driver Software for OneNAND Flash Memory

There are two types of flash memories: NAND and NOR flash. NAND flash has higher density and thus is typically used as a storage medium. NOR flash is typically used to store software binaries, because it can execute software in place (XIP) whereas NAND cannot. OneNAND is a single chip comprising a NOR flash interface, a NAND flash controller logic, a NAND flash array, and a small internal RAM. OneNAND provides a NOR interface through its internal RAM. When an application executes a program in OneNAND, the corresponding page of the program is loaded into the RAM in OneNAND by the demand paging manager (DPM) for XIP (execution in place).

A unified storage platform (USP) is a software solution for OneNAND based mobile embedded systems. Fig. 1 presents an overview of USP. It manages both code storage and data storage. USP allows applications to store and retrieve data on OneNAND through a file system. USP contains a flash translation layer (FTL) through which data and programs in the OneNAND device are accessed. The FTL consists of three layers - a sector translation layer (STL), a block management layer (BML), and a low-level device driver layer (LLD). Generic I/O requests from applications are fulfilled through the file system, STL, BML, and LLD, in order. A *prioritized read request* for executing a program is made by DPM and this request goes to BML directly. Although USP allows concurrent I/O requests from multiple applications through STL, BML operations must be executed sequentially, not concurrently. For this purpose, BML uses a binary semaphore to coordinate concurrent I/O requests from STL. Furthermore, a prioritized read request from DPM can preempt generic I/O operations requested from STL. Thus, it is important to guarantee the correctness of I/O operations in concurrent settings. In this verification project, we analyzed FTL and DPM components of the USP.

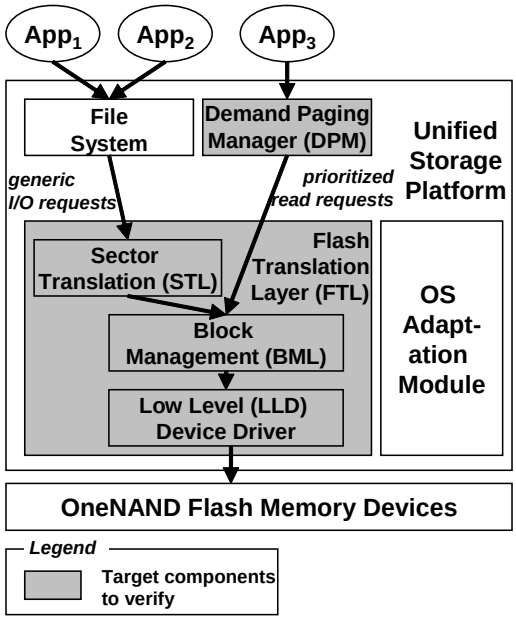

Figure 1. An overview of USP

2.2 Overview of Logical-to-Physical Sector Translation

A NAND flash device consists of a set of *pages*, which are grouped into *blocks*. A *unit* can be equal to a block or multiple blocks. Each page contains a set of *sectors*. When new data is written to flash memory, rather than overwriting old data directly, the data is written on empty physical sectors and the physical sectors that contain the old data are marked as invalid. Since the empty physical sectors may reside in separate physical units, one logical unit (LU) containing data is mapped to a linked list of physical units (PU). STL manages this mapping from logical sectors (LS) to

physical sectors (PS) and performs garbage collection. This mapping information is stored in a sector allocation map (SAM), which returns the corresponding PS offset from a given LS offset. Each PU has its own SAM.

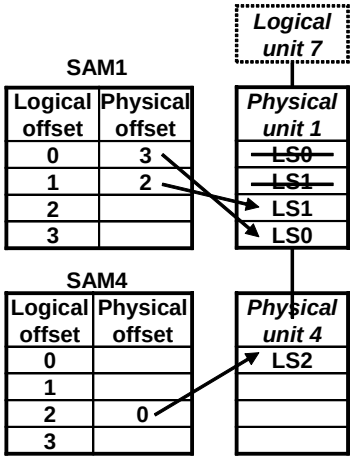

Figure 2. Mapping from logical sectors to physical sectors

Fig. 2 illustrates a mapping from logical sectors to physical sectors where 1 unit contains 4 sectors. Suppose that a user writes LS0 of LU7. An empty physical unit PU1 is then assigned to LU7, and LS0 is written into PS0 of PU1 (SAM1[0]=0). The user continues to write LS1 of LU7, and LS1 is subsequently stored into PS1 of PU1 (SAM1[1]=1). The user then updates LS1 and LS0 in order, which results in SAM1[1]=2 and SAM1[0]=3. Finally, the user adds LS2 of LU7, which adds a new empty physical unit PU4 to LU7 and yields SAM4[2]=0.

3 Overview of SAT-based Model Checking Technology

3.1 Boolean Satisfiability Solver

A Boolean satisfiability problem (SAT) entails determining whether there exists a propositional variable assignment σ that makes a given Boolean formula ϕ evaluate to true (i.e. $\exists\sigma.\sigma(\phi) = true$). SAT is a canonical NP-Complete problem and has received intensive theoretical treatment. In spite of its theoretical complexity, SAT solvers find applications in many fields including AI planning, circuit testing, and software model checking, since structured formulas generated from real-world problems are successfully solved by SAT solvers in many cases. The modern SAT solvers such as MiniSAT [9] and Chaff [15] exploit various heuristics [13] and can solve a large formula containing millions of variables and clauses [2] in a modest time. For example, [16] explains the mechanism by which modern SAT solvers can solve large formulas efficiently in spite of intractable complexity.

3.2 Translation from C Code to SAT Formulas

To use a SAT solver as a bounded model checker [5] to check whether a given C code (C) satisfies a requirement property (R), it is necessary to translate both C and R into Boolean formulas ϕ_C and ϕ_R, respectively. A SAT solver then determines whether $\phi_C \wedge \neg\phi_R$ is satisfiable: if the formula is satisfiable, it means that C violates R; if not, C satisfies R.

A brief sketch of the translation process is as follows [8]. We assume that a given C program is already preprocessed. First, the C program is transformed through the following steps

- The `break`, `continue`, and `return` statements are replaced by semantically equivalent `goto` statements.
- `switch` statements are transformed into semantically equivalent `if` and `goto` statements.
- The `for` and `do while` statements are replaced by equivalent `while` statements.
- Function calls are inlined and side effects such as ++ are replaced with equivalent statements by using new auxiliary variables.

Loop statements are then unwound. The while loops are unwound using the following transformation n times:

```
while(e) stm  ⇒ if(e) { stm; while(e) stm}
```

After unwinding the loop n times, the remaining while loop is replaced by an *unwinding assertion* that guarantees that the program does not execute more iterations. If the unwinding assertion is violated, n is increased until the unwinding bound is sufficiently large. Note that this bound n is just an upper bound of loop iteration, and does not have to be the exact number of iterations.

Finally, the transformed C program consists of only nested `if`, assignments, assertions, labels, and `goto` statements. This C program is transformed into a static single assignment (SSA) form. Fig. 3.b) illustrates this SSA transformation. This SSA program is converted into corresponding bit-vector equations and the final Boolean formula is a conjunction of all these bit-vector equations.

For example, Fig. 3.c) illustrates a Boolean conjunction of SSA statements (they are not converted into bit-vector equations yet for the reader's convenience). We know that

Fig. 3.a) violates `assert(x<3)`, since $\phi_C \wedge \neg\phi_R$ is satisfiable by σ such that $\sigma(x_0) = 1, \sigma(x_1) = 1, \sigma(x_2) = 2, \sigma(x_3) = 3, \sigma(x_4) = 3, \sigma(y_0) = 0$.

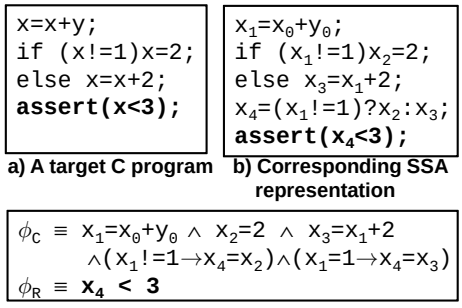

Figure 3. An example of translating a C program into a (semi) Boolean formula

3.3 C-based Bounded Model Checker (CBMC)

CBMC [8] is a bounded model checker for ANSI-C developed at CMU. CBMC receives a C program as its input and analyzes all C statements including pointer arithmetics, array, struct, etc with a bit-level accuracy as if it actually executes the target program. A requirement property is written as an assert statement in a target C program. The loop unwinding bound n is given explicitly as a command line parameter. If $\phi_C \wedge \neg\phi_R$ is satisfiable, CBMC generates a counter example showing an execution leading to the violation of the requirement property step by step.

One distinct feature of a CBMC based analysis, compared to testing, is the capability of handling *non-deterministic* values (i.e. function parameters, uninitialized local variables, or variables assigned with an non-deterministic values), which are useful to model unexpected user inputs or a range of values as a whole. Using this feature, CBMC provides a convenient method to *exhaustively* analyze all execution scenarios of a target C program. For example, if we analyze the `adder(unsigned char x, unsigned char y)` function, CBMC symbolically analyzes all $65536 (= 256^2)$ possible cases. If we provide an explicit constraint `__CPROVER_assume(x==1)`, the total number of cases to be analyzed is reduced to 256 cases since only `y` can have non-deterministic values ranging from 0 to 255. This capability of analyzing non-deterministic values can help automated unit testing of a C function by reducing the need of explicit generation of numerous test cases (see Sect. 5).

4 Project Overview

4.1 Overall Project Plan

Our team consists of two professors, one graduate student, and one senior engineer at Samsung Electronics. We worked on this verification project for six months. We spent the first three months reviewing the USP design documents and code to become better familiarized with USP and OneNAND flash. Most parts of USP are written in C and a small portion of USP is written in ARM assembler. Source codes of FTL and DPM are roughly 30000 lines long.

The goal of the project is to increase the reliability of USP by finding hidden bugs that have not been detected thus far. To this end, we applied a *top-down* approach to define code level properties from high level requirements. First, we selected target requirements to check from the USP documents. USP has a set of elaborated design documents in the following hierarchy.

- Software Requirement Specification (SRS)
- Architecture Design Specification (ADS)
- Detailed Design Specification (DDS)
  - DPM Detailed Design Specification
  - STL Detailed Design Specification
  - BML Detailed Design Specification
  - LLD Detailed Design Specification

SRS contains both functional and non-functional requirement specifications with priorities. We selected three functional requirements with very high priorities (see Sect. 4.2). Then, from the selected functional requirements, we investigated relevant ADS, DDS, and corresponding C codes to specify concrete code level properties (see Sect. 4.3). We then inserted these code level properties into the target C files as assert statements and subsequently analyzed those C files to check whether inserted assert statements are violated or not by using CBMC (see Sect. 5).

4.2 High-level Requirements

The SRS document specifies 13 functional requirements and 18 non-functional requirements for USP. Each requirement specifies its own priority. There are three functional requirements that have "very high" priority.

- *Support prioritized read operation*

  In order to execute a program through demand paging, DPM loads a code page into the internal RAM when a page fault exception occurs. Since the fault latency

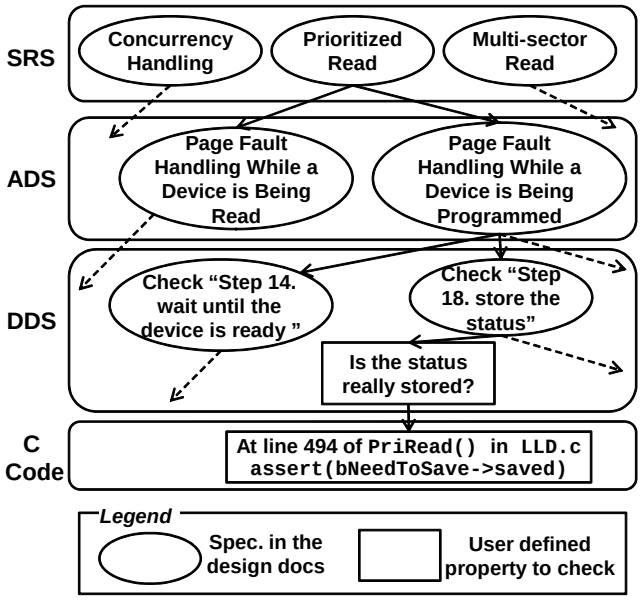

Figure 4. Top-down approach to identify code level properties to check

should be minimized, FTL should serve a read request from DPM prior to generic requests from a file system. This prioritized read request can preempt a generic I/O operation and resume the preempted operation later.

- *Concurrency handling*

  There are two types of concurrent behaviors in USP. The first is concurrency among multiple generic I/O operations. The second is concurrency between generic I/O operations and a prioritized read operation. USP should handle these two types of concurrent behaviors correctly, i.e., avoid a race condition or deadlock through synchronization mechanisms such as semaphores and locks.

- *Manage sectors*

  A file system assumes that flash memory is composed of contiguous logical sectors. Thus, FTL provides logical-to-physical mapping, i.e., multiple logical sectors are written over distributed physical sectors and these distributed physical sectors should be read back correctly.

We concentrate on checking the above three requirements and analyze relevant structures described in ADS. For example, as depicted in Fig. 4, a functional requirement on a prioritized read operation is related with page fault handling mechanisms, which are described in ADS. Again, such page fault handling mechanisms (e.g. page fault handling while a device is being programmed) are elaborated in the related DDS documents.

4.3 Low-level Properties

From the ADS document, we determine which DDS documents are related to the ADS description that is relevant to the three high-level requirements. The DDS documents contain elaborated sequence diagrams for various execution scenarios for the structures described in ADS. For example, as depicted in Fig. 4, we reviewed details of DPM DDS and LLD DDS that are relevant to the page fault handling mechanism while a device is being programmed. In LLD DDS, for example, a concrete sequence diagram for fault handling while a device is being programmed is described step by step. Fig. 5 describes one such sequence diagram.

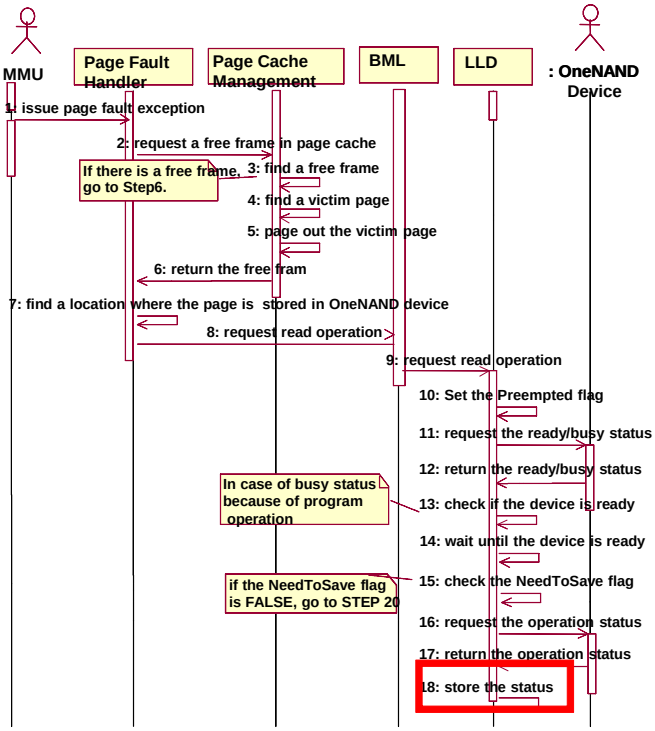

Figure 5. A sequence diagram of page fault handling while a device is being programmed in LLD DDS

USP allows a prioritized read operation to preempt currently executing generic operations. Thus, the status of a preempted operation should be saved and restored back in order to resume the preempted operation when the preempting prioritized read operation is completed. These saving and restoring operations are implemented in `PriRead`, which handles prioritized read operations. Step 18 in Fig. 5 describes a saving operation.

To check the correctness of step 18, i.e., whether or not the current status of a preempted generic operation is actually saved, we insert the following assert statement at line 494 of PriRead.

```
assert(!(pstInfo->bNeedToSave) || saved)
```

pstInfo and bNeedToSave are original program variables and saved is a newly added variable for verification purpose, which indicates whether the status has been saved. In a similar manner, we define 43 such code-level properties regarding the three high level requirements in a top-down manner.

5 Unit Testing Results

5.1 Prioritized Read Operation

A prioritized read operation is implemented in a function PriRead in LLD layer. This function is 234 lines long (including comments) and has 30 distinct paths in its control flow graph. Thus, to achieve full path coverage, a user has to generate at least 30 different test cases. The test case generation for path coverage is a difficult and time consuming task since the user has to analyze the target code to determine which input data will exercise which path. Instead, we used CBMC to *automatically* try all *exhaustive* value combinations of function parameters and global data which satisfy explicit user-defined constraints.

For example, a function parameter nDev indicates a physical device number, which can be 0 to 7 according to the OneNAND hardware specification. Thus, we add the following constraint statement (__CPROVER_assume(*Boolean constraint*)) to the head of PriRead. This restricts the possible range of nDev between 0 and 7 in the exhaustive analysis performed by CBMC.

```
__CPROVER_assume(0<=nDev && nDev<=7)
```

In addition, nPbn, another function parameter, indicates a physical block number, which has its maximal value according to the type of NAND device in which it is employed. This constraint is given as follows.

```
(!(NANDspec[nDev].nDID==SML)||nPbn<256)  &&
(!(NANDspec[nDev].nDID==LRG)||nPbn<2048)
```

CBMC tries not only all possible values of function parameters, but also global data being used by PriRead. For example, a global data SHDC contains a shared context for each OneNAND device and it is retrieved by PriRead. Following the LLD design document, several constraints can be specified. The following constraint is one such example, indicating that the number of physical sectors per a single unit should be equal to the multiplication of the number of blocks per unit, the number of pages per block, and the number of sectors per one page.

```
SHDC.nPhySctsPerUnit==SHPC.nBlksPerUnit
    * SHVC.nPgsPerBlk * SHVC.nSctsPerPg
```

CBMC translates PriRead into a SAT formula containing one million Boolean variables and 1340 clauses. In spite of large computational cost due to an exhaustive analysis, CBMC analyzed PriRead and found a violation in 8 seconds after consuming 325 megabytes of memory.[1] From this exhaustive analysis, it is found that the low-level property described as an example in Sect. 4.3 is violated and a counter example, as depicted in Fig. 6, is generated. The counter example describes that PriRead does *not* save the current status of an erase operation (see line 9-12 of Fig. 6), when the erase operation is preempted by a prioritized read operation (see line 3 of Fig. 6, which indicates that the current operation is an erase operation, because bEraseCmd is assigned as 1).

```
01:...
02:State 14 file LLD.c line 408 function PriRead thread 0
03:  LLD::PriRead::1::bEraseCmd=1
04:State 15 file LLD.c line 412 function PriRead thread 0
05:  LLD::PriRead::1::1::2::nWaitingTimeOut=(assignment removed)
06:State 17 file LLD.c line 412 function PriRead thread 0
07:  LLD::PriRead::1::1::2::nWaitingTimeOut=(assignment removed)
08:
09:Violated property:
10:  file LLD.c line 424 function PriRead
11:  assertion !(_Bool)pstInfo->bNeedToSave || (_Bool)saved
12:VERIFICATION FAILED
```

Figure 6. A counter example violating the assertion

5.2 Concurrency Handling

5.2.1 BML Semaphore Usage Pattern

Although USP allows concurrent I/O requests through STL, BML does not execute a new BML generic operation while another BML generic operation is running (i.e., BML operations must be executed sequentially, not concurrently). For this purpose, BML uses a binary semaphore to coordinate concurrent I/O requests from STL. Standard requirements for a binary semaphore are as follows:

- Every semaphore acquire operation (OAM_AcquireSM) should be followed by a semaphore release operation (OAM_ReleaseSM).

[1] All experiments presented in this paper are performed on a workstation equipped with 3Ghz Xeon and 32 gigabytes memory running 64 bit Fedora Linux 7. We used CBMC version 2.6.

- Every function should return with a semaphore released unless the semaphore operation raises an error.

A total of 14 BML functions use the BML semaphore. Each function is around 220 lines long and its cyclomatic complexity is 21 on average. We inserted an integer variable `smp` to indicate the status of the semaphore and simple codes to decrease/increase `smp` at corresponding semaphore operations in these 14 BML functions. We checked the following two properties.

1. $0 \leq$smp≤ 1 at every semaphore acquire operation.
2. `smp`==1 when a function using the semaphore returns unless a semaphore error occurs.

As we analyze `PriRead`, relevant constraints were specified in order to reduce the possible state space for the CBMC analysis. CBMC concludes that all these 14 functions satisfy the above two properties. CBMC takes 10 seconds while consuming around 300 megabytes of memory on average to analyze each function.

5.2.2 Handling Semaphore Exception

BML semaphore operation may cause an exception in some instances depending on the hardware status. Once such BML semaphore exception is raised, normal operations cannot be performed unless initialization is forced by a file system application. All BML functions that use the BML semaphore return `BML_ACQUIRE_SM_ERROR` to its caller immediately when the semaphore operation raises an exception. This error flag should be propagated through call graph chain to the topmost STL function, which should return `STL_CRITICAL_ERROR` to the application. Fig. 7 presents a partial call graph of functions that eventually perform the semaphore acquire operation by calling `OAM_AcquireSM`.

We analyzed the topmost STL functions such as `STL_Write` (depicted in the left most part of Fig. 7) to check whether these functions always return `STL_CRITICAL_ERROR` if `OAM_AcquireSM` raises an exception. For example, for `STL_Write`, CBMC should analyze 8 levels of the call graph.

We add a global variable `SMerr` to indicate when a semaphore exception is raised. We can then check whether the semaphore exception is correctly propagated to the topmost application by checking the return value `nErr` of the topmost STL functions. This property check can be done by inserting the following assert statement before the `return` statement in the topmost STL functions.

```
assert(!(SMerr==1)||nErr==STL_CRITICAL_ERROR)
```

For example, we verified `STL_write`, which is 84 lines long. In this analysis, however, all sub functions of `STL_write` (e.g. `SM_WriteSectors` (104 lines), `_KeepBoundsOfDepth`(31 lines), etc) should be analyzed together, which further increases the complexity of the analysis. To reduce the analysis complexity, we start the analysis by setting the loop unwinding bound as 1, which means that CBMC analyzes scenarios where all loop bodies are executed only once or passed. In this setting, CBMC does not report any violations. When the loop bound is increased to 2, a violation is detected in 97 seconds with 616 megabytes of memory consumed. After analyzing the counter example, it is found that a subfunction `_GetSInfo` has a bug. When `_GetSInfo` calls `BML_Read`, `_GetSInfo` does *not* check the return flag of `BML_Read`. Therefore, `_GetSInfo` fails to recognize the exception raised in BML_Read and does not propagate the exception to `_LoadSam` and upto `STL_Write`.

5.3 Multi-sector Read (MSR) Operation

5.3.1 Overview of MSR

USP provides a mechanism to simultaneously read as many multiple sectors as possible in order to improve the reading speed. Due to the non-trivial traversal of data structures for logical-to-physical sector mapping (see Section 2.2), the function for MSR is 157 lines long and highly complex, having 4-level nested loops. The requirement for MSR is that the content of the read buffer should correspond to the original data in the flash memory when MSR finishes reading.

Due to the complexity of the nested loops, MSR has a notorious bug history, to the extent that the developers seriously considered removing the feature. For example, if MSR is designed incorrectly, MSR may read data of Fig. 8.b) incorrectly when PU0, PU1, and PU2 are mapped to LU0 in order and PU3 and PU4 are mapped to LU1 in order, because data are not distributed over PS's in order.

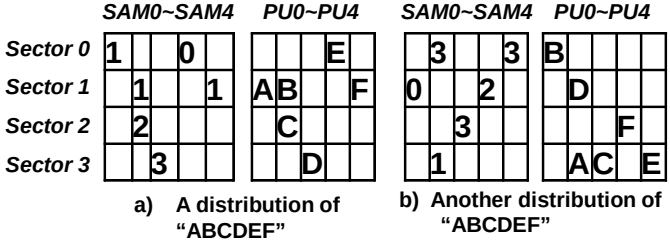

Figure 8. Two different distributions of data "ABCDEF" to physical sectors

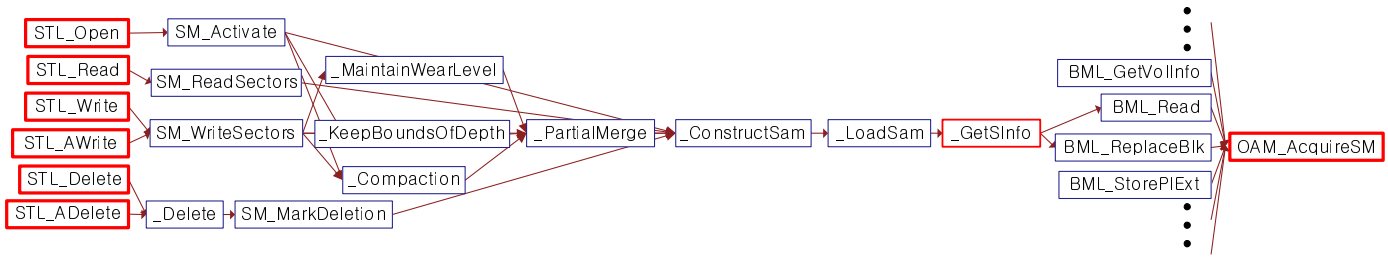

Figure 7. Call graph of functions using the BML semaphore

5.3.2 Test Environment for MSR

MSR assumes randomly written logical data on PUs and a corresponding SAM records the actual location of each LS. The writing of data to read is, however, not purely random. This means that a test environment has to be created such that a logical relation is maintained between SAMs and PUs as shown in Fig. 8, in order to generate valid test scenarios. In this analysis task, we create a test environment for MSR by specifying constraints representing this relationship. For example, the test environment should conform to the following rules.

1. One PU is mapped to at most one LU.
2. If the i_{th} LS is written in the k_{th} sector of the j_{th} PU, then the $(i \ mod \ m)_{th}$ offset of the j_{th} SAM is valid and indicates the PS number k, where m is the number of sectors per unit (4 in our experiments).
3. The PS number of the i_{th} LS must be written in *only* one of the $(i \ mod \ m)_{th}$ offsets of the SAM tables for the PUs mapped to the $\lfloor \frac{i}{m} \rfloor_{th}$ LU.

For example, the last two rules can be specified by the following invariants. [2]

$$\begin{aligned}\forall i,j,k \qquad & (logical\_sectors[i] = PU[j].sect[k] \rightarrow \\ & (SAM[j].valid[i \ mod \ m] = true \ \& \\ & SAM[j].offset[i \ mod \ m] = k \ \& \\ & \forall p.(SAM[p].valid[i \ mod \ m] = false) \\ & \text{where } p \neq j \text{ and } PU[p] \text{ is mapped to} \lfloor \frac{i}{m} \rfloor_{th} LU)\end{aligned}$$

Note that the total number of SAMs and PUs configurations increases exponentially as the size of logical sectors or the number of PUs increases. Thus, it is necessary to limit the size of the test environment within a reasonable range.

For the number of loop unwindings, the upper bound of each loop can be obtained from the algorithm of MSR as follows. The outermost loop iterates at most L times, where L is the length of logical data in sectors. The second outermost loop iterates at most M times, where M is the size of LUs. The third loop iterates at most N times where N is the total number of PUs. The innermost loop iterates at most 4 times, since one PU contains 4 PS's. For example, $L = 6, M = 2, N = 5$ for Fig. 8.a).

[2] These invariants allow spurious value combinations in SAMs, to reduce the complexity of imposing invariants. However, this weakening of invariants does not produce false positives when checking the requirement.

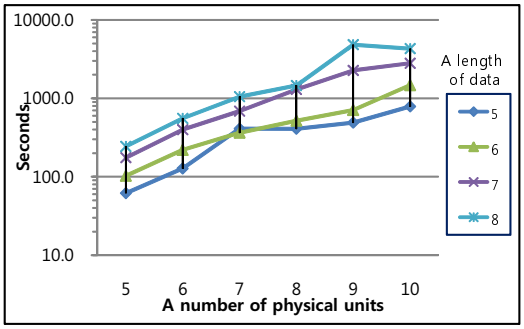

Figure 9. Time complexity of the MSR testings

5.3.3 Testing Results

We tested MSR for 5 to 8 sectors long data distributed over 5 to 10 PUs. Through CBMC experiments, no violations were detected. Note that the test environment created here exhaustively generate all scenarios. For example, for 6 sectors long data distributed over 10 PUs, there exist 2.7×10^8 distinct test scenarios. CBMC has analyzed all such scenarios *exhaustively*. Therefore, compared to randomized testing, this exhaustive analysis capability can provide higher confidence on the correctness of MSR. The experimental results are illustrated in Fig. 9. For example, it takes 1471 seconds to test all 2.7×10^8 scenarios 6 sectors long data distributed over 10 PUs. For each of the experiments, 200 to 700 megabytes of memory were consumed. For more details on the analysis of MSR, see [14].

6 Lessons Learned

6.1 Software Model Checker as an Effective Unit Testing Tool

Model checking techniques have been used as a means to improve the reliability of computing systems by detecting subtle flaws through an exhaustive analysis. However, due to the state explosion problem [11] and the gap between the target program and the corresponding formal model, model checkers have not been widely used as a software validation tool.

Through this project, however, we found that cutting-edge SAT based model checkers can overcome these weaknesses. The capability of the direct C code analysis of CBMC elimitates the overhead to create a formal model. In addition, the SAT solver could exhaustively test units of codes while consuming only modest time and memory, although not all software codes as a whole can be tested and the binary libraries used by the target program cannot be analyzed if any. Furthermore, a CBMC based code analysis is more convenient than actual testing, because CBMC does *not* require any test harnesses except environment constraints. A step-by-step counter example generated by CBMC also serves as an effective debugging aid.

Finally, as demonstrated in Sect. 5, a CBMC based analysis, due to its exhaustive analysis capability, could detect several hidden bugs that had not been previously detected by Samsung. Therefore, a SAT-based model checker should be evaluated tried as a unit test tool in software development, as it can provide higher confidence of code quality with an acceptable overhead.

6.2 Benefits of Constraint Based Exhaustive Testing

Although to date there active research on model based testing has been carried out [20, 19], generation of test cases adequate for various test criteria [21] still requires a great deal of human effort in many cases. In this project, we avoided this laborious task of explicit test case generation. Instead, we mechanically tested all possible execution scenarios that satisfy environmental constraints. This approach was successful for unit testing of USP; exhaustive unit testing consumed only modest amounts of time and memory.

In addition, even when a set of test cases reaches a complete statement coverage or branch coverage, the absence of error is still not guaranteed, since different input values can generate different outputs even in the same execution path. Thus, the necessity of an exhaustive analysis remains even when test criteria are satisfied. Therefore, for unit testing, this exhaustive analysis with constraints can produce better confidence in the correctness of the target code while requiring minimal human efforts.

6.3 Advantages of a SAT-based Model Checker

Several works have applied model checking techniques to verify C programs directly [12, 6, 18]. Software model checkers such as Blast [7] and SLAM [17] use various abstraction techniques such as counter example based abstraction refinement (CEGAR) to alleviate the state space explosion problem. However, these approaches suffer from various problems due to excessive abstraction and limitation of underlying decision theories. Consequently, the analysis results are inaccurate or the analysis may halt unexpectedly due to a failure to find proper predicates. For example, we performed the same analysis tasks using Blast as we did with CBMC. Blast produced correct results for the analysis of the BMC semaphore usage pattern (see Sect. 5.2.1). However, Blast unexpectedly halted the analysis of `PriRead` (see Sect. 5.1), because it failed to find proper predicates. Furthermore, we could not expect Blast to analyze MSR correctly, since Blast has a very limited analysis capability on array operations.

CBMC can produce accurate results because it straightforwardly (i.e. at a bit-level accuracy) transforms a target C program into a huge SAT formula consiting of millions of variables and clauses. However, an underlying SAT solver can solve this huge formula very efficiently with the help of advanced heuristics (see Sect. 3.1).

One difficulty of SAT based model checking is finding the upper bounds for loop unwindings. Although this problem is in general difficult to solve, a user can obtain a good approximation for unwinding upper bounds after analyzing the target program. In this project, we could decide the unwinding bounds from the DDS documents and the target code without difficulty. In addition, even if we fail to obtain the unwinding upper bounds, CBMC can still evaluate scenarios within the user designated unwinding bounds. This can still provide opportunities to detect bugs in those incomplete scenarios. [3]

7 Conclusion and Future Work

In this project, we successfully applied a SAT based software model checker CBMC to detect bugs in the device driver software for OneNAND flash memory. These bugs include incomplete handling of semaphore errors and a logical bug that does not store the current status of an erase operation that is preempted by a prioritized read operation.

[3] A user has to check whether a detected bug is a real one through a counter example generated, if the bug is detected when an insufficient bound is used.

These bugs had not been previously detected. In addition, we could establish confidence in the correctness of the very complex multi-sector read function, although complete verification on a large size flash was not established.

It must be noted that the current model checking technology is not yet scalable to automatically verify the entire C code of a safety critical system [3]. However, it is still beneficial to use SAT-based model checkers when testing units of a target C program with nominal overhead, as demonstrated in this project. Samsung highly valued the project results and, as a next project, the authors plan to analyze a flash file system to check data consistency at the events of random power-off. We also plan to apply the Satisfiability Modulo Theories (SMT) solver [4] instead of a SAT solver for realization of an efficient decision procedure, as this can remove huge bit-vector equations generated from the target code.

Acknowledgments

We would like to thank Prof. Yunja Choi at Kyungpook National University for valuable discussion on the verification of MSR.

References

[1] Samsung OneNAND fusion memory. `http://www.samsung.com/global/business/semiconductor/products/fusionmemory`.

[2] Sat competition 2007: a satellite event of the sat 2007 conference, 2007. http://www.satcompetition.org/2007/.

[3] A.Groce, G.Holzmann, and R.Joshi. Randomized differential testing as a prelude to formal verification. In *International Conference on Software Engineering*, May 2007.

[4] C. Barrett, L. de Moura, and A. Stump. SMT-COMP: Satisfiability Modulo Theories Competition. *Computer Aided Verification*, pages 20–23, 2005.

[5] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking, 2003.

[6] C.Killian, J.W.Anderson, R.Jhala, and A.Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *USENIX Symposium on Networked Systems Design and Implementation*, 2007.

[7] D.Beyer, T.A.Henzinger, R.Jhala, and R.Majumdar. The software model checker blast: Applications to software engineering. *Int. Journal on Software Tools for Technology Transfer*, 2007.

[8] E.Clarke, , D.Kroening, , and F.Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, pages 168–176. Springer, 2004.

[9] N. Een and N. Sorensson. An extensible sat-solver. In *SAT*, 2003.

[10] E.M.Clarke, O.Grumberg, and D.A.Peled. *Model Checking*. MIT Press, January 2000.

[11] E.M.Clarke, O.Grumberg, S.Jha, Y.Lu, and H.Veith. Progress on the state explosion problem in model checking. *Lecture Notes in Computer Science*, 2000, 2001.

[12] J.Yang, P.Twohey, D.Engler, and M.Musuvathi. Using model checking to find serious file system errors. In *Operating System Design and Implementation*, 2004.

[13] L.Zhang and S.Malik. The quest for efficient boolean satisfiability solvers. In *Computer Aided Verification*, 2002.

[14] M.Kim, Y.Kim, Y.Choi, and H.Kim. Pre-testing flash device driver through model checking techniques. In *IEEE Int. Conf. on Software Testing, Verification and Validation*, 2008.

[15] M.Moskewicz, C.Madigan, Y.Zhao, L.Zhang, and S.Malik. Chaff: Engineering an efficient sat solver. In *Design Automation Conference*, June 2001.

[16] R.Williams, C.P.Gomes, and B.Selman. Backdoors to typical case complexity. In *Intl. Joint Conf. on Artificial Intelligence*, 2003.

[17] T.Ball and S.K.Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.

[18] T.Witkowski, N.Blanc, D.Kroening, and G.Weissenbacher. Model checking concurrent linux device drivers. In *Automated Software Engineering*, November 2007.

[19] M. Utting and B. Legeard. Practical Model-Based Testing. *Morgan Kaufmann*, 1(1.4):1–5, 2007.

[20] G. Wimmel, H. Loetzbeyer, A. Pretschner, and O. Slotosch. Specification based test sequence generation with propositional logic. *Software Testing Verification and Reliability*, 10(4):229–248, 2000.

[21] H. ZHU, P.A.V. HALL, and J.H.R. MAY. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4), 1997.