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

Moonzoo Kim and Yunho Kim
CS Dept. KAIST, Daejeon, South Korea
moonzoo@cs.kaist.ac.kr, kimyunho@kaist.ac.kr

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 operate successfully, it is essential that the 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 overheads to the manufacturers.*

*Model checking techniques have been proposed to compensate for the 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 a model checking technique based on a 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 automated and bit-level accurate verification. In this project, we have applied CBMC, a SAT-based software model checker, to the unit testing of the Samsung OneNAND$^{\text{TM}}$ device driver. Through this project, we detected several bugs that had not been discovered previously.*

## 1 Introduction

Among the various storage platforms available, flash memory has become the most popular choice for mobile devices. 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 overheads to the manufacturers.

Conventional testing has limitations in verifying whether a target software satisfies a given requirement specification, since testing does not provide complete coverage. Furthermore, it requires significant human effort to generate effective test cases that provide a certain degree of statement/branch coverage. As a result, subtle bugs are hard to detect by testing and can cause significant overheads after the target software is deployed. In addition, even after detecting a violation, debugging requires much human effort to step-by-step replay and analyze what lead to the scenario where the violation occurred. These limitations were manifest in the development of flash software for Samsung's OneNAND$^{\text{TM}}$ flash memory [1]. For example, a multi-sector read function was added to flash software to optimize the reading speed (see Section 5.3); however, this function caused numerous errors despite extensive testing and debugging efforts, to the extent that the developers seriously considered removing the feature.

Model checking techniques [12] have been proposed to compensate for the aforementioned weaknesses of the conventional testing methods by automatically exploring the entire state space of an abstract target model. In addition, if a violation is detected, a model checker generates a concrete counter example through which the bug can be conveniently identified.

However, model checking techniques are not widely applied in industry since a gap exists between the target software and its abstracted model. To apply model checking, significant additional efforts are required to create an abstract target model, which is not affordable for most indus-

*This work was supported by the KAIST Institute for Information Technology Convergence and Samsung Electronics.

trial software projects. However, software model checkers with automated abstraction capabilities [9, 18] often result in inaccurate analyses due to limited abstraction capabilities. Thus, these weaknesses of model checkers hinder the adoption of model checking techniques as main stream verification and validation (V&V) methods.

In this project, we applied a SAT-based model checker, the C bounded model checker (CBMC) [10], to find subtle bugs in the Samsung's OneNAND$^{TM}$ device driver. CBMC directly analyzes a C program without an abstract model and provides accurate analysis results. Through this project, we have demonstrated that a model checker can be used as an automated and productive unit testing tool, which increases the reliability of an embedded C program as well as the productivity of software testing in an industry setting.

# 2 Overview of the OneNAND$^{TM}$ Flash Device Driver

## 2.1 Overview of the Device Driver Software for OneNAND$^{TM}$ Flash Memory

There are two types of flash memories: NAND and NOR. NAND flash has a 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$^{TM}$ is a single chip comprising a NOR flash interface, a NAND flash controller logic, a NAND flash array, and a small internal RAM. OneNAND$^{TM}$ provides a NOR interface through its internal RAM. When an application executes a program in OneNAND$^{TM}$ , the corresponding page of the program is loaded into the RAM in OneNAND$^{TM}$ using demand paging manager (DPM) for XIP.

Unified storage platform (USP) is a software solution for OneNAND$^{TM}$ based mobile embedded systems. Figure 1 presents an overview of USP: it manages both code storage and data storage. USP allows applications to store and retrieve data on OneNAND$^{TM}$ through a file system. USP contains a flash translation layer (FTL) through which data and programs in the OneNAND$^{TM}$ device are accessed. 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 that order. A *prioritized read request* for executing a program is made by DPM and this request goes directly to BML. 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 by STL. Thus, it is important to guarantee the correctness of the I/O operations in concurrent settings. In this project, we analyzed FTL and DPM components of USP.

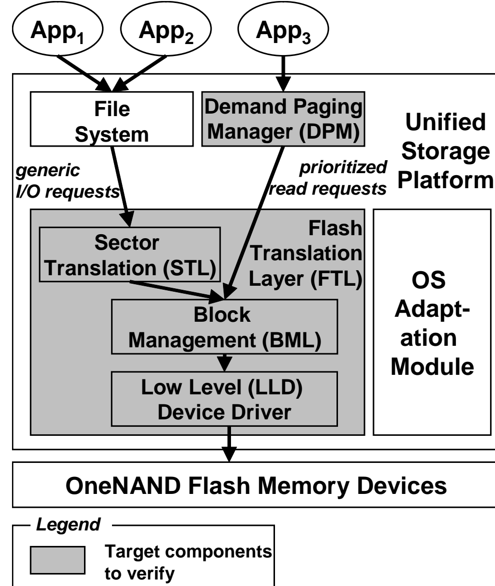


**Figure 1. Overview of USP**

## 2.2 Overview of the Logical-to-Physical Sector Translation

A NAND flash device consists of a set of *pages* that 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 the flash memory, rather than directly overwriting old data, 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 the mapping from the logical sectors (LS) to the physical sectors (PS). 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 illustrates the mapping from logical sectors to physical sectors where one unit contains four sectors. Suppose that a user writes LS0 of LU7. An empty physical unit PU1 is then assigned to LU7, and LS0 is written into the PS0 of PU1 (SAM1[0]=0). The user continues to write the LS1 of LU7, and the LS1 is subsequently stored into the 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 the LS2 of LU7, which adds a new empty physical unit PU4 to LU7 and yields SAM4[2]=0.

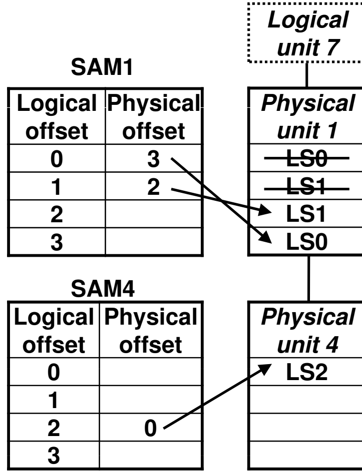


**Figure 2. Mapping from logical sectors to physical sectors**

# 3 Overview of the SAT-based Model Checking Technology

## 3.1 Boolean Satisfiability Problem

A Boolean satisfiability problem (SAT) verifies whether a propositional variable assignment $\sigma$ that makes a given Boolean formula $\phi$ evaluate to true (i.e. $\exists\sigma.\sigma(\phi) = true$) exists. SAT is a canonical NP-complete problem and has received intensive theoretical treatment. Despite its theoretical complexity, SAT finds applications in many fields including AI planning, circuit testing, and software model checking, since the structured formulas generated from real world problems are successfully solved by SAT solvers in many cases. The modern SAT solvers, such as MiniSAT [11] and Chaff [17], exploit various heuristics [15] and can solve a large SAT formula containing millions of variables and clauses in a modest time [2].

## 3.2 Translation from a C Code to a SAT Formula

To use a SAT solver as a bounded model checker [6] to verify whether a given C code ($C$) satisfies a requirement property ($R$), it is necessary to translate both $C$ and $R$ into Boolean formulas $\phi_C$ and $\phi_R$, 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 $R$ with respect to the bound.

A brief sketch of the translation process is as follows [10]. We assume that a given C program is already preprocessed. First, the C program is transformed through the following steps:

- `break`, `continue`, and `return` statements are replaced by semantically equivalent `goto` statements.
- `switch` statements are transformed into semantically equivalent `if` and `goto` statements.
- `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 using new auxiliary variables.

The loop statements are then unwound. The `while` loops are unwound using the following transformation $n$ times:

`while(e) stm` $\Rightarrow$ `if(e)` $\{$`stm; while(e) stm`$\}$

After unwinding the loop $n$ times, the remaining `while` loop is replaced by an unwinding assertion `assert(!e)` 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 only an upper bound of the loop iteration and does not need 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. Figure 3(b) illustrates the SSA form of the C program. 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, Figure 3(c) illustrates a Boolean conjunction of the SSA statements; however, they are not converted into bit-vector equations yet. We know that Figure 3(a) violates `assert(x<3)`, since $\phi_C \wedge \neg\phi_R$ is satisfiable by $\sigma$ such that $\sigma(x_0) = 1, \sigma(x_1) = 1, \sigma(x_2) = 2, \sigma(x_3) = 3, \sigma(x_4) = 3, \sigma(y_0) = 0$.

## 3.3 The C Bounded Model Checker (CBMC)

CBMC [10] is a bounded model checker for ANSI-C developed at CMU. CBMC receives a C program as its input and analyzes all C statements (e.g. pointer arithmetics, arrays, structs, function calls, etc.) with bit-level accuracy. 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 that shows a step-by-step execution leading to the violation of the requirement property.

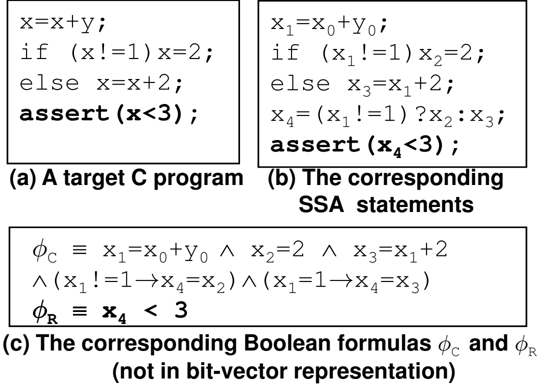


**Figure 3. Example of translating a C program into a Boolean formula**

One distinct feature of the CBMC based analysis, compared with testing, is its capability of handling *non-deterministic values*, which are useful in modeling unexpected user inputs, a range of values as a whole, or return values of undefined functions. Using this feature, CBMC can conveniently analyze all execution scenarios of a target C program. For example, if we analyze `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 analyze is reduced to 256, since only `y` has a non-deterministic value ranging from 0 to 255. This capability of analyzing non-deterministic values helps the unit testing of C functions by reducing the manual effort to explicitly generate test cases (see Section 5).

# 4 Project Overview

## 4.1 Overall Project Plan

Our team consisted of two professors, one graduate student, and one senior engineer from 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 familiarized with USP and OneNAND$^{TM}$ flash. Most parts of USP were written in C and a small portion of USP was written in ARM assembler. The source codes of FTL and DPM were roughly 30000 lines long.

The goal of this project was to increase the reliability of USP by finding hidden bugs that had not been detected. For this purpose, it was not enough to check the pre-defined API interface rules as found in other research [4, 7]. Instead, we needed to verify the *functional correctness* which can assure conformance to the given high-level requirements. Thus, we needed to identify the properties to verify first, and the identification of such code-level properties required significant effort, since it requires complete knowledge of the high-level design and requirements of the target system, low-level implementation, and mapping between the design and the implementation. Although most formal verification research assumes that these properties are given from somewhere, we must define these properties by ourselves in the real world. We spent two months defining the code-level properties based on the given high-level requirements.

To this end, we applied a *top-down* approach to identify the code-level properties to verify from the high-level requirements (see Figure 4). First, we selected the target requirements from the USP documents. USP has a set of elaborated design documents as follows:

- Software requirement specifications (SRS)
- Architecture design specifications (ADS)
- Detailed design specifications (DDS)
  - DPM, STL, BML, and LLD DDS's

SRS contains both functional and non-functional requirement specifications with priorities. We selected three functional requirements with very high priorities (see Section 4.2). Then, from the selected functional requirements, we investigated the relevant ADS, DDS, and corresponding C codes to specify concrete code-level properties (see Section 4.3). We inserted these code-level properties into the target C files as `assert` statements and analyzed those C files to verify whether the inserted `assert` statements are violated or not using CBMC (see Section 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 were three functional requirements that have "very high" priorities as follows:

- *Support prioritized read operation*

  In order to execute a program, DPM loads a code page into the internal RAM when a page fault exception occurs. Since the fault latency 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 the preempted operation can be resumed later.

- *Concurrency handling*

  There are two types of concurrent behaviors in USP. The first behavior is concurrency among multiple

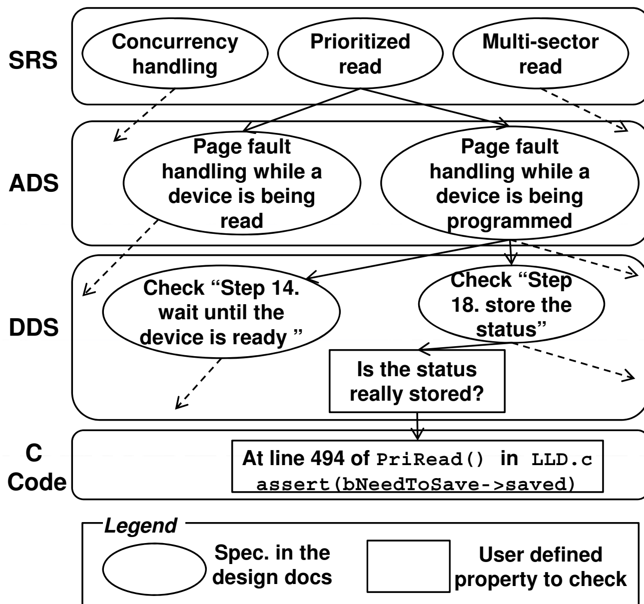


Figure 4. Top-down approach used to identify the code-level properties to verify

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. it should avoid a race condition or deadlock through synchronization mechanisms such as semaphores and locks.

- *Manage sectors*

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

We concentrated on verifying the above three requirements and analyzing the relevant structures described in ADS. For example, as depicted in Figure 4, a functional requirement on a prioritized read operation is related to the 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 determined which DDS documents were related to the ADS description relevant to the three high-level requirements. The DDS documents contain elaborated sequence diagrams of various execution scenarios for the structures described in ADS. For example, as depicted in Figure 4, we reviewed the details of the DPM DDS and LLD DDS that are relevant to the page fault handling mechanism while a device is being programmed. In the LLD DDS, for example, concrete sequence diagrams for fault handling while a device is being programmed are described (see Figure 5).

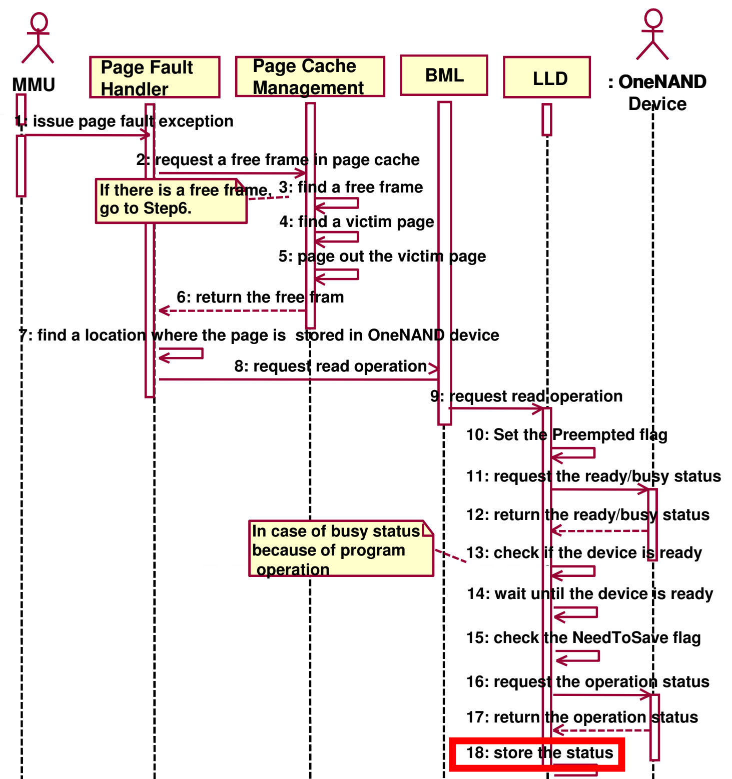


Figure 5. Sequence diagram of page fault handling while a device is being programmed

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

To check the correctness of Step 18, i.e. whether or not the current status of a preempted generic operation was actually saved, we inserted the following `assert` statement

at line 494 of `PriRead()`:

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

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

# 5 Unit Testing through CBMC

## 5.1 Prioritized Read Operation

A prioritized read operation is implemented in the `PriRead()` function in the LLD layer. This function is 234 lines long and has 21 independent paths in its control flow graph. Thus, to achieve full path coverage, a user must generate at least 21 different test cases. This test case generation for path coverage is a difficult and time consuming task, since a human tester must analyze the target code to determine which input data exercises which path. Instead, we used CBMC to automatically test all value combinations of the function parameters and global data that satisfy the explicit user-defined constraints.

For example, a function parameter `nDev` of `PriRead()`, which indicates a physical device number, can be 0 to 7 according to the OneNAND™ hardware specification. Thus, the following constraint statement was added to the head of `PriRead()`:

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

which restricts the possible range of `nDev` to between 0 and 7 in the analysis performed by CBMC. In addition, another function parameter `nPbn`, which indicates a physical block number, obtains its maximal value according to the type of NAND device. This constraint is given as follows:

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

CBMC uses not only all possible values of function parameters, but also the 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()`. Based on the LLD design document, several constraints can be specified. The following constraint is one such example, indicating that the number of physical sectors per 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 page.

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

With such constraints, CBMC translates `PriRead()` into a SAT formula containing one million Boolean variables and 1340 clauses. Despite the large computational cost due to the exhaustive analysis, CBMC analyzed `PriRead()` and found a violation in 8 seconds after consuming 325 megabytes of memory.[1] CBMC found that the code-level property described in Section 4.3 was violated and a counter example was generated as shown in Figure 6. The counter example describes that `PriRead()` does *not* save the current status of an erase operation (see lines 9-12 of Figure 6), when the erase operation is preempted by a prioritized read operation. Note that line 3 of Figure 6 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** `assert(!(pstInfo->bNeedToSave)||saved)`

## 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. the BML operations must be executed sequentially, not concurrently). For this purpose, BML uses a binary semaphore to coordinate concurrent I/O requests from STL. The 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()`).
- Every function should return with a semaphore released unless the semaphore operation creates an error.

Fourteen BML functions that use the BML semaphore exist. Each function is 220 lines long on average and its cyclomatic complexity is 13 on average. We inserted an integer variable `smp` to indicate the status of the semaphore and

[1] All experiments presented in this paper were performed on a workstation equipped with 3 Ghz Xeon and 32 gigabytes memory running 64 bit Fedora Linux 7. We used CBMC version 2.6 with MiniSAT 1.1.4 [11].

simple codes to decrease/increase smp at the corresponding semaphore operations in these 14 BML functions. We verified the following two properties:

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

CBMC concluded that all 14 BML functions satisfy the above two properties. CBMC took 10 seconds while consuming 300 megabytes of memory on average to analyze each function.

#### 5.2.2 Handling Semaphore Exception

The BML semaphore operation might cause an exception depending on the hardware status. Once such BML semaphore exception occurs, USP cannot operate correctly unless a re-initialization is forced by a file system. All BML functions that use the BML semaphore immediately return BML_ACQUIRE_SM_ERR or BML_RELEASE_SM_ERR to its caller when a semaphore operation raises an exception. This error flag should be propagated through a call-chain to a topmost STL function, which should return STL_CRITICAL_ERR to the file system. Figure 7 presents a partial call graph of the topmost STL functions (depicted in the leftmost area of Figure 7) that eventually call OAM_AcquireSM().

We verified whether the topmost STL functions such as STL_Write() always returned STL_CRITICAL_ERR if OAM_AcquireSM() called by the STL functions raises an exception. For example, to verify whether STL_Write() always returns STL_CRITICAL_ERR in the event of a BML semaphore exception, CBMC should analyze nine levels of the call graph, namely STL_Write() → SM_WriteSectors() → _KeepBoundsOfDepth() → _PartialMerge() → _ConstructSAM() → _LoadSam() → _GetSInfo() → BML_Read() → OAM_AcquireSM().

We added a global variable SMerr to indicate when a semaphore exception is raised. Then, we could verify whether the semaphore exception had been correctly propagated to the file system by verifying the return value nErr of the topmost STL functions. This property was checked by the following assert statement inserted before the return statement of the topmost STL functions:

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

We analyzed STL_write() (84 lines) first. In the 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 began the analysis by setting the loop unwinding bound to 2 and ignoring the unwinding assertions, which meant that CBMC analyzed the scenarios where all loop bodies were executed only once or passed. In this setting, a violation is detected in 97 seconds with 616 megabytes of memory consumed. After reviewing the counter example, we found that the violation was real and a sub-function _GetSInfo() had a bug. When _GetSInfo() called BML_Read(), _GetSInfo() may *not* have checked the return flag of BML_Read(). As a result, _GetSInfo() failed to recognize the exception raised in BML_Read() and did not propagate the exception to _LoadSam() and up to STL_Write(). Therefore, all topmost STL functions that called _GetSInfo() may not have properly handled the BML semaphore exception.

### 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 the logical-to-physical sector mapping (see Section 2.2), the function for MSR is 157 lines long and highly complex, having four-level nested loops. When MSR finishes the reading operation, the content of the read buffer should correspond to the original data in the flash memory.

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 was designed incorrectly, MSR may have read the data of Figure 8(b) incorrectly when PU0, PU1, and PU2 were mapped to LU0 in order, and PU3 and PU4 were mapped to LU1 in order, because the data were not distributed over the PS's in order.

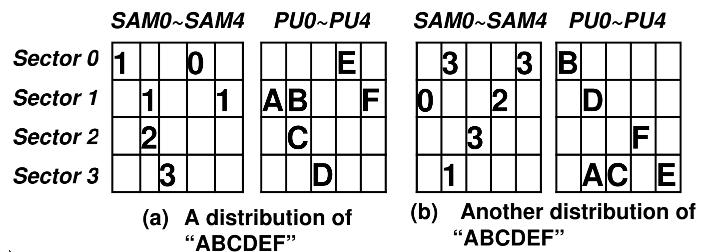


**Figure 8. Two different distributions of data "ABCDEF" to physical sectors**

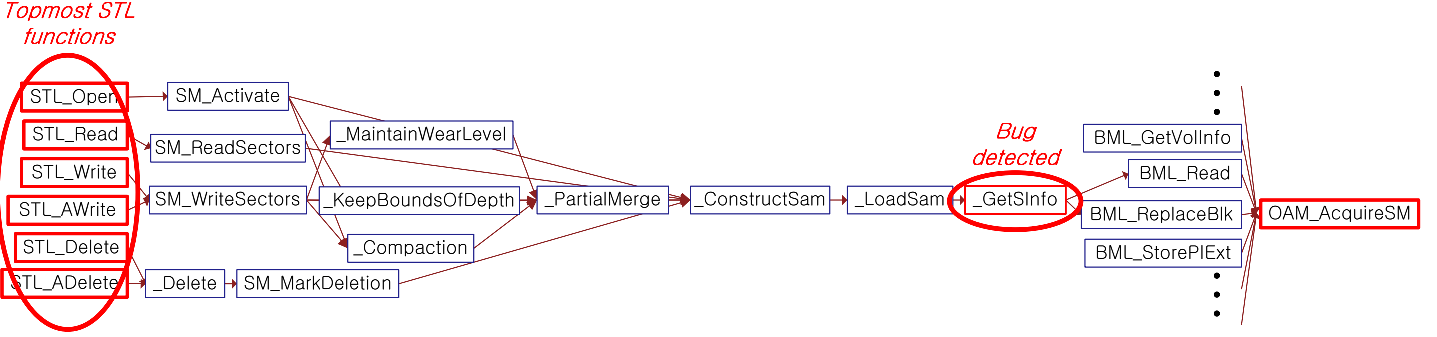


**Figure 7. A partial call graph of the topmost STL functions using the BML semaphore**

#### 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/operational environment should be created so that a logical relation is maintained between the SAMs and the PUs, as shown in Figure 8. In this analysis task, we created a test environment for MSR by specifying the constraints representing this relationship. For example, some of the rules describing a valid environment are as follows:

1. For each logical sector, at least one physical sector that has the same value exists.

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.

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]

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

[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 verifying the requirements.

Note that the total number of SAM and PU configurations increases exponentially as the size of the logical sectors or the number of PUs increases. For example, for data that is six sectors long and distributed over 10 PUs, $2.7\times10^8$ distinct test scenarios exist. 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 calculated from the algorithm of MSR and a given configuration of SAMs and PUs.

#### 5.3.3 Testing Results

We tested MSR for data that was 5 to 8 sectors long and distributed over 5 to 10 PUs. Through the CBMC experiments, no violations were detected. Note that the test environment generated all possible scenarios and CBMC analyzes all generated scenarios. Therefore, compared with randomized testing, this exhaustive analysis capability can provide a higher confidence of the correctness of MSR. The experimental results are illustrated in Figure 9. For example, it took 1471 seconds to test all $2.7 \times 10^8$ scenarios for data that was six sectors long and 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 [16].

## 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. However, due to the state explosion problem [13] and the gap between the target program and the abstract model, model checkers have not been widely used as software validation tools.

Through this project, however, we found that a SAT-based model checker can overcome these weaknesses.

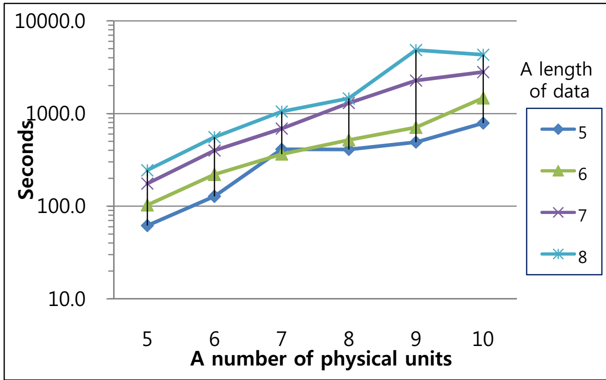


**Figure 9. Time complexity of the MSR analysis**

CBMC eliminates the overheads required to create an abstract formal model, since it can directly analyze a C program. In addition, CBMC can analyze units of codes while consuming only a modest amount of time and memory, although an entire program cannot be analyzed as a whole and the binary libraries used by the target program cannot be analyzed. Furthermore, a CBMC based analysis can be more convenient than actual testing, since CBMC does not require any test harnesses except the environment constraints. In addition, a counter example generated by CBMC serves as an effective debugging aid.

Finally, as demonstrated in Section 5, a CBMC based analysis can detect subtle hidden bugs that are hard to detect through conventional testing. Therefore, a SAT-based model checker can be used as an effective unit test tool, as it can provide a high confidence of the code quality with modest overheads.

### 6.2 Benefits of Constraint-based Exhaustive Testing

Although active research on model based testing has been undertaken [20], the generation of test cases adequate for various test criteria [21] still requires significant human effort. In this project, we avoided this laborious task of explicit test case generation. Instead, we mechanically tested all possible execution scenarios that satisfied the environmental constraints. This approach was successful for unit testing of USP; exhaustive unit testing consumed only modest amounts of time and memory and found hidden bugs.

In addition, even when a set of explicitly generated test cases reaches a complete statement coverage or branch coverage, the absence of error is still not guaranteed, since different input values generate different outputs, even in the same execution path (e.g. overflow error, divide-by-zero error, etc.). Therefore, for unit testing, this exhaustive analysis with constraints can produce a greater confidence in the correctness of the target code while requiring a reduced amount of human effort.

### 6.3 Advantages of a SAT-based Model Checker

Several research projects that apply model checking techniques to directly verify C programs without abstract models exist [14, 8, 19]. Software model checkers such as Blast [9] and SLAM [18] use various abstraction techniques, such as counter example based abstraction refinement (CEGAR), to alleviate the state space explosion problem. However, these approaches suffer from excessive abstraction and limitations of underlying decision theories. Consequently, the analysis results can be inaccurate or the analysis may halt unexpectedly due to a failure to find the 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 BML semaphore usage pattern (see Section 5.2.1); however, Blast unexpectedly halted the analysis of `PriRead()` (see Section 5.1), because it failed to find the proper predicates. Furthermore, we could not expect Blast to analyze MSR correctly, since Blast has a very limited analysis capability in array operations.

CBMC, however, produced accurate analysis results because it transformed a target C program into a SAT formula without abstraction. Then, this formula was solved efficiently by an underlying SAT solver with the help of advanced heuristics. One difficulty of the SAT based model checking is obtaining the upper bounds for the loop unwindings. Although this problem is difficult to solve in general, a user can obtain a good approximation for the unwinding upper bounds after reviewing the target program. In this project, we could obtain the unwinding bounds from the DDS documents and the target code without much difficulty.

## 7 Conclusions 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 Samsung's OneNAND$^{TM}$ flash memory. These bugs included incomplete handling of the semaphore exceptions and a logical bug that did not store the current status of an erase operation that was preempted by a prioritized read operation. These bugs had not been previously detected by Samsung. In addition, we established confidence in the correctness of the 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 verify an entire C program [3]. However, it is still beneficial to use a SAT-based model checker to test units of a target program as demonstrated in this project. Samsung highly valued the project results and, for the next project, we plan to analyze a flash file system to verify data consistency at the event of random power-offs. We also plan to apply the Satisfiability Modulo Theories (SMT) solver [5] instead of a SAT solver to exploit efficient decision procedures, as this can reduce the huge bit-vector equations generated from a target C code.

## Acknowledgments

We would like to thank Prof. Yunja Choi at Kyungpook National University for our valuable discussions on the verification of MSR. Also, we would like to thank the reviewers for their constructive and detailed comments.

## References

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

[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*, pages 621–631, 2007.

[4] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN Workshop*, pages 103–122, 2001.

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

[6] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58, 2003.

[7] A. Chakrabarti, L. de Alfaro, T. Henzinger, M. Jurdzínski, and F. Mang. Interface compatibility checking for software modules. In *Computer Aided Verification (CAV)*, pages 428–441, 2001.

[8] 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*, pages 243–256, 2007.

[9] 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*, 9:505–525, 2007.

[10] E.Clarke, D.Kröening, 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.

[11] N. Eén and N. Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 502–518, 2003.

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

[13] 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*, 2000:176 – 194, 2001.

[14] J.Yang, P.Twohey, D.Engler, and M.Musuvathi. Using model checking to find serious file system errors. In *Operating System Design and Implementation*, pages 273–288, 2004.

[15] L.Zhang and S.Malik. The quest for efficient boolean satisfiability solvers. In *Computer Aided Verification*, pages 313–331, 2002.

[16] M.Kim, Y.Choi, Y.Kim, and H.Kim. Formal verification of a flash memory device driver - an experience report. In *Spin Workshop*, pages 144–159, 2008.

[17] M.Moskewicz, C.Madigan, Y.Zhao, L.Zhang, and S.Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535, June 2001.

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

[19] T.Witkowski, N.Blanc, D.Kröening, and G.Weissenbacher. Model checking concurrent linux device drivers. In *Automated Software Engineering*, pages 501–504, November 2007.

[20] M. Utting and B. Legeard. Practical model-based testing: a tools approach. *Morgan Kaufmann*, 2007.

[21] H. Zhu, P.A.V. Hall, and J.H.R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.