A Comparative Study of Software Model Checkers as Unit Testing Tools: An Industrial Case Study

Moonzoo Kim, *Member*, *IEEE*, Yunho Kim, and Hotae Kim

Abstract—Conventional testing methods often fail to detect hidden flaws in complex embedded software such as device drivers or file systems. This deficiency incurs significant development and support/maintenance cost for the manufacturers. Model checking techniques have been proposed to compensate for the weaknesses of conventional testing methods through exhaustive analyses. Whereas conventional model checkers require manual effort to create an abstract target model, modern software model checkers remove this overhead by directly analyzing a target C program, and can be utilized as unit testing tools. However, since software model checkers are not fully mature yet, they have limitations according to the underlying technologies and tool implementations, potentially critical issues when applied in industrial projects. This paper reports our experience in applying Blast and CBMC to testing the components of a storage platform software for flash memory. Through this project, we analyzed the strong and weak points of two different software model checking technologies in the viewpoint of real-world industrial application—counterexample-guided abstraction refinement with predicate abstraction and SAT-based bounded analysis.

Index Terms—Embedded software verification, software model checking, bounded model checking, CEGAR-based model checking, flash file systems.

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 storage platform software of the flash memory operate correctly. However, as is typical of embedded software, conventional testing methods often fail to detect bugs hidden in the complex storage platform software. This deficiency incurs significant cost to the manufacturers.

Conventional testing has limitations in verifying whether a target software satisfies given requirement specifications since testing does not provide complete coverage. Furthermore, significant human effort is required to generate effective test cases that provide a certain degree of coverage. As a result, subtle bugs are difficult to detect by testing and can cause significant overhead 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 the factors leading up to the scenario where the violation occurred. These limitations were manifest in many industrial projects, including the unified storage platform (USP) software for Samsung's OneNAND flash memory [3]. For example, a multisector read (MSR) function was added to USP to optimize the reading speed (see Section 8). This function, however, caused errors, despite extensive testing and debugging efforts, to the extent that the developers seriously considered removing the feature.

Model checking techniques [26] have been proposed to alleviate the aforementioned weaknesses of the conventional testing methods by automatically exploring the entire state space of an abstract target model. If a violation is detected, the model checker generates a concrete counterexample through which the bug can be conveniently identified. However, model checking techniques are not widely applied in industry since discrepancies exist between the target software and its abstract model. Furthermore, significant additional efforts are required to create an abstract target model, which is not affordable for most industrial projects. Modern software model checkers [34], [25] remove this overhead by directly analyzing a target C program through either automated abstraction (e.g., counterexample-guided abstraction refinement (CEGAR) [18] with predicate abstraction [31]) [6], [21], [8], [14] or limiting the range of analysis (e.g., bounded analysis [10]) [19]. However, since software model checkers are not fully mature yet, they have limitations according to the underlying technologies and tool implementations, which can constitute critical issues when applied to real industrial projects. Therefore, developing strategies for effective use of software model checkers through concrete case studies is very important for successful industrial application. In addition, it is challenging to verify a flash storage platform since it makes heavy use of complex data structures to manage linked lists of physical units and corresponding sector allocation maps (see Section 3.2 and Section 8), while

- *M. Kim and Y. Kim are with the Computer Science Department, KAIST, 373-1 Guseong-dong, Yuseong-gu, Daejon 305-701, South Korea. E-mail: moonzoo@cs.kaist.ac.kr, kimyunho@kaist.ac.kr.*
- *H. Kim is with Samsung Electronics, Linux Lab., 18th floor, R3, Maetan 3 dong 416, Suwon, Kyungki-do, South Korea. E-mail: hotae.kim@samsung.com.*

Manuscript received 30 Sept. 2009; revised 5 Feb. 2010; accepted 8 Feb. 2010; published online 9 July 2010.
Recommended for acceptance by R. Cleaveland.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2009-09-0239.
Digital Object Identifier no. 10.1109/TSE.2010.68.

TABLE 1
Related Case Studies Applying Off-the-Shelf SMCs

Case study	Target domain	Size of a target module (LOC)	SMC tools used	Properties
Groce et al. [32]	Flash file systems	50	Blast, CBMC, and MAGIC	Correctness of the pathname canonizer
Kolb et al. [40]	OPC UA protocol	N/A	Blast	Lock policies and correctness of the message context management and the encoder management
Ku et al. [41]	Server applications	Mostly less than 100	SATABS	Buffer overflow
Mühlberg et al. [45]	Linux device drivers	N/A	Blast	Memory safety and lock policies
Post et al. [46]	Linux device drivers	N/A	CBMC	Memory safety and lock policies, etc
Kim et al. (this article)	Flash file systems	150 to 2450	Blast and CBMC	Lock policies (Section 7.1), correctness of preemption (Section 6), exception handling (Section 7.2), and multi-sector read operation (Section 8)

most embedded systems are control-oriented without complex data structures.

In this project, we applied two software model checkers to find hidden bugs in the USP for Samsung's OneNAND flash memory, namely, Blast [8] and CBMC [19], which utilize CEGAR with predicate abstraction and SAT-based bounded analysis, respectively. We performed an in-depth comparative study of these two different software model checking techniques and summarized the lessons learned regarding the advantages and weaknesses of software model checking for industrial projects.

We selected these two software model checkers for the following reasons. First, Blast and CBMC are relatively stable compared to other tools. The development of both tools has progressed for almost a decade, and both have user communities. Thus, it is likely that many of the bugs and limitations in the original versions have been fixed and USP can therefore be verified using these tools without much difficulty. Second, the source code of CBMC and that of Blast are publicly available. Although model checking technique claims to be a "push-button" technique, a user can improve the verification performance or bypass the underlying limitations of the tool through knowledge of the internal behaviors of the model checking tools, which is difficult to obtain without the tool source code. In addition, it is occasionally necessary to customize the tool depending on the needs of the projects in an industrial setting. Finally, we have expertise in the area of both Blast and CBMC through several years of experience. Hence, we can conduct comparative experiments in a fair and technically sound manner. Therefore, given that the goal here is to determine practical strengths and weaknesses of software model checking techniques on actual industrial software (i.e., flash storage platform software), Blast and CBMC can adequately serve this purpose.

This paper is organized as follows: Section 2 discusses related work. Section 3 describes background of the OneNAND flash memory and USP. Section 4 provides an overview of CEGAR with predicate abstraction and SAT-based bounded analysis techniques. Section 5 explains the scope and methodology used in the project. Sections 6-8 present the experiments of analyzing the USP using Blast and CBMC. Section 9 summarizes the lessons learned from this project, and Section 10 concludes with future work. In addition, for the convenience of the readers, Table 4 at the end of the paper explains the list of acronyms in the domain of flash storage platforms.

2 Related Work

This section discusses related case studies in which software model checkers (SMCs) are applied to achieve high reliability of the target systems. Although SMC was originally proposed nearly a decade ago, this promising technique has not been widely adopted by industries. One main reason is that it is difficult for field engineers to estimate how successfully an SMC will detect bugs in their target programs and how much effort is required to apply an SMC to the projects. As many SMC research papers only demonstrate the strengths of an SMC and do not describe their limitations or efforts related to actual projects, detailed industrial case studies can persuade practitioners to adopt an SMC with an understanding of the pros and cons of the application of the SMC to their projects.

There have been several studies in which an SMC was applied to various target systems such as automotive software [47], microcontroller programs [48], Linux device drivers [45], [46], file systems [52], network filters [13], a protocol stack [40], and server applications [41]. For a flash storage platform, which is our main target domain, one recent study [32] analyzes flash file systems as an effort of the mini-challenge [35]. However, the majority of verification research [36], [28], [11] on flash file systems focus on the specifications of the file system design and not on an actual instance of implementation.

We focus on the related case studies (as shown in Table 1) which report the application of off-the-shelf SMCs to various target systems,[1] as the goal here is to determine the applicability of an SMC to industrial projects in which the development of custom tools optimized for a project is often not feasible due to tight time-to-market restrictions.

Groce et al. [32] applied various verification techniques to their flash file systems. In particular, they applied Blast, CBMC, and MAGIC [14] to the pathname canonizer of the file systems. It was reported that Blast and MAGIC failed to extract a model from the pathname canonizer and it failed as well to find a proof or a counterexample. In contrast, CBMC was able to analyze the pathname canonizer with pathnames containing up to six characters. The authors in [32] suspected

1. "N/A" in the third column of Table 1 indicates no available information on the size of the target module. For a more extensive list of case studies using Blast and CBMC, see http://mtc.epfl.ch/software-tools/blast/index-epfl.php and http://www.cprover.org/cbmc/applications.shtml, respectively.

that heavy use of complex data structure in the file system might cause failures to SMCs, which was similarly observed in our case study. Kolb et al. [40] used Blast to verify the components of the Object linking and embedding for Process Control Unified Architecture (OPC UA) [42] protocol stack with regard to lock policy, message context management, and encoder management. They reported that several bugs were detected in a few seconds to a few minutes, but tool failures and false alarms were observed as well. Ku et al. [41] applied SATABS [21] to detect buffer overflows on the 300 sample codes of server applications such as apache and sendmail. Although 90 percent of the sample codes are less than 100 lines long, they could analyze the sample codes with only minimal buffer sizes (1 and 2) to avoid explosion of analysis time. They reported that SATABS found buffer overflows in 71.4 percent of the sample codes in 600 seconds. Mühlberg and Luttgen [45] and Post and Küchlin [46] verified the same Linux device drivers with Blast and CBMC, respectively. They analyzed eight device drivers with regard to use-after-free (UAF) bugs and other eight device drivers with regard to the lock policies. An interesting observation was made that neither Blast nor CBMC outperformed in these two case studies. For example, both tools detected the UAF bugs in five out of eight device drivers. However, Blast and CBMC failed on different targets. Mühlberg and Luttgen [45] failed to detect the UAF bug in `dpt-i2p.c` but found the bug in `pciehp_ctrl.c`. In contrast, Post and Küchlin [46] failed to detect the UAF bug in `pciehp_ctrl.c` but found the bug in `dpt-i2p.c`. Mühlberg and Luttgen [45] reported that the lock policy bugs were detected in a few seconds to minutes, but no performance information was available on the experiments of [46].

There are several important issues in the related case studies. First, the related case studies often fail to describe real effort required in the application of an SMC to a target system. For example, most case studies (implicitly) assume that requirement properties can be easily identified and written in assertion statements; this is not true in actual industrial projects in which a user has to understand and capture the requirement as assertions. For example, Post and Küchlin [46] mentioned that they extracted API conformance rules from the Linux documentation, but no additional explanation was given in the paper. In contrast, we describe the effort required to apply an SMC to the flash storage platform software in an industrial setting (see Section 5).

Second, apart from Groce et al. [32], few case studies compare different SMCs on target systems. The selection of the SMC has a crucial impact on the verification project. However, it is difficult to decide which SMC to use in a project as different SMCs have their own strengths and weaknesses (see [45] and [46]). This paper compares Blast and CBMC on the flash storage platform in a fair and technical manner.

Finally, although success stories exist regarding the application of SMCs in industrial projects, the limitations of SMCs should be considered as well. For example, in spite of the well-known success of the CEGAR-based SMC on Windows device drivers [5], there are case studies that report the limitations of a CEGAR-based SMC in which the inaccurate handling of arrays [41] and pointers [40], [45] is outlined (see Section 8.2 and Section 9.2).

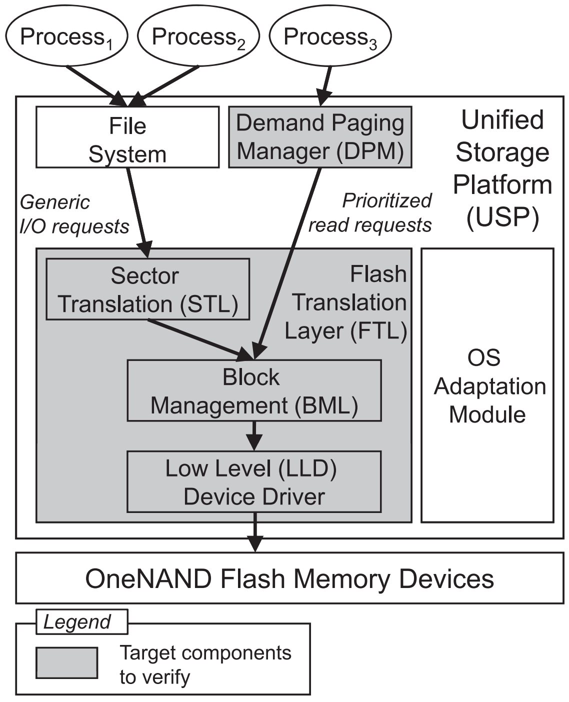
Process$_1$
Process$_2$
Process$_3$
File System
Demand Paging Manager (DPM)
Unified Storage Platform (USP)
Generic I/O requests
Prioritized read requests
Sector Translation (STL)
Flash Translation Layer (FTL)
Block Management (BML)
Low Level (LLD) Device Driver
OS Adaptation Module
OneNAND Flash Memory Devices
Legend
Target components to verify

Fig. 1. Overview of the USP.

3 The Unified Storage Platform for OneNAND Flash Memory

3.1 Overview of the Unified Storage Platform

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 supports byte-addressing and executes software in place (XIP) without loading the binary into the main memory, whereas NAND cannot (see Section 3.2).[2] OneNAND is a single chip comprised of 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 a process executes a program in OneNAND, the corresponding page of the program is loaded into the internal RAM in OneNAND using the demand paging manager (DPM) for XIP.

The USP is a software solution for OneNAND-based embedded systems. Fig. 1 presents an overview of the USP: It manages both code storage and data storage. USP allows processes 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. FTL is a core part of the storage platform for flash memory since logical data can be mapped to separated physical sectors due to the physical characteristics of flash memory (see Section 3.2 and Section 8). FTL consists of the 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 processes are fulfilled through the file system, STL, BML, and LLD, in that order. Although the USP allows concurrent I/O requests from

2. The NAND and NOR technologies are explained in more detail in [2].

multiple processes through the STL, the BML operations must be executed sequentially, not concurrently. For this purpose, the BML uses a binary semaphore to coordinate concurrent I/O requests from the STL (see Section 7).

In addition to generic I/O requests, a process can make a *prioritized read request* for executing a program through the DPM and this request goes directly to the BML. A prioritized read request from the DPM can preempt generic I/O operations requested by STL. After the prioritized read request is completed, the preempted generic I/O operations should be resumed again (see Section 6).

In this project, we analyzed the FTL and the DPM components of the USP.

3.2 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*. Operations are either read/write operations on a page or erase operations on a block.[3] NAND can write data only on an empty page and the page can be emptied by erasing the block containing the page. Therefore, when new data are written to the flash memory, rather than directly overwriting old data, the data are 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 (PUs). STL manages the mapping from the logical sectors (LSs) to the physical sectors (PSs). 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.[4]

Fig. 2 illustrates the mapping from logical sectors to physical sectors where one unit contains one block and a block consists of four pages, each of which has one sector. 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 ($\mathrm{SAM1}[0] = 0$). The user continues to write the LS1 of LU7, and the LS1 is subsequently stored into the PS1 of PU1 ($\mathrm{SAM1}[1] = 1$) (see Fig. 2a). The user then overwrites LS1 and LS0 in order, which results in $\mathrm{SAM1}[1] = 2$ and $\mathrm{SAM1}[0] = 3$ (see Fig. 2b). Finally, the user adds the LS2 of LU7, which adds a new empty physical unit PU4 to LU7 and yields $\mathrm{SAM4}[2] = 0$ (see Fig. 2c).

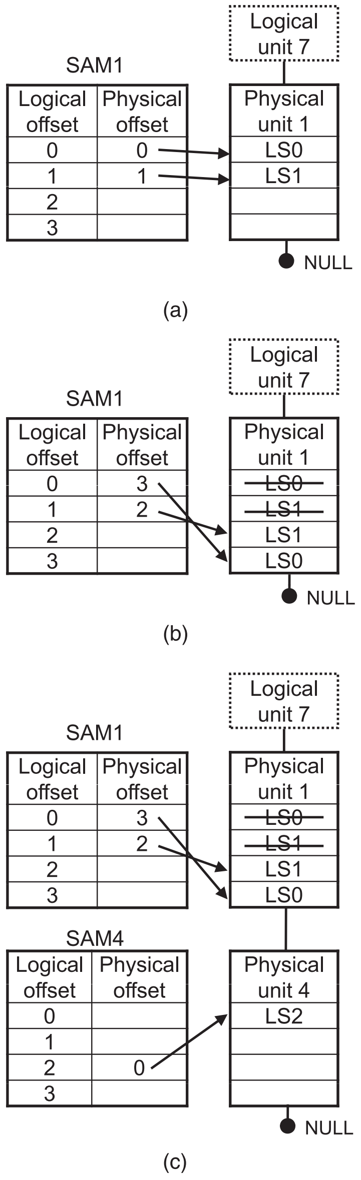

Fig. 2. Mapping from logical sectors to physical sectors. (a) Mapping after LS0 and LS1 is written fresh in order. (b) Mapping after LS1 and LS0 is updated in order. (c) Mapping after LS2 is written fresh.

4 CEGAR WITH PREDICATE ABSTRACTION AND SAT-BASED BOUNDED ANALYSIS

This section gives a brief overview of CEGAR with predicate abstraction as well as SAT-based bounded model checking, as used by Blast and CBMC, respectively.

4.1 Counterexample-Guided Abstraction Refinement with Predicate Abstraction

In general, software programs are infinite-state systems due to variables of infinite domains and recursion. Thus, it is not possible to apply model checking to programs without abstraction. In addition, to alleviate the state explosion problem, many abstraction techniques [23] have been studied to simplify the target program under analysis. One successful abstraction technique in software model checking is predicate abstraction [31]. Predicate abstraction is an abstraction technique whose abstract domain is constructed using a given set of predicates over program variables, which are generated from the program text. However, a model checker may generate a spurious counterexample from the abstract target system due to a coarse abstraction function. In such a case, the current abstraction function should be refined to avoid this type of spurious counterexample. The CEGAR technique [18] can automatically and iteratively refine the abstraction function using the spurious counterexample generated due to the coarse abstraction.

3. Flash may support writing a page *partially*, which is equivalent to sector-based writing.

4. Gal and Toledo [29] describe the algorithms and data structures for flash storage platforms in detail.

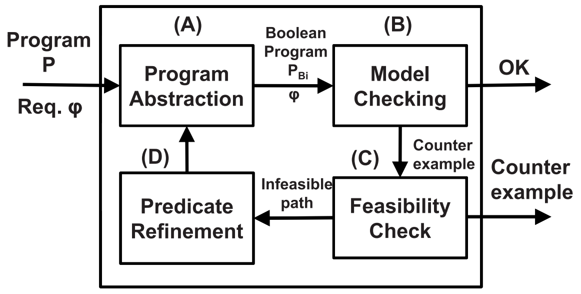

Fig. 3. Overview of CEGAR with predicate abstraction.

Software model checkers utilizing CEGAR with predicate abstraction [6], [8], [21], [14] automatically abstract a concrete program P into a Boolean program P_B through an iterative process. The goal of the combined technique is to develop an automated procedure to transform P conservatively into the Boolean program P_B so that P_B is smaller than P and the model checking of P_B is sound in terms of the given requirement ϕ. The overall process of CEGAR with predicate abstraction is given below (see Fig. 3):

1. Initially, a given program P is abstracted into a Boolean program P_{B_0} whose set of predicates is either an empty set or a set of predicates specified by a user.
2. Subsequently, P_{B_0} is model checked; if P_{B_0} satisfies ϕ, the entire model checking process terminates as the abstraction is conservative.
3. If not, a counterexample is generated and checked as to whether or not it is spurious. If the counterexample is not spurious, the model checking process terminates.
4. If the counterexample is spurious, new predicates are obtained from the counterexample to rule it out. A refined version of the Boolean program P_{B_1} is then constructed at 1 and the entire model checking process reiterates with P_{B_1}.

Blast [8] is a software model checker for C programs based on CEGAR with predicate abstraction. In addition, Blast exploits the lazy abstraction technique to enhance analysis performance so that a large program can be analyzed efficiently. Blast receives a C program as its input and checks whether or not the target C program satisfies a requirement ϕ written in monitoring automata and/or `assert` statements. For computing the next abstract reachable states, Blast uses Simplify [24] as an internal decision procedure which solves linear arithmetic and uninterpreted functions. In addition, to obtain new predicates, Blast uses a decision procedure such as FOCI [44] or CSIsat [9] to calculate interpolants from infeasible paths. Consequently, Blast models complex C operations (i.e., bitwise operators) that cannot be handled by these decision procedures as uninterpreted functions, which may cause false alarms.

4.2 SAT-Based Bounded Model Checking

Bounded model checking [10] unwinds the control flow graph of a target program P for a fixed number of times n, and then checks if an error can be reached within these n steps. SAT-based bounded model checking [17] unrolls the target program Pn times, transforms this unrolled program into the SAT formula ϕ_P, and then checks whether P can reach an error within this bound n by checking the satisfiability of ϕ_P [33]. In spite of the NP-complete complexity, structured Boolean satisfiability (SAT) formulas generated from real-world problems are successfully solved by SAT solvers in many cases. Modern SAT solvers exploit various heuristics [53] and can solve some large SAT formulas containing millions of variables and clauses in modest time [4]. To use a SAT solver as a bounded model checker to verify whether a given C program P satisfies a requirement property R, it is necessary to translate both P and R into Boolean formulas ϕ_P and ϕ_R, respectively. A SAT solver then determines whether $\phi_P \wedge \neg\phi_R$ is satisfiable: If the formula is satisfiable, it means that P violates R; if not, P satisfies R (note that each satisfying assignment to ϕ_P represents a possible execution trace in P).

```
x=x+y;
if (x!=1)x=2;
else x=x+2;
assert(x<3);
```

(a)

```
x1=x0+y0;
if (x1!=1)x2=2;
else x3=x1+2;
x4=(x1!=1)?x2:x3;
assert(x4<3);
```

(b)

$$\phi_P \equiv x_1=x_0+y_0 \wedge x_2=2 \wedge x_3=x_1+2$$
$$\wedge(x_1!=1 \rightarrow x_4=x_2)\wedge(x_1=1\rightarrow x_4=x_3)$$
$$\boldsymbol{\phi_R \equiv x_4 < 3}$$

(c)

Fig. 4. Example of translating a C program into a Boolean formula. (a) A target C program. (b) The corresponding SSA statements. (c) The corresponding Boolean formulas ϕ_P and ϕ_R.

A brief sketch of the translation process follows [19]. We assume that a given C program is already preprocessed. First, the C program is transformed into a canonical form, containing only `if`, `goto`, and `while` statements without side effect statements such as `++`. Then, the loop statements are 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 `assert(!e)` that guarantees that the program does not execute more iterations. A similar procedure is applied to loops containing the backward `goto` statements. 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 forward `goto` statements. This C program is transformed into static single assignment (SSA) form. Fig. 4b illustrates the SSA form of the C program in Fig. 4a. This SSA program is converted into corresponding bit-vector equations through combinatorial circuit encoding and the final Boolean formula is a conjunction of all of these bit-vector equations. For example, Fig. 4c illustrates a Boolean conjunction of the SSA statements ϕ_P; however, they are not converted into bit-vector equations yet.

Note that bounded model checking is incomplete on infinite-state systems. Thus, several approaches based on k-induction [50] or interpolation [43] have been studied to make SAT-based bounded model checking complete. Although bounded model checking may be inefficient in the presence of deep loops, it can be used as an effective debugging method up to small loop bounds (see Section 7.2.2).

CBMC [19] is a bounded model checker for ANSI-C programs. CBMC receives a C program as its input and analyzes all C statements (e.g., pointer arithmetic, arrays, structs, function calls, etc.) with bit-level accuracy and translates the C program into an SAT formula automatically. A requirement property is written as an assert statement in a target C program. The loop unwinding bound n_l for loop l can be given as a command line parameter; for simple loops with constant upper bounds, CBMC automatically calculates n_l. If $\phi_P \wedge \neg\phi_R$ is satisfiable, CBMC generates a counterexample that shows a step-by-step execution leading to the violation of the requirement property.

5 Project Overview

5.1 Project Scope

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 and spent the first three months reviewing the USP design documents and code to become familiar with the USP and OneNAND flash. However, we estimate that the net total effort for this project was five man-months (three man-months were spent working to understand the USP and OneNAND flash because most of us were new to the flash storage platform) as we did not fully dedicate our time to this project. It took an additional one man-month to apply Blast to the project after we had initially applied CBMC. Most parts of the USP were written in C and a small portion of the USP was written in the ARM assembly language. The source codes of the FTL and the DPM are roughly 30,000 lines long.

The goal of this project was to increase the reliability of the USP by finding hidden bugs that had not yet been detected. For this purpose, it was not enough to check the predefined API interface rules as found in other research work [7], [15]. 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* demanded significant effort since it required knowledge of the target system requirements, the high-level design, low-level implementation, and mapping between the design and the implementation. Although most formal verification research assumes that these code-level properties are given from somewhere, in real industrial projects we must define these properties ourselves.

To this end, we applied a *top-down* approach to identify the code-level properties from the high-level requirements (see Fig. 5). First, we selected the target requirements from the USP documents. The USP has a set of elaborated documents as follows:

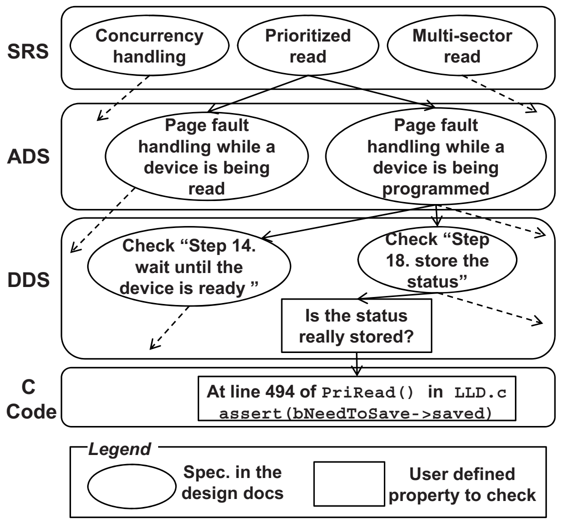
SRS
Concurrency handling
Prioritized read
Multi-sector read
ADS
Page fault handling while a device is being read
Page fault handling while a device is being programmed
DDS
Check "Step 14. wait until the device is ready "
Check "Step 18. store the status"
Is the status really stored?
C Code
At line 494 of PriRead() in LLD.c
assert(bNeedToSave->saved)
Legend
Spec. in the design docs
User defined property to check

Fig. 5. Top-down approach used to identify the code-level properties to verify.

- software requirement specifications (SRSs),
- architecture design specifications (ADSs),
- detailed design specifications (DDSs):
    - DPM, STL, BML, and LLD DDSs.

The SRS contains both functional and nonfunctional requirement specifications with priorities. We selected three functional requirements with very high priorities (see Section 5.2). Then, from the selected functional requirements, we investigated the relevant ADS, DDS, and corresponding C code fragments to specify concrete code-level properties (see Section 5.3). 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.

5.2 High-Level Requirements

The SRS document specifies 13 functional requirements and 18 nonfunctional requirements for the USP. Each requirement specifies its own priority. There were three functional requirements that have "very high" priorities:

- *Support prioritized read operation.* In order to execute a program, the DPM loads a code page into the internal RAM when a page fault exception occurs. Since the fault latency should be minimized, the FTL should serve a read request from the 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 the USP. The first behavior is concurrency among multiple generic I/O operations; the second is concurrency between generic I/O operations and a prioritized read operation. The USP should handle these two types of concurrent

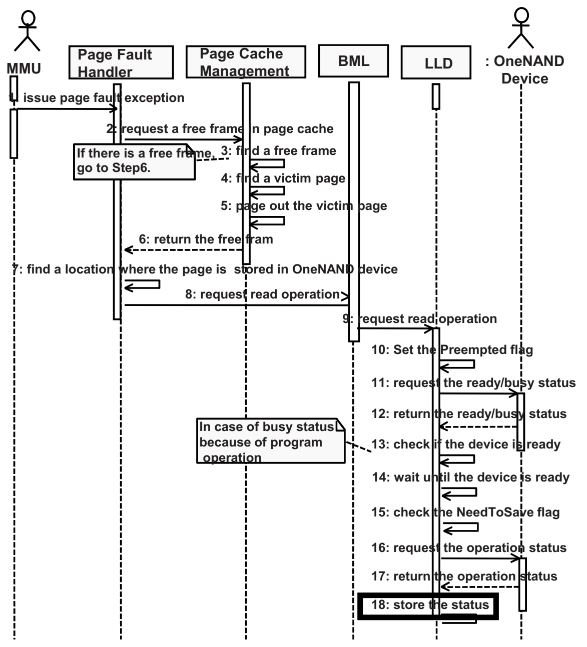
MMU
Page Fault Handler
Page Cache Management
BML
LLD
: OneNAND Device
1: issue page fault exception
2: request a free frame in page cache
If there is a free frame, go to Step6.
3: find a free frame
4: find a victim page
5: page out the victim page
6: return the free fram
7: find a location where the page is stored in OneNAND device
8: request read operation
9: request read operation
10: Set the Preempted flag
11: request the ready/busy status
12: return the ready/busy status
In case of busy status because of program operation
13: check if the device is ready
14: wait until the device is ready
15: check the NeedToSave flag
16: request the operation status
17: return the operation status
18: store the status

Fig. 6. Sequence diagram of page fault handling while a device is being programmed.

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, the 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 three requirements above and analyzing the relevant structures described in the ADS. For example, as depicted in Fig. 5, a functional requirement on a prioritized read operation is related to the page fault handling mechanisms, which are described in the 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. Sections 6-8 describe experiments for verifying these three requirements, respectively.

5.3 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 Fig. 5, 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 provided (see Fig. 6).

The 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. For example, Step 18 in Fig. 6 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 an `assert` statement at the corresponding location of `PriRead`.

5.4 Environment Models

In addition, we built an *environment model* for the unit under analysis which was similar to a test harness for unit testing. For example, MSR assumes that SAMs and PUs satisfy several constraint rules (see Section 8.1) and does not check whether a given input (i.e., a configuration of SAMs and PUs) actually satisfies those constraints. Thus, if an invalid input which does not satisfy those constraints is given to MSR during model checking, the result is unreliable. In other words, given an invalid input, MSR may violate the given properties, even when MSR is correct, or vice versa. Therefore, it is necessary to build an environment model to feed only valid cases to MSR.

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

$$\texttt{\_\_CPROVER\_assume}(0 <= \texttt{nDev}\ \&\&\ \texttt{nDev} <= 7),$$

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:

$$(!(\texttt{NAND[nDev].type} == \texttt{SMALL})\ ||\ \texttt{nPbn} < 256)\ \&\&$$
$$(!(\texttt{NAND[nDev].type} == \texttt{LARGE})\ ||\ \texttt{nPbn} < 2048)$$

An environment model specifies not only possible values of function parameters, but also the global data being used by `STL_Read`. For example, a global object `CTX` contains a shared context for each OneNAND device and it is retrieved by `STL_Read`. Based on the STL 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.

$$\texttt{CTX.physSctsPerUnit} == \texttt{CTX.blksPerUnit}$$
$$* \texttt{CTX.pgsPerBlk} * \texttt{CTX.sctsPerPg}$$

For the analysis using Blast, we used `if(!constraint) goto out` to describe an environment model/constraints in a similar manner. With such an explicit environment model, we can improve the accuracy of the analysis and reduce false alarms generated from invalid scenarios.

6 PRIORITIZED READ OPERATION

A prioritized read operation is implemented in the `PriRead` function in `LLD.c` of the LLD layer. This

```
01:...
02:Location: id=5#16 src="LLD.c"; line=406
03:  Block(*(pstReg@PriRead ).nInt =0;
04:         *(pstReg@PriRead ).nCmd = 176;
05:         bEraseCmd@PriRead = 1;)
06:Location: id=5#18 src="LLD.c"; line=410
07:  Pred(bEraseCmd@PriRead? != 0)
08:...
09:Location: id=5#46 src="LLD.c"; line=494
10:  FunctionCall(__assert_fail(
11:  "!(pstInfo->bNeedToSave==1)||saved" ... ))
12:Error found! The system is unsafe :-(
```

Fig. 7. Counterexample generated by Blast.

function is 234 lines long. It has two simple loops whose upper bound are constants, but does not have a subfunction call. PriRead 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. Instead, we used Blast and CBMC to automatically test all valid value combinations of the function parameters and global data to check whether the status of a preempted operation is correctly saved, so as to resume the preempted operation later. This property is written as the following assert statement at line 494 of PriRead:

$$\texttt{assert}(!(\texttt{pstInfo}- > \texttt{bNeedToSave}) \mid\mid \texttt{saved})$$

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 Blast version 2.5 and CBMC version 2.6 with MiniSAT 1.1.4 [27].

6.1 Analysis by Blast

With -foci -cref options (i.e., using the FOCI interpolant solver and pointer aliasing information for counterexample analysis), Blast iterated the CEGAR process three times and added two new predicates, bEraseCmd==0 and pstInfo->bNeedToSave==1. Blast took 0.18 seconds and consumed 6.6 megabytes of memory to detect a violation of the property. We confirmed that the counterexample generated by Blast (see Fig. 7) was real through manual analysis, since Blast might generate a false alarm (see Section 4.1).

The counterexample generated by Blast is a sequence of blocks (e.g., lines 2-5 in Fig. 7) and predicates (e.g., lines 6-7) in the reachability tree [8] generated from the target program. The partial abstract reachability tree generated from PriRead has 51 nodes and its depth is 18. The counterexample consists of 18 steps in total and illustrates that PriRead does *not* save the current status of an erase operation (see lines 9-12), when the erase operation is preempted by a prioritized read operation. Note that line 5 indicates that the current operation is an erase operation because bEraseCmd is assigned as 1. After fixing the bug, Blast did not find a violation anymore and the verification consumed 6.3 megabytes of memory in 0.13 seconds.

6.2 Analysis by CBMC

CBMC translated PriRead with the assert statement into an SAT formula containing one million Boolean variables and 1,340 clauses. We applied - -slice - -no-bounds-check - -no-pointer-check - -no-div-by-zero-check

```
01:...
02:State 14 file LLD.c line 408
03:  LLD::PriRead::1::bEraseCmd=1
04:State 15 file LLD.c line 412
05:  LLD::PriRead::1::1::2::nWaitingTimeOut...
06:...
07:Violated property:
08:  file LLD.c line 494 function PriRead
09:  assertion !(_Bool)pstInfo->bNeedToSave...
10:VERIFICATION FAILED
```

Fig. 8. Counterexample generated by CBMC.

options to make a fair comparison between the Blast experiments and the CBMC experiments.[5] Note that we did not provide loop upper bounds as options, which were obtained automatically by CBMC. CBMC analyzed PriRead and found a violation in 8 seconds (including both SAT-translation time and SAT-solving time) after consuming 325 megabytes of memory. The counterexample generated by CBMC (see Fig. 8) is a sequence of assignment statements, not including predicates. The counterexample consists of 12 steps in total.

Note that no manual analysis is necessary to check whether the counterexample is real or not since CBMC does not generate false alarms, unlike Blast. Line 3 in Fig. 8 indicates that the current operation is an erase operation and lines 7-10 show a violation of the property. After fixing the bug, CBMC did not find a violation anymore and the verification consumed 259 megabytes of memory in 4.86 seconds.

7 Concurrency Handling

7.1 BML Semaphore Usage

Although the USP allows concurrent I/O requests through the STL, the 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, the BML uses a binary semaphore to coordinate concurrent I/O requests from the 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).
- A semaphore should be released before a function returns unless the semaphore operation creates an error.

Fourteen BML functions that use the BML semaphore exist. Each function is 150 lines long (excluding comments) on average. We inserted an integer variable smp to indicate the status of the semaphore (1 when the semaphore is released, 0 otherwise) and simple codes to decrease/increase smp at the corresponding semaphore operations in these 14 BML functions. We verified the following two properties:

5. We applied the same options to all CBMC experiments in this paper. Note that Blast does not check pointers, arrays, nor divide-by-zero, unless a user adds supplementary checking codes for these properties. Also, Blast can be considered as performing slicing through CEGAR with predicate refinement.

TABLE 2
Performance of Analyzing the BML Semaphore Operations

Function	LOC	Loop bound	Blast Time (sec)	Blast Mem (MB)	CBMC Time (sec)	CBMC Mem (MB)
`_Close`	101	No loop	1.3	12	1.8	38
`BML_Copy`	216	Unknown	1.3	12	(14.5)	(135)
`BML_CopyBack`	248	Unknown	1.7	13	(14.4)	(111)
`BML_EraseBlk`	127	<10	1.8	13	5.7	53
`BML_FlushOp`	82	<10	1.5	12	1.8	38
`BML_GetVolInfo`	57	No loop	1.1	12	1.9	38
`BML_IOCtl`	174	<10	2.3	13	1.8	38
`BML_MEraseBlk`	202	Unknown	1.3	12	(39.4)	(237)
`BML_MRead`	167	Unknown	1.2	12	(5.9)	(62)
`BML_MWrite`	178	Unknown	1.2	12	(7.2)	(72)
`BML_Read`	236	<10	1.9	13	6.2	59
`BML_ReplaceBlk`	66	No loop	1.3	12	1.7	31
`BML_StorePIExt`	49	No loop	1.2	12	1.9	32
`BML_Write`	195	<10	2.0	13	7.3	37
Average	150		1.5	12	8.0	70

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

The target BML function is analyzed without its subfunctions through overapproximation for the sake of the analysis performance. Both Blast and CBMC found no violations of the above two properties in the 14 BML functions. The reachability trees generated from the 14 BML functions had 177 nodes on average. The SAT formulas translated from the BML functions (with loop upper bound 10) have 6,300 Boolean variables and 10,400 clauses on average. The performance of the analysis is shown in Table 2. Blast analyzed the BML functions five times faster and consumed only one-sixth of memory compared to CBMC. Note that the performance of CBMC varied much according to the loop structure of the target BML functions. The loop characteristics of the 14 BML functions are as follows and Table 2 reflects those characteristics:

- No loop inside (four functions):
  `_Close`, `BML_GetVolInfo`, `BML_ReplaceBlk`, and `BML_StorePIExt`
- Loops whose upper bound are constants less than 10 (five functions):
  `BML_EraseBlk`, `BML_FlushOp`, `BML_IOCtl`, `BML_Read`, and `BML_Write`
- Loops bounded by a user given value (i.e., an amount of data to read) (five functions):
  `BML_Copy`, `BML_CopyBack`, `BML_MEraseBlk`, `BML_MRead`, and `BML_MWrite`

Therefore, by setting the unwinding upper bound to 10, 9 out of the 14 BML functions were analyzed completely. For the remaining five functions which receive an unbounded user parameter, we set loop upper bounds as 10 and could obtain only limited results from CBMC (the CBMC results for those five functions are written in parentheses in Table 2).

7.2 Handling Semaphore Exception

The BML semaphore operation might cause an exception depending on the hardware status. Once such a BML semaphore exception occurs, USP cannot operate correctly unless a reinitialization 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 their caller when a semaphore operation raised 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. Fig. 9 presents a partial call graph of the topmost STL functions (depicted in the leftmost area of Fig. 9) 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, an interprocedural analysis of functions in nine levels should be performed (i.e., `STL_Write` to `OAM_AcquireSM`).[6]

We added a global variable `SMerr` to indicate when a semaphore exception is raised. Then, we were able to verify whether the semaphore exception had been correctly propagated to the file system by checking 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:

$$\texttt{assert(!(SMerr == 1)||nErr == STL\_CRITICAL\_ERR)}$$

We set 1 hour as a timeout threshold for the experiments.

7.2.1 Analysis by Blast

At the first attempt, a false alarm was generated due to bitwise operations which were handled as uninterpreted functions by Blast. Return flags of the STL functions and its subfunctions were defined through bitwise operations, and Blast could not distinguish different return flags and raised a false alarm. Even after modifying the return flags as constants without bitwise operations, however, Blast with FOCI continued to raise other false alarms and some false alarms could not even be suppressed by modifying the target code nor by adding predicates.

For example, in the counterexample below, after `nNumOfScts=1` and `nSctIdx=0` in the block starting from line 3,649 in `_GetSInfo`, Blast considers that `nSctIdx >= nNumOfScts` in the block starting from line 3,655, which is clearly not true as there is no statement between these two blocks. Therefore, there is no simple remedy to suppress a false alarm caused by this erroneous step.

```
...
Location: id=28#4 src="ALL_STL.c";
 line=3649
Block( ...
  nNumOfScts@_GetSInfo = 1;
  nSctIdx@_GetSInfo = 0;)
Location: id=28#5 src="ALL_STL.c";
 line=3655
  Pred(nSctIdx@_GetSInfo>=nNumOfScts@_
   GetSInfo)
...
```

6. To reduce the analysis complexity, we excluded bad block management functions and the LLD layer in the experiments through overapproximation.

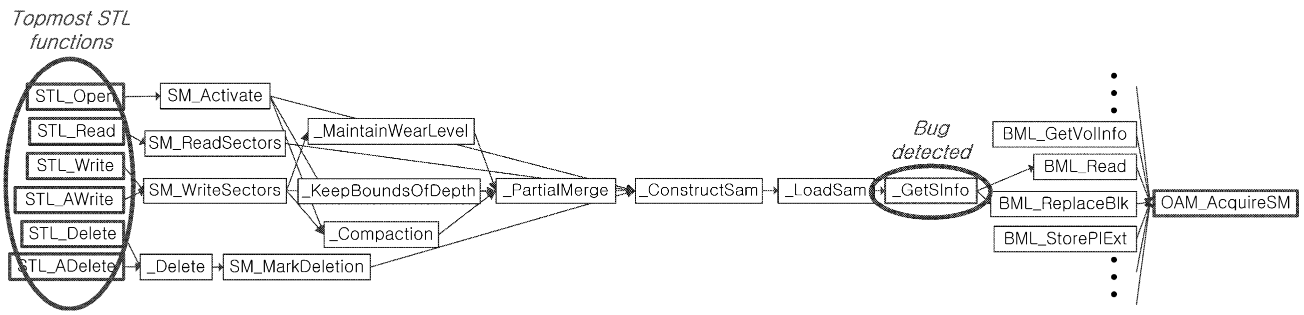

Fig. 9. Partial call graph of the topmost STL functions using the BML semaphore.

This erroneous behavior was caused by the limitation of FOCI, which could not solve full linear arithmetic formulas, but only difference logic formulas [16]. When CSIsat was used instead of FOCI, Blast generated counterexamples for `STL_Read`, `STL_Delete`, and `STL_ADelete`. After reviewing the counterexamples, it was found that the subfunction `_GetSInfo` had a bug. When `_GetSInfo` called `BML_Read`, `_GetSInfo` might *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_Read`, `STL_Delete`, and `STL_ADelete`. However, Blast spent more than 1 hour to perform pointer analysis for `STL_Open`, `STL_Write`, and `STL_AWrite`, and could not complete the analysis. The performance statistics on these experiments are described in Table 3, where LOC counts all subfunctions in the call graph.

After this bug was fixed, Blast spent more than 1 hour and failed again to generate a verification result for `STL_Open`, `STL_Write`, and `STL_AWrite`. Furthermore, for the bug-fixed code, Blast raised false alarms for `STL_Read`, `STL_Delete`, and `STL_ADelete` due to the limitations of Blast for analyzing nested field accesses (e.g., Blast may analyze `pA->pB->pC` incorrectly and cause a false alarm), which had no easy remedy.

7.2.2 Analysis by CBMC

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 only the scenarios where all loop bodies were executed once or passed. In this setting, CBMC generated counterexamples for all six STL functions. After reviewing the counterexample, it was found that a subfunction `_GetSInfo` had a bug, as found in the Blast experiments. The generated SAT formulas contain 4.4×10^6 variables and 1.8×10^7 clauses on average. As shown in Table 3, the analysis time of CBMC was five times longer and memory consumption was 15 times larger than those of Blast. After fixing `_GetSInfo`, CBMC did not find a violation anymore and completed the verification task consuming 3,201 megabytes of memory in 120.5 seconds on average. This verification task was faster than the previous bug-finding task since the bug-fixed code executed only exception handling logic in the presence of a BML semaphore exception, while the original code executed most logic since a BML semaphore exception was lost.

TABLE 3
Performance of Analyzing Propagation of BML Semaphore Exception

Call graph root	LOC (call graph)	Blast Time (sec)	Blast Mem (MB)	CBMC Time (sec)	CBMC Mem (MB)
`STL_Read`	1654	109.6	19	131.2	1762
`STL_Write`	2646	T/O	T/O	1404.9	3718
`STL_Awrite`	2611	T/O	T/O	1014.9	3244
`STL_Delete`	2489	154.0	23	565.9	2139
`STL_Adelete`	2497	148.8	22	808.0	2196
`STL_Open`	2785	T/O	T/O	324.1	5781
Average	2447	137.5	21.5	708.2	3140

8 Multisector Read (MSR) Operation

The USP uses a mechanism to simultaneously read as many multiple sectors as possible in order to improve the reading speed. Due to the nontrivial traversal of data structures for the logical-to-physical sector mapping (see Section 3.2), the function for the MSR is 157 lines long and highly complex, having four-level nested loops. Fig. 10 describes simplified pseudocode of these four-level nested loops. The outermost loop iterates over LUs of data (line 2-18). The second outermost loop iterates until the LSs of the current LU are completely read (line 4-16). The third loop iterates over PUs mapped from the current LU (line 6-15). The innermost loop

TABLE 4
List of Acronyms in the Domain of Flash Storage Platform

Acronym	Original term
ADS	Architecture Design Specification
BML	Block Management Layer
DDS	Detailed Design Specification
DPM	Demand Paging Manager
FTL	Flash Translation Layer
LLD	Low-Level Device driver layer
LS	Logical Sector
LU	Logical Unit
MSR	Multi-Sector Read
PS	Physical Sector
PU	Physical Unit
SAM	Sector Allocation Map
SPU	Sector Per Unit
SRS	Software Requirement Specification
STL	Sector Translation Layer
USP	Unified Storage Platform
XIP	Execution In Place

```
01:curLU = LU0;
02:while(curLU != NULL ) {
03:  readScts = # of sectors to read in curLU
04:  while(readScts > 0 ) {
05:    curPU = LU->firstPU;
06:    while(curPU != NULL ) {
07:      while(...) {
08:        conScts=# of consecutive PSs
09:        offset=the starting offset of these
10:               consecutive PSs is curPU
11:      }
12:      BML_READ(curPU, offset, conScts);
13:      readScts = readScts - conScts;
14:      curPU = curPU->next;
15:    }
16:  }
17:  curLU = curLU->next;
18:}
```

Fig. 10. Pseudoloop structure of the MSR.

identifies consecutive PSs that contain consecutive LSs in the current PU (line 7-11). This loop calculates `conScts` and `offset`, which indicate the number of such consecutive PSs and the starting offset of these PSs, respectively. Once `conScts` and `offset` are obtained, `BML_READ` reads these consecutive PSs as a whole fast (line 12). When the MSR finishes the reading operation, the content of the read buffer should correspond to the original data in the flash memory.

Due to the complex nested loops, a bug of the MSR might be detected only in specific SAM and PU configurations. For example, a buggy MSR may correctly read the data in Fig. 11a where the data are distributed over the PUs in order, but not in Fig. 11b where the data are not distributed in order. Thus, it is necessary to analyze the MSR with various SAM and PU configurations.

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 are six sectors long and distributed over 10 PUs, 2.7×10^8 distinct test scenarios exist. Therefore, a large number of randomized testings hardly provides enough confidence; it is more desirable to exhaustively analyze all valid distribution cases specified by the environment model within a reasonable range (see Section 8.1).[7]

8.1 Environment Model for MSR

The 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 should be created so that a logical relation is maintained between the SAMs and the PUs, as shown in Fig. 11. In this analysis task, we created an environment for the 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 ith LS is written in the kth sector of the jth PU, then the $(i \; mod \; m)$th offset of the jth SAM is valid and indicates the PS number k, where m is the number of sectors per unit.
3. The PS number of the ith 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.

7. The description of the MSR operation and its environment model are taken from [37].

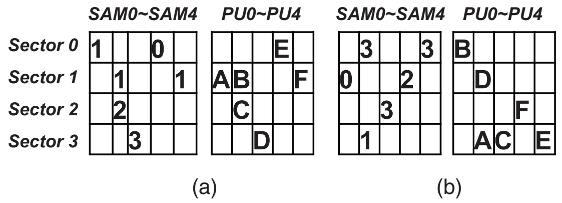

Fig. 11. Two different distributions of data "ABCDEF" to physical sectors. (a) A distribution of "ABCDEF." (b) Another distribution of "ABCDEF."

For example, the last two rules can be specified by the following constraints:[8]

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

8.2 Analysis by Blast

It turned out that Blast could not analyze the MSR accurately due to the following limitations: First, manual modification of the MSR was necessary to replace unsupported operations (e.g., division, mod, etc.) which are treated as uninterpreted functions and often cause false alarms. Second, Blast assumes that an address of an array element can be NULL for the following reason: Blast obtains the address of the ith element of the array a through `offset(`a, i`)`, which is handled as an uninterpreted function. In addition, all pointer operations are assumed safe, and thus the NULL pointer does *not* have a special meaning. Thus, Blast reports a false alarm in the following example:

```
void f() {
  int array[1];
  /* Blast considers that p can be NULL */
  int *p= &array[0];

  assert(p != NULL); /* Blast says "Error" */
}
```

As a consequence, a pointer to a valid physical unit might have the NULL value in the analysis of Blast, which may confuse a termination condition in the traversal of linked list of physical units and logical units whose end is marked as NULL (see line 2 and line 6 in Fig. 10).

Finally, the most serious limitation was that array operations involving an index variable caused false alarms

8. These constraints allow spurious value combinations in SAMs to reduce the complexity of imposing the constraints. However, this weakening does not produce false positives when verifying the requirements.

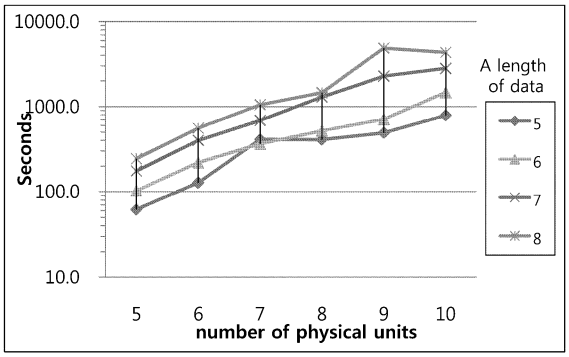

Fig. 12. Time complexity of the MSR analysis.

as shown in the following example, due to the limitation of the underlying decision procedures. Since the theory of arrays is not supported by the underlying decision procedures, BLAST models the memory axioms using lvalue maps and instantiates the array axioms on demand, which may produce imprecise results.

```
void g() {
  int array[1],i,j;
  i=0;array[i]=1;
  j=0;assert(array[j]==1);/* Blast says
  "Error"*/
}
```

Therefore, it was concluded that Blast was inadequate to analyze the MSR since the MSR contained many statements with array and pointer operations.

8.3 Analysis by CBMC

As CBMC performs bounded analysis through loop unwinding, the characteristics of the nested loop structure of the MSR (see Fig. 10) should be analyzed first. We found that the second loop (line 4) and the fourth loop (line 7) were bounded by the number of sectors per unit (SPU) since `readScts` is less than or equal to SPU and decreased by at least one in each iteration of the second loop, and the maximal length of the consecutive physical sectors in one physical unit is trivially less than or equal to SPU. The first loop (line 2) and the third loop (line 6) were bounded by the size of data and the number of physical units available.

We analyzed the MSR with the environment model (see Section 8.1) for data that were five to eight sectors long and distributed over 5-10 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, this exhaustive analysis capability can provide a high confidence of the correctness of MSR, although this verification was performed on the small flash memory only (i.e., 5-10 PUs). The experimental results are illustrated in Fig. 12. For example, it took 1,471 seconds to test all 2.7×10^8 scenarios for data that were six sectors long and distributed over 10 PUs. The SAT formula for this experiment contains 8.0×10^5 Boolean variables and 2.7×10^6 clauses. For each of the experiments, 200-700 megabytes of memory were consumed.

9 Lessons Learned

9.1 Effectiveness of Software Model Checkers for Embedded Software

Through this project, we found that a software model checker can be used as an effective software analysis tool for embedded software since it can directly analyze a target program without a manually created abstract model. Also, the tools show reasonable performance for analyzing components of the target program, although the entire program cannot be analyzed as a whole and the binary libraries used by the target program cannot be analyzed. For embedded software, however, these limitations are less critical and software model checkers offer several advantages for the following reasons.

First, direct testing of embedded software is less effective in an industrial setting. In order to reduce the development period, embedded software and its hardware are usually developed in parallel; embedded software is often developed with hardware specifications, not with physical devices. Thus, direct testing is only possible at the later stage of development, although early detection of design faults can reduce development cost. Software model checkers can be used to detect faults at an early stage of development by analyzing a target component without an execution platform or the entire program.

Second, embedded software operates on bare hardware and dependency on external binary libraries is minimal. Thus, the source code analysis capability of software model checkers can be maximally utilized on embedded software and precise analysis results can be obtained.

Finally, unit analysis by means of software model checking can be more productive than actual testing. Although active research on model-based testing has been undertaken [51], the generation of test cases adequate for various test criteria [54] 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. In addition, even when a set of explicitly generated test cases reaches a high degree of statement/branch coverage, the absence of errors is still not guaranteed; different input values generate different outputs, even in the same execution path (e.g., caused by overflow/underflow 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.

9.2 Limitations of Blast for Complex Embedded Software

Section 6 and Section 7 reported that Blast showed superior performance than CBMC in terms of speed and memory usage on the relatively simple target codes. This is because Blast analyzes an abstract Boolean model through CEGAR with predicate abstraction, while CBMC analyzes a large Boolean formula obtained through multiple unrolling of a target program, SSA transformation, and combinatorial circuit encoding. However, this project revealed several limitations of Blast when applied to complex embedded software (e.g., MSR) as opposed to simple device driver interfaces. Therefore, we suggest that a user should check the amount of complex operations in the target program, as increased complexity could cause Blast to produce imprecise results.

The first limitation is that Blast may produce false alarms due to the limitations of the underlying decision procedures as well as its flow-insensitive may-alias analysis. For example, a target program containing arrays or pointers may cause Blast to generate false alarms (see Section 8.2). Therefore, a manual analysis of counterexamples is required to check the validity of the counterexamples, which is not favorable in terms of productivity. Another problem is that Blast might generate unsound results.[9] For example, Blast handles a scalar value as an unbounded number, as its internal decision procedure does, and therefore it cannot detect errors due to overflow or underflow. In addition, in order to analyze pointer operations efficiently, Blast applies several assumptions on pointers. For example, Blast assumes that two pointers cannot be aliased if their types are different. As a result, the may-alias analysis of Blast is incomplete and Blast may miss a bug. These limitations can cause serious problems in industrial application since the above issues can be easily observed in complex embedded software.

9.3 Industrial Strength of an SAT-Based Bounded Model Checker

CBMC analyzes a target C program with bit-level accuracy since it transforms a target C program into an SAT formula by unwinding loops without abstraction. A subsequent drawback is the requirement of upper bounds for loops, as these bounds are generally difficult to obtain. Also, analysis performance varies in a large degree according to the loop characteristics of a target program (see Section 7.1 and Section 8.3). In practice, however, the user can obtain sound approximate upper bounds by reviewing the target program and its design documents. Even if upper bounds are completely unknown, arbitrary upper bounds given by the user can provide useful debugging information (see Section 7.2.2). Although an entire C program cannot be analyzed by CBMC, we believe that CBMC can be utilized as an effective unit analysis tool for industrial software.

This relatively straightforward analysis technique relies on the high applicability and performance of SAT solvers which possess industrial strength. Several approaches to using a SAT-solver instead of a decision procedure in CEGAR [20], [21], [22] have been studied in efforts to overcome the limitations of decision procedures. However, they are still immature research prototypes and the source codes are not publicly available. Thus, those techniques were not applied in this project.

10 Conclusions

In this project, we successfully applied Blast and CBMC, which utilize CEGAR with predicate abstraction and SAT-based bounded analysis, respectively, to detect bugs in the USP for Samsung's OneNAND 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; they had not been previously detected by Samsung. In addition, we established confidence in the correctness of the complex multisector read function through CBMC. Although current software model checking technologies are not yet scalable to verify an entire C program, it is still beneficial to apply software model checkers to analyze units of a target program, as demonstrated in this project.

In future work, we will continue to apply software model checking techniques to embedded software domains to identify practical issues in which these techniques can be applied in an industrial setting. We hope that we can apply more recent techniques such as SATABS [20], [21] and COGENT [22] to industrial projects in the future.

We also plan to compare software model checking techniques to the concolic testing approach [49], [30], [12] thoroughly; software model checking and concolic testing have several goals in common but have different trade-offs between their verification accuracy, verification time, and applicability. A preliminary result of applying concolic testing to MSR was reported in [38]. This study found that CREST [1], which is an open-source concolic testing tool, can be used as an effective bug-finding tool; however, its analysis speed was much slower than that of CBMC since CREST invoked a constraint solver a few million times to generate test cases for all possible paths. Finally, we plan to carry out a systematic construction of an environment model for software model checking to automate the analysis of embedded software further.

9. The validity of the result produced by Blast in Section 7.1 was not investigated further since CBMC produced the same result.

Acknowledgments

The authors would like to thank Professor Dirk Beyer and Professor Daniel Kroening for the valuable discussion on the features of Blast and CBMC, respectively. In addition, they thank Professor Yunja Choi at Kyungpook National University for the verification of MSR. This research was supported by the Ministry of Knowledge Economy (MKE), Korea, under the Information Technology Research Center (ITRC) support program supervised by the National IT Industry Promotion Agency (NIPA) (NIPA-2009-(C1090-0902-0032)), the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST)/National Research Foundation of Korea (NRF) (Grant 2009-0063252), and Basic Science Research Program through the NRF funded by the Ministry of Education, Science and Technology (2009-0064639). A preliminary version appeared in ASE 2008 [39].

References

[1] CREST—Automatic Test Generation Tool for C, http://code.google.com/p/crest/, 2010.

[2] Flash Memory (from Wikipedia), http://en.wikipedia.org/wiki/Flash_memory, 2010.

[3] Samsung OneNAND Fusion Memory, http://www.samsung.com/global/business/semiconductor/products/fusionmemory/Products_OneNAND.html, 2010.

[4] *Proc. 12th Int'l Conf. Theory and Applications of Satisfiability Testing,* 2009.

[5] T. Ball, B. Cook, V. Levin, and S.K. Rajamani, "Slam and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft," *Proc. Int'l Conf. Integrated Formal Methods,* 2004.

[6] T. Ball, R. Majumdar, T. Millstein, and S.K. Rajamani, "Automatic Predicate Abstraction of c Programs," *Proc. Conf. Programming Language Design and Implementation,* 2001.

[7] T. Ball and S.K. Rajamani, "Automatically Validating Temporal Safety Properties of Interfaces," *Proc. Spin Workshop,* pp. 103-122, 2001.

[8] D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar, "The Software Model Checker Blast: Applications to Software Engineering," *Int'l J. Software Tools for Technology Transfer,* vol. 9, pp. 505-525, 2007.

[9] D. Beyer, D. Zufferey, and R. Majumdar, "CSIsat: Interpolation for LA+EUF," *Proc. Int'l Conf. Computer Aided Verification,* 2008.

[10] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu, "Bounded Model Checking," *Advances in Computers,* vol. 58, pp. 117-148, 2003.

[11] A. Butterfield, L. Freitas, and J. Woodcock, "Mechanising a Formal Model of Flash Memory," *Science of Computer Programming,* vol. 74, no. 4, pp. 219-237, 2009.

[12] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," *Proc. Conf. Operating System Design and Implementation,* 2008.

[13] C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill, and D.R. Engler, "EXE: Automatically Generating Inputs of Death," *Proc. ACM Conf. Computer and Comm. Security,* 2006.

[14] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith, "Modular Verification of Software Components in C," *IEEE Trans. Software Eng.,* vol. 30, no. 6, pp. 388-402, June 2004.

[15] A. Chakrabarti, L. de Alfaro, T. Henzinger, M. Jurdzínski, and F. Mang, "Interface Compatibility Checking for Software Modules," *Proc. Int'l Conf. Computer Aided Verification,* 2001.

[16] A. Cimatti, A. Griggio, and R. Sebastiani, "Efficient Interpolant Generation in Satisfiability Modulo Theories," *Proc. Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems,* 2008.

[17] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded Model Checking Using Satisfiability Solving," *Formal Methods in System Design,* vol. 19, no. 1, pp. 7-34, 2001.

[18] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counter Example-Guided Abstraction Refinement for Symbolic Model Checking," *J. ACM,* vol. 50, no. 5, pp. 752-794, 2003.

[19] E. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," *Proc. Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems,* 2004.

[20] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "Predicate Abstraction of ANSI-C Programs Using SAT," *Formal Methods in System Design,* vol. 25, nos. 2/3, pp. 105-127, 2004.

[21] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: SAT-Based Predicate Abstraction for ANSI-C," *Proc. Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems,* 2005.

[22] B. Cook, D. Kroening, and N. Sharygina, "Cogent: Accurate Theorem Proving for Program Verification," *Proc. Int'l Conf. Computer Aided Verification,* 2005.

[23] P. Cousot, "Abstract Interpretation," *ACM Computing Surveys,* vol. 28, no. 2, pp. 324-328, 1996.

[24] D. Detlefs, G. Nelson, and J.B. Saxe, "Simplify: A Theorem Prover for Program Checking," *J. ACM,* vol. 52, no. 3, pp. 365-473, 2005.

[25] V. D'Silva, D. Kroening, and G. Weissenbacher, "A Survey of Automated Techniques for Formal Software Verification," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 27, no. 7, pp. 1165-1178, July 2008.

[26] E. Clarke, O. Grumberg, and D.A. Peled, *Model Checking.* MIT Press, Jan. 2000.

[27] N. Eén and N. Sörensson, "An Extensible SAT-Solver," *Proc. Int'l Conf. Theory and Applications of Satisfiability Testing,* 2003.

[28] M.A. Ferreira, S.S. Silva, and J.N. Oliveira, "Verifying Intel Flash File System Core Specification," *Proc. Fourth VDM-Overture Workshop,* 2008.

[29] E. Gal and S. Toledo, "Algorithms and Data Structures for Flash Memories," *ACM Computing Surveys,* vol. 37, no. 2, pp. 138-163, 2005.

[30] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," *Proc. Conf. Programming Language Design and Implementation,* 2005.

[31] S. Graf and H. Saidi, "Construction of Abstract State Graphs with PVS," *Proc. Int'l Conf. Computer Aided Verification,* pp. 72-83, 1997.

[32] A. Groce, G. Holzmann, R. Joshi, and R.G. Xu, "Putting Flight Software through the Paces with Testing, Model Checking, and Constraint-Solving," *Proc. Workshop Constraints in Formal Verification,* 2008.

[33] J. Gu, P.W. Purdom, J. Franco, and B.W. Wah, "Algorithms for the Satisfiability (SAT) Problem: A Survey," *Satisfiability Problem: Theory and Applications,* Am. Math. Soc., 1996.

[34] R. Jhala and R. Majumdar, "Software Model Checking," *ACM Computing Surveys,* vol. 41, no. 4, pp. 21-74, 2009.

[35] R. Joshi and G. Holzmann, "A Mini-Challenge: Build a Verifiable Filesystem," *Proc. Int'l Conf. Verified Software: Theories, Tools, Experiments,* 2005.

[36] E. Kang and D. Jackson, "Formal Modeling and Analysis of a Flash File System in Alloy," *Abstract State Machines, B and Z,* Springer, 2008.

[37] M. Kim, Y. Choi, Y. Kim, and H. Kim, "Formal Verification of a Flash Memory Device Driver—An Experience Report," *Proc. Spin Workshop,* 2008.

[38] M. Kim and Y. Kim, "Concolic Testing of the Multi-Sector Read Operation for Flash Memory File System," *Proc. Brazilian Symp. Formal Methods,* 2009.

[39] M. Kim, Y. Kim, and H. Kim, "Unit Testing of Flash Memory Device Driver through a SAT-Based Model Checker," *Proc. IEEE/ACM Int'l Conf. Automated Software Eng.,* Sept. 2008.

[40] E. Kolb, O. Sery, and R. Weiss, "Applicability of the Blast Model Checker: An Industrial Case Study," *Proc. Conf. Perspectives of System Informatics,* 2009.

[41] K. Ku, T.E. Hart, M. Chechik, and D. Lie, "A Buffer Overflow Benchmark for Software Model Checkers," *Proc. Int'l Conf. Automated Software Eng.,* 2007.

[42] W. Mahnke, S.H. Leitner, and M. Damm, *OPC Unified Architecture.* Springer, 2009.

[43] K.L. McMillan, "Interpolation and SAT-Based Model Checking," *Proc. Int'l Conf. Computer Aided Verification,* 2003.

[44] K.L. McMillan, "An Interpolating Theorem Prover," *Theoretical Computer Science,* vol. 345, no. 1, pp. 101-121, 2005.

[45] J.T. Muhlberg and G. Luttgen, "BLASTing Linux Code," *Proc. Int'l Workshop Formal Methods for Industrial Critical Systems,* 2006.

[46] H. Post and W. Küchlin, "Integrated Static Analysis for Linux Device Driver Verification," *Proc. Int'l Conf. Integrated Formal Methods,* 2007.

[47] H. Post, C. Sinz, A. Kaiser, and T. Gorges, "Reducing False Positives by Combining Abstract Interpretation and Bounded Model Checking," *Proc. Int'l Conf. Automated Software Eng.,* 2008.

[48] B. Schlich and S. Kowalewski, "Model Checking c Source Code for Embedded Systems," *Software Tools for Technology Transfer,* vol. 11, no. 3, pp. 187-202, 2009.

[49] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," *Proc. European Software Eng. Conf./Foundations of Software Eng.,* 2005.

[50] M. Sheeran, S. Singh, and G. Stalmarck, "Checking Safety Properties Using Induction and a SAT-Solver," *Proc. Int'l Conf. Formal Methods in Computer Aided Design,* 2000.

[51] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann, 2007.

[52] J. Yang, P. Twohey, D. Engler, and M. Musuvathi, "Using Model Checking to Find Serious File System Errors," *Proc. Symp. Operating System Design and Implementation,* 2004.

[53] L. Zhang and S. Malik, "The Quest for Efficient Boolean Satisfiability Solvers," *Proc. Int'l Conf. Computer Aided Verification,* 2002.

[54] H. Zhu, P.A.V. Hall, and J.H.R. May, "Software Unit Test Coverage and Adequacy," *ACM Computing Surveys,* vol. 29, no. 4, pp. 366-427, 1997.

Moonzoo Kim received the PhD degree in computer and information science from the University of Pennsylvania in 2001. After working as a researcher at Samsung SECUi.COM and POSTECH, he joined the faculty of KAIST in 2006. He currently focuses on developing software verification frameworks for embedded systems through software model checking, runtime verification, and concolic testing techniques. His research interests include the specification and analysis of embedded systems, automated software engineering techniques, and formal methods. He is a member of the IEEE.

Yunho Kim received the BS degree in computer science in 2007 from KAIST, where he is currently working toward the PhD degree in the Computer Science Department. His research interests include embedded software, software model checking, and test-case generation.

Hotae Kim received the BS degree in computer science from KAIST and the MS degree in computer science and engineering from POSTECH. He is currently a researcher at Samsung Electronics and his recent project involvement has focused on the LiMo software platform and embedded software for flash memory devices such as file systems and databases. His research interests include software development and evolution applying various software engineering practices such as formal design and verification, model checking, and software product line.