Model-based Kernel Testing for Concurrency Bugs through Counter Example Replay

Moonzoo Kim, Shin Hong, Changki Hong[1] ,[2]

CS Department KAIST
Daejeon, South Korea

Taeho Kim[3]

ETRI
Daejeon, South Korea

Abstract

Despite the growing need for customized operating system kernels for embedded devices, kernel development continues to suffer from high development and testing costs for several reasons, including the high complexity of the kernel code, the infeasibility of unit testing, exponential numbers of concurrent behaviors, and a lack of proper tool support. To alleviate these difficulties, this study proposes the MOdel-based KERnel Testing (MOKERT) framework, which supports detection of concurrency bugs in the kernel by combining both model checking techniques and testing methods. The MOKERT framework was applied to the file systems of the Linux 2.6 kernel and found a data race bug in the proc file system.

Keywords: Model checking, testing, counter example analysis, and model extraction

1 Introduction

Software testing often requires more than 50% of the total development time and available resources. Among the many different software products, the operating system kernel (henceforth, simply termed the kernel) is one of the most difficult software to develop and analyze. Reasons for this include the following:

- *Complexity of code*
  In order to manage various HW and SW components, the kernel contains a large

* Preliminary version appeared in Korea Conference on Software Engineering 2009

[1] This work was supported by the IT R&D program of MKE/IITA (2006-S-038-01, Development of Device-Adaptive Embedded Operating System for Mobile Convergence Computing), and Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST)/Korea Science and Engineering Foundation(KOSEF), grant number R11-2008-007-03002-0.

[2] Email: `moonzoo@cs.kaist.ac.kr, {hongshin,kornin83}@kaist.ac.kr`

[3] Email: `taehokim@etri.re.kr`

number of service routines. In addition, in order to maximize performance, it often uses optimized but complex algorithms.

- *An exponential number of concurrent behaviors*
  The kernel is a multi-threaded program in which it is very difficult to detect *concurrency bugs*, as they occur only when specific scheduling sequences are enforced. However, there exist exponentially many scheduling scenarios in terms of the number of concurrent threads. Furthermore, for most kernels, a user does not have a fine-grain control over the scheduler, which increases the difficulty involved with analyzing the concurrent behaviors of a kernel.
- *Difficulty of unit testing*
  Most kernels utilize a monolithic architecture for the sake of performance. Thus, it is challenging to decompose a kernel into components or units in order to test each independently. For example, to test the proc file system, an environment that contains a virtual file system layer, memory management, and a scheduler is required, as these components are closely involved in the operation of the proc file system. In such a situation, huge effort is inevitable for unit testing, as it is necessary to configure the environment in detail; this environment setup causes more overhead than that involved in the unit testing itself.
- *Lack of proper tool support*
  There are few tools that can support kernel testing, as the side effects caused by the testing tool/environment at the kernel space can halt the kernel or cause unexpected behaviors. In addition, testing tools often depend on libraries that cannot be used in the kernel environment. For these reasons, kernel developers continue to use `printk()` or a kernel log as a main debugging aid.

Considering the high complexity of the kernel, it is essential to analyze the kernel components one-by-one. This, however, is hard to achieve in practice due to the above reasons. Consequently, traditional testing methods do not support kernel development satisfactorily, and the kernel remains an area suitable for only gifted guru developers. However, as embedded systems become more prevalent, the need to create/customize special-purpose operating systems (e.g., operating systems for mobile phones or sensor networks) continues to grow. Therefore, it is important to have a framework that can analyze the kernel unit by unit.

This study proposes a model-based testing framework for *concurrency bugs* in the kernel by using model extraction [3] and model checking techniques [4]. Considering the difficulties of testing the kernel as mentioned earlier, model checking presents a viable alternative approach for analyzing kernels, since this technique can analyze each component of the complex kernel independently by modeling the component and the related portion of the environment in an *abstract* manner. In addition, model checking can explore numerous scenarios exhaustively, and generate concrete counter examples, which can be a useful aid to debugging. However, a pure model checking approach alone is problematic in that a developer cannot know whether a counter example detected through model checking is an actual bug or a false alarm due to the gap between the abstract model and target code.

A salient advantage of the MOdel-based KERnel Testing (MOKERT) framework proposed in this paper is the capability of *replaying a counter example* on the actual kernel code. Through this capability, MOKERT provides benefits of both model checking and testing to kernel developers. This feature can assist kernel developers with bug identification, model refinement, and validation of a bug patch (see Section 3 for more details). Therefore, the MOKERT framework can alleviate the difficulty of analyzing the kernel through pure testing, although it requires human effort to create an abstract model and its environment. The effectiveness of the MOKERT framework was demonstrated by applying it to the file systems of the Linux 2.6 kernel and a hidden data race bug in the proc file system was detected .

2 The Spin Model Checker and Modex

This section provides an overview of the Spin model checker [4] and Modex [3], which are utilized as the underlying components in the MOKERT framework.

2.1 The Spin Model Checker

Promela (a modeling language of the Spin model checker) is similar to the C programming language in several aspects. First, Promela provides control statements such as `if`, `goto`, and `do`. In addition, Promela models complex data structures using `typedef` and arrays, although pointer variables are not directly supported. Also, Promela supports `inline` functions which can simulate C functions.

The Spin model checeker exhaustively explore the state space explicitly generated from a given Promela model and generates a counter example if it finds a violation of a requirement property. A counter example contains the following information:

- Step number n
- Process ID of the current process
- `proctype` name of the current process
- Line number l of the Promela model being executed
- Statement of the Promela model at line l

For example, the 45th step of the counter example in Figure 1 indicates that process 1 executes the statement at line 125 of the `proc_readdir()`, which tests whether `proc_subdir_lock==1` or not. Then, process 1 is preempted and process 2 executes the statement at line 156 of `remove_proc_entry()`, and so on.

```
45: proc 1 (proc_readdir)      line 125 [((proc_subdir_lock==1))]
46: proc 2 (remove_proc_entry) line 156 [((proc_subdir_lock==0))]
```

Fig. 1. Counter example generated from Spin

2.2 Modex

Modex automatically translates a C program into a corresponding Promela model. Modex translates the control structure of a target C function, such as `if` and `while`, into the corresponding Promela control structure. As a result, the Promela model generated by Modex has the same control structure as the target C program. Other

C statements are inserted as embedded C code into the Promela model starting with a keyword `c_expr{...}` for Boolean expressions and `c_code{...}` for assignments and function calls [3]. These embedded C codes are blindly copied from the text of the Promela model into the code of the verifier that Spin generates.

In addition, a user can control the translation by specifying a *translation script*. A translation script for Modex contains a set of translation tables, each of which specifies translation rules for a function in the target C program. A translation rule specifies that the lines of the target C code starting with a given textual pattern should be replaced by the given Promela statements (see Figure 2(b)). In addition, the translation script contains a user-specified environment of a target model and the requirement properties to verify. Figure 2(a) shows an excerpt of `proc_readdir()` in the proc file system of the Linux 2.6.21 and the Promela code generated by Modex using the translation table of Figure 2(b). Note that Modex indicates which Promela statements are translated from which C statements through comments (i.e., `/* line` *l* `*/`) showing the corresponding line numbers of the original C program.

C code	Promela code
68: do {	123: do
69: spin_unlock(&proc_subdir_lock);	124: ::
70: if (filldir(dirent, de->name, de->namelen,	125: inlined_spin_unlock(proc_subdir_lock);/* line 69 */
71: filp->f_pos, de->low_ino, de->mode >> 12) < 0)	126: if
72: goto out;	127: :: false; /* line 70 */
73: spin_lock(&proc_subdir_lock);	128: goto out;
74: filp->f_pos++;	129: :: else; /* line 72 */
75: next = de->next;	130: fi;
76: de=next;	131: inlined_spin_lock(proc_subdir_lock); /* line 73 */
77: } while (de);	132: i = proc_subdir[i].next; /* line 75 */
	133: if
	134: :: i != NULL; /* line 77 */
	135: :: else; break; /* line 77 */
	136: fi;
	137: od;

(a) `proc_readdir()` and the corresponding Promela model

Patterns in C code	Corresponding Promela code
spin_unlock(&proc_subdir_lock);	inlined_spin_unlock(proc_subdir_lock)
(filldir(dirent…	false
spin_lock(&proc_subdir_lock);	inlined_spin_lock(proc_subdir_lock)
next = de->next;	i = proc_subdir[i].next
de	i != NULL
de=…	comment
filp->f_pos++;	comment

(b) A translation table for Modex

Fig. 2. Excerpt of `proc_readdir()`and the corresponding Promela model created by Modex

3 The Model-based Kernel Testing Framework

This section describes the MOKERT framework, particularly its architecture and instrumentation scheme to replay a counter example.

3.1 Overview

The MOdel-based KERnel Testing (MOKERT) framework applies the analysis result (i.e., a counter example) from model checking to the original target program so as to alleviate the difficulties of debugging concurrency bugs in the kernel code. Most model checking frameworks do not suitably consider this issue, although it is crucial for the success of model checking techniques in the software industry.

A traditional model-based testing approach is illustrated in Figure 3(a), in which a formal model is written manually and a counter example obtained from model checking is checked with regard to the target program by a human user. This process requires significant manual effort. The MOKERT framework, as illustrated in Figure 3(b), uses Modex to extract a formal model from a target C program in a (semi) automatic manner with a translation script $\mathcal{T}$; $\mathcal{T}$ maps the statements/lines of the target C program to the statements/lines of the corresponding Promela model. [4] Following this, $\mathcal{T}$ is used in a *reverse* manner ($\mathcal{T}^{-1}$) to map the lines of the Promela model specified in a counter example to the corresponding lines of the original C target program. Therefore, it is possible to understand which sequences of lines of the target program should be executed according to the counter example and instrument the target program to follow the counter example.

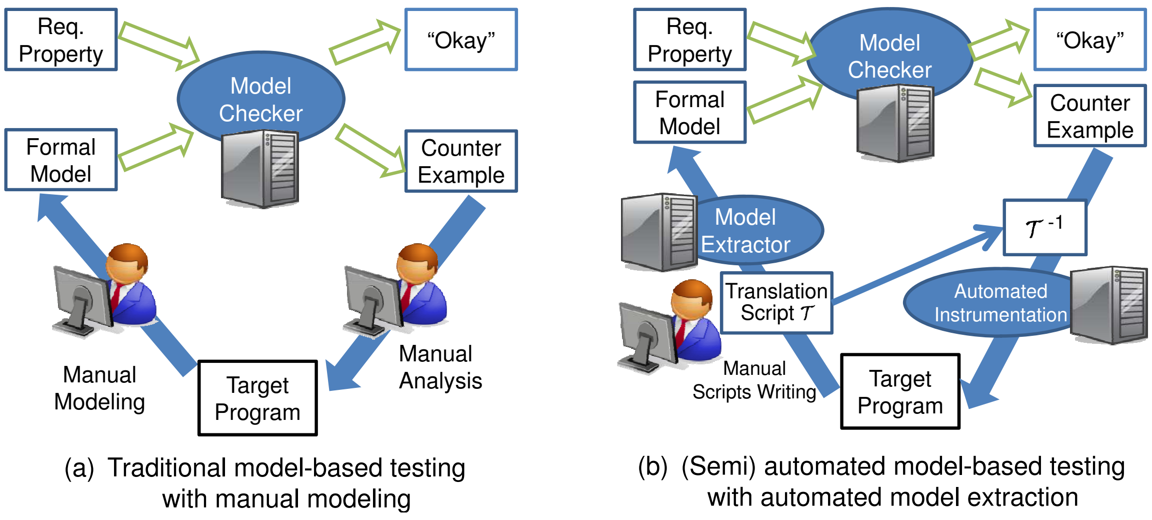


Fig. 3. Two different model-based testing approaches

The MOKERT framework has the following advantages:

- *Identification of a cause of a bug*
  MOKERT allows a user to replay one counter example repeatedly. Thus, a user can conveniently identify the genuine cause of the counter example in the kernel code, not merely in the corresponding abstract formal model. This can be a very useful aid for detecting concurrency bugs, as thread scheduling should be controlled manually otherwise.
- *Model refinement*

[4] We can consider a translation script $\mathcal{T}$ contains a line number mapping from the original C code $\mathcal{C}$ to the Promela model $\mathcal{P}$ translated from $\mathcal{C}$, since Modex generates this line number mapping information when it generates $\mathcal{P}$ from $\mathcal{C}$ based on $\mathcal{T}$.

An initial formal model is highly abstract and the model is refined iteratively by adding more details. When a counter example is generated from model checking, it is possible to check whether the counter example is real or a false alarm attributable to abstraction by running the counter example on the actual code (see Section 3.4). If the counter example is a false alarm, it is necessary to refine/fix the formal model.

- *Validation of a bug patch*
  Kernel bugs are often reported through manual code inspections, discussions, or artificial scenarios without actual testing, due to the difficulties involved with kernel testing (see Section 1). Thus, for example, the Linux Changelog contains dozens of bug patches that are not necessary or that do not fix a bug correctly. MOKERT can help users validate a bug patch by modifying the model according to the bug patch and model checking it to check whether the bug patch actually removes the bug. If the model checker generates a counter example, MOKERT can replay the counter example to check whether it is due to the incorrect bug patch or a false alarm.

3.2 Architecture of the MOKERT Framework

The overall architecture of the MOKERT framework is illustrated in Figure 4. The static phase of the MOKERT framework consists of the following tasks:

(1) Given a target C program and a translation script, Modex translates the target program into a Promela model that contains the target processes, an environment model, and requirement properties. At the same time, *the mapping information* from the line numbers of the Promela model to the line numbers of the target program is obtained from the intermediate results that Modex generates.

(2) Spin verifies whether the generated Promela model satisfies the requirement properties with the environment or not. [5] If yes, Spin delivers a verification result to a user. Otherwise, Spin generates a counter example.

(3) The portion of the target C code to analyze is automatically instrumented based on the mapping information and the counter example. The instrumented target code contains probes at lines corresponding to the context switch points in the counter example, a *controller thread*, and a *monitor thread*. In addition, a *test driver* for the target program should be given by a user. The driver contains a testing environment that is equivalent to the environment model used in model checking. This instrumented portion of the target code and the test driver are compiled with the remaining target code to build the executable program.

At the run-time phase, the target threads execute with a controller thread and a monitor thread. The controller thread controls scheduling of the target threads via the inserted probes. The monitor thread monitors the target threads and the

[5] Requirement properties are given as linear temporal logic formulae or assert statements. Currently, however, MOKERT does not check whether a target program correctly replays a lasso-shaped counter example of a liveness property or not.

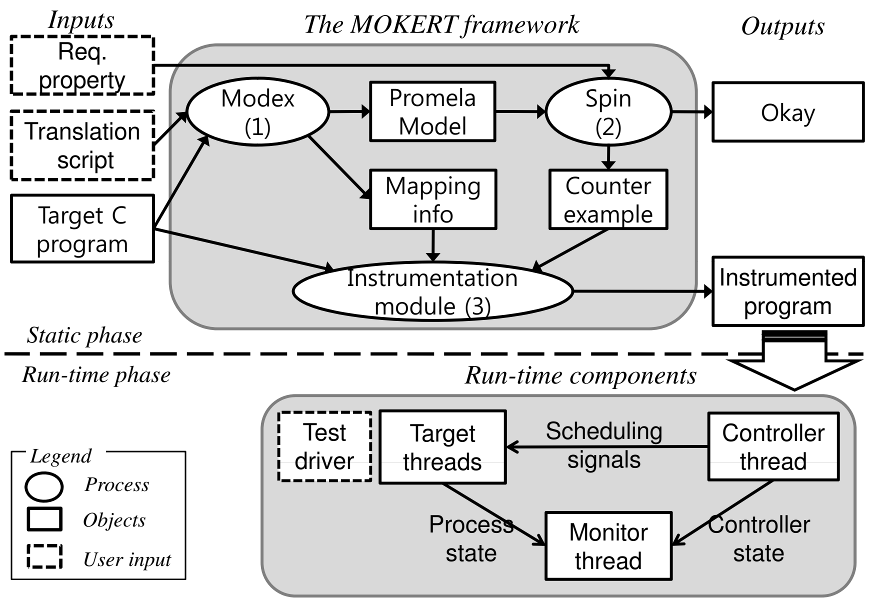


Fig. 4. Architecture of the MOKERT framework

controller thread, and checks whether the counter example is replayed correctly or not.

Note that if a counter example is generated, MOKERT requires a user to build a test driver. This task is, however, simpler than the task of building a test driver for general test cases, for the following reasons. First, an abstract environment model used for model checking can help a user set up the testbed [7]. Furthermore, a test driver required by MOKERT covers only a specific scenario represented by the single counter example. This reduces a significant amount of effort required to build a general testbed and to run numerous test cases.

3.3 Instrumentation of the Target Programs

In order to enforce the target program to follow the scheduling sequence of the counter example, MOKERT inserts probes at the locations corresponding to the *context switch points* of the counter example based on the mapping information. For example, Figure 1 shows one context switch point occurring immediately after the execution of line 125 of the Promela model. Line 125 of the Promela model corresponds to line 69 of the target C code as indicated in Figure 2(a). Consequently, a probe is inserted immediately after line 69 of `proc_readdir()`.

As a counter example represents concurrent execution in an interleaved manner, the probes inserted in the target program enforce the target threads to execute in a serialized way (i.e., one target thread at a time), strictly following the *global schedule table* generated from the counter example. Figure 5 describes the basic structure of a probe. To minimize unnecessary side effects, a probe is activated only when the `test_start` flag is set as true by the test driver (see line 1). Since one function can be invoked by multiple threads, a probe should know which thread is executing the probe and perform a context switch accordingly (see line 4 to line 20).

A probe inserted at the statement located at `CUR_LINE` (e.g., line 69 of `proc_readdir()` in Figure 2(a)) of the target program checks whether a context switch should occur now or not (line 5 to line 19), since `CUR_LINE` may be executed multiple times through a loop but a context switch should occur at n_1th, n_2th,... and n_kth executions only (see line 6 to line 9). For example, line 69 of `proc_readdir()` may be executed 7 times in the scenario of the counter example. However, a context switch may occur at the 3rd (n_1) and 5th (n_2) executions only. If a context switch should occur, the probe sets its `runnable[m_pid]` as false (line 11), notifies the controller thread (line 12), yields current execution (line 14), and waits until the controller thread allows the thread to run by setting `runnable[m_pid]` as true (line 13). Note that the controller thread keeps only one `runnable[i]` as true at any given instant.

```
01:if (test_start == true) {
02:  m_pid = get_model_pid(current_pid)
03:  switch(m_pid) {
04:    case m_pid1 :
05:      switch(executed[m_pid][CUR_LINE]) {
06:        case N_1:  /* Constant N_1, N_2,..., N_k are calculated from
07:                      the counter example at the static phase */
08:        ...
09:        case N_k:  /* if the statement at CUR_LINE is executed
10:                      by the process m_pid1 N_k th time */
11:          runnable[m_pid] = false;
12:          notify_controller();
13:          while (runnable[m_pid]==false) {
14:            sys_sched_yield()
15:          }
16:        default:
17:          executed[m_pid][CUR_LINE]++;
18:          break;
19:      }
20:      break;
21:    case m_pid2 :
22:      ...
```

Fig. 5. Structure of a probe inserted at `CUR_LINE`

3.4 Run-time Phase

At run-time, the monitor thread observes the target threads and the controller thread, and checks whether the counter example is being replayed correctly or not. When the execution of the target program reaches the last step of the counter example, the monitor thread notifies a user that the counter example is completely replayed. At the same time, as a test result, the test driver shows if the requirement property is violated or not. There are three possible results from replaying a counter example.

(i) The last step of the counter example is reached and the requirement property is violated.

(ii) The last step is reached but the requirement property is *not* violated.

(iii) The last step is *never* reached.

The first case is what we expect from replaying the counter example. The second and the third cases indicate that replaying the counter example fails. If the last step of the counter example is not reached within a given amount of time, the monitor notifies a user of the replay failure with log information such as the status of each target thread and the function call stack. Replay failures can occur for the following reasons:

(i) *Imprecise modeling*
A model of the target threads is not refined enough and, as a result, the actual target threads cannot progress following the counter example. The global configuration of the kernel might unexpectedly change according to numerous causes, which might not be included in the model due to abstraction. For example, the model may assume that the interrupt function is disabled, but the actual kernel execution might enable the interrupt function unexpectedly, which might block the interleaving sequence enforced by the controller thread.

(ii) *Incorrect test driver*
A test driver may be built incorrectly and as a result the test driver does not fully reflect the verification environment used in model checking.

For example, Section 5 describes a case study where the counter example could not be replayed due to imprecise modeling. If a replay failure is detected, the user should refine the abstract model and/or fix the test driver.

4 Case Study 1: Data Race between `proc_readdir()`and `remove_proc_entry()`in Proc File System

4.1 Bug Description

The proc file system is a virtual file system in UNIX-like operating systems. This file system allows a user to access process information and communicate with the kernel modules through the file interface. Linux Changelog 2.6.22 reported that `proc_readdir()` and `remove_proc_entry()` had a data race bug in the Linux 2.6.21. This data race scenario is depicted in Figure 6. Before `remove_proc_entry()` removes `de` (a directory entry representing a file in proc file system) at line 24b, it checks the read count of `de` at line 23b to guarantee that no one is using the `de` now. However, `proc_readdir()` does not increase the read count of `de` at line 68a before reading `de`. Thus, `remove_proc_entry()` removes `de` without knowing that `proc_readdir()` is trying to read `de`. Consequently, `proc_readdir()` tries to read `de` at line 70a, which was removed by `remove_proc_entry()` at line 24b. This may cause a segmentation fault due to an invalid memory access.

`proc_readdir()`	`remove_proc_entry()`
68a:do { /* missing atomic_inc(&**de**->count) */	
69a: spin_unlock(&proc_subdir_lock);	
	10b:spin_lock(&proc_subdir_lock);
	...
	23b: if (!atomic_read(&**de**->count))
	24b: free_proc_entry(**de**);
	...
	31b:spin_unlock(&proc_subdir_lock);
70a: if (filldir(dirent, **de**->name, **de**->namelen,	
71a: filp->f_pos,de->low_ino,**de**->mode>>12)<0)	

Fig. 6. Data race scenario between `proc_readdir()` and `remove_proc_entry()`

4.2 Replaying the Bug

Through the MOKERT framework, `proc_readdir()` (67 lines long), `remove_proc_entry()` (35 lines long), and related sub-functions such as locking functions were modeled in Promela through Modex. Corresponding Promela processes for `proc_readdir()` and `remove_proc_entry()` were 68 lines and 36 lines long, respectively. In addition, a verification environment (76 lines long) having four proc directory entries, namely "Jan", "Feb", "Mar", and "Apr", as depicted in Figure 7, was modeled in the translation script (128 lines long). The environment created two processes for `proc_readdir()` and `remove_proc_entry()`, and made `remove_proc_entry()` remove the "Feb" directory entry while `proc_readdir()` tried to read the same directory entry.

The requirement property asserted that the directory entry being read at line 70a of `proc_readdir()` must be available (i.e., not removed previously). To check this property, an auxiliary flag `removed` was added to each directory entry, which was set as true when the directory entry was removed. Note that it took two days for two graduate students to understand the bug description and corresponding portion of the proc file system C code, one day for specifying a translation script, and another day for building the test driver and replaying the counter example.

Spin generated a counter example (59 steps long) in our model, as expected. When the counter example was replayed, the removed `de` was displayed as if it had not been removed and the directory entries linked from `de` were not displayed. (note that directory entries form a single linked list in the proc file system). In other words, `proc_readdir()` displayed "Jan" and "Feb" only, not "Mar" nor "Apr" when `remove_proc_entry()` removed "Feb" in the data race scenario in Figure 6. This data race did not cause a segmentation fault, since `free_proc_entry(de)`(line 24b) did not fill the memory area occupied by `de` with null values, but kept the old values of the memory area. [6] This data race bug was fixed in Linux 2.6.22 by adding `atomic_inc(&de->count)` at line 68a of `proc_readdir()`. The Promela

[6] `free_proc_entry(de)` only removed the outgoing link of `de` and updated the incoming link to `de` correctly. Thus, `remove_proc_entry()` updated the outgoing link of "Jan" to point to "Mar" correctly. However, `proc_readdir()` used an old link, not the updated link in the scenario of Figure 6 and printed "Jan" and "Feb" only.

model for `proc_readdir()` was modified according to the bug patch and ceased to violate the property.

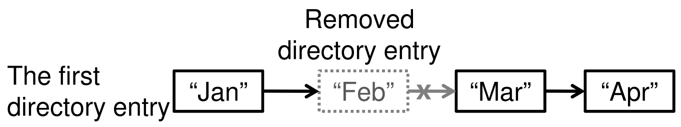


Fig. 7. Example of the directory entry structure

5 Case Study 2: Model Refinement of `ext2_readdir()` and `ext2_rmdir()` in the Ext2 File System

5.1 Bug Description

During the modeling and analysis of the ext2 file system in Linux 2.6.25, a data race between `ext2_readdir()`(78 lines long) and `ext2_rmdir()`(15 lines long) was detected through Spin. This data race scenario is depicted in Figure 8. Since `de` is not protected by any lock at line 67a of `ext2_readdir()`, `ext2_rmdir()` can remove `de` at line 98b. Then, `ext2_readdir()` tries to read `de` at line 75a, which was already removed by `ext2_rmdir()`, thus possibly causing a null pointer dereference.

`ext2_readdir()`	`ext2_rmdir()`

67a: if(de->inode) {
…
74a: offset = (char *) de – kaddr;	

	93b: struct inode * inode = dentry->d_inode;
	…
	97b: if (ext2_empty_dir(inode)) {
	98b: err = ext2_unlink(dir, dentry);

75a: over = filldir(dirent, de->name, de->name_len, 76a: (n<PAGE_CACHE_SHIFT) | offset, 77a: le32_to_cpu(de->inode), d_type);	

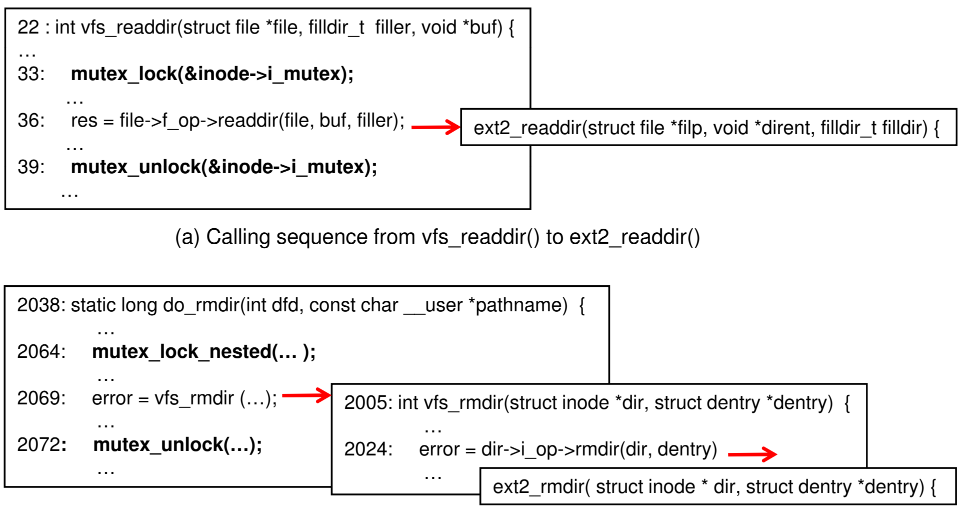
Fig. 8. Data race scenario between `ext2_readdir()` and `ext2_rmdir()`

5.2 Replaying the Bug

The corresponding Promela model had two processes for `ext2_readdir()` and `ext2_rmdir()`. The processes were 65 lines long and 15 lines long, respectively. The environment (241 lines long) contained two directories, namely, "parent" and "child"; the "parent" directory had the "child" as its sub-directory and "child" directory was empty. The environment made a scenario that `ext2_rmdir()` removed "child" while `ext2_readdir()` read "parent". Each directory entry had an auxiliary flag `removed`, which was set as true when the directory entry was removed. The requirement property, similar to that of Section 4, asserted that the flag of the directory entry being read at line 75a of `ext2_readdir()` should be false.

Spin found a counter example (77 steps long) from the model. To replay the counter example, the wrapping functions `vfs_readdir()` and `do_rmdir()` should be invoked instead, since `ext2_readdir()` and `ext2_rmdir()` require a sophisticated environmental configuration and could not be invoked directly. The calling sequences from `vfs_readdir()` to `ext2_readdir()` and `do_rmdir()` to `ext2_rmdir()` are illustrated in Figure 9. However, when the counter example was replayed on the kernel, `de` was not removed.

The monitor thread indicated that the status of the thread executing `vfs_readdir()` was "running". However, the detailed log in the monitor showed that the thread was spinning inside the inserted probe at line 74a of `ext2_readdir()`, i.e., executing `sys_sched_yield()` in an infinite while loop (see line 13 to line 15 of Figure 5). Also, the monitor thread indicated that the thread executing `do_rmdir()` continued to sleep even without invoking `vfs_rmdir()`(thus, `ext2_rmdir()` neither). These observations led to the conclusion that the thread scheduling could not be enforced as directed by the counter example.



```
22 : int vfs_readdir(struct file *file, filldir_t  filler, void *buf) {
...
33:    mutex_lock(&inode->i_mutex);
       ...
36:    res = file->f_op->readdir(file, buf, filler);  -->  ext2_readdir(struct file *filp, void *dirent, filldir_t filldir) {
       ...
39:    mutex_unlock(&inode->i_mutex);
       ...
```

(a) Calling sequence from vfs_readdir() to ext2_readdir()

```
2038: static long do_rmdir(int dfd, const char __user *pathname)  {
         ...
2064:    mutex_lock_nested(... );
         ...
2069:    error = vfs_rmdir (...);  -->  2005: int vfs_rmdir(struct inode *dir, struct dentry *dentry)  {
         ...                                  ...
2072:    mutex_unlock(...);              2024:   error = dir->i_op->rmdir(dir, dentry)  -->  ext2_rmdir( struct inode * dir, struct dentry *dentry) {
         ...                                  ...
```

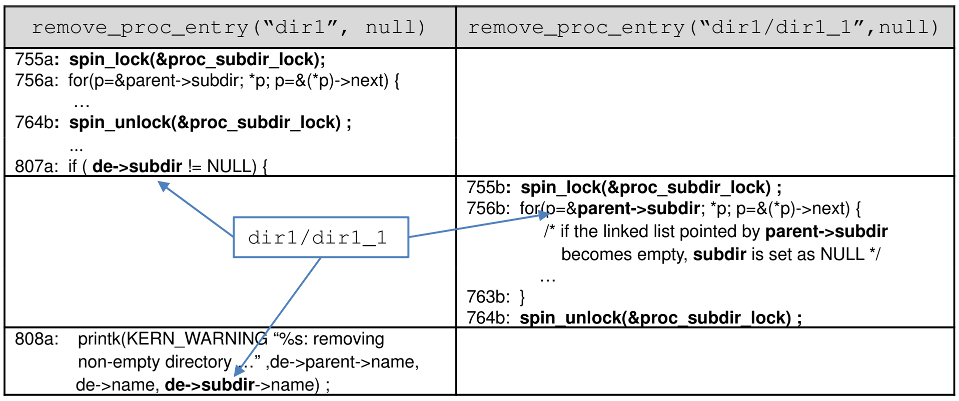
(b) Calling sequence from do_rmdir() to ext2_readdir()

Fig. 9. Calling sequence to `ext2_readdir()` and `ext2_rmdir()`from `vfs_readdir()`and `do_rmdir()`

By analyzing `vfs_readdir()` and `do_rmdir()` where the execution stalled, it was found that `ext2_readdir()` was protected by the mutex in `vfs_readdir()`, as indicated at lines 33 and 39 of `vfs_readdir()`, and `ext2_rmdir()` was protected by the same mutex at lines 2064 and 2072 in `do_rmdir()`. Considering that `ext2_readdir()` and `ext2_rmdir()` could be invoked only from `vfs_readdir()` and `do_rmdir()`, the data race detected from our model for `ext2_readdir()` and `ext2_rmdir()` was a false alarm. After we refined the model to include `vfs_readdir()` and `do_rmdir()`, Spin did not generate a counter example.

6 Case Study 3: Data Race between Multiple `remove_proc_entry()`'s in Linux Kernel 2.6.28.2

MOKERT was applied to verify the proc file system in Linux 2.6.28.2 (the most latest release at the time of writing). From the generated Promela model, Spin detected a data race bug which had not been discovered before. The corresponding data race scenario is depicted in Figure 10. The data race occured when one thread removed `dir1` and another thread removed `dir1/dir1_1` at the same time. Although the code of removing a directory was protected by the spin lock (see lines 755b to 764b), the thread removing `dir1` might refer to its child directory `dir1/dir1_1` (see line 808a) which had been already removed by another thread, thus possibly causing a null pointer dereference.



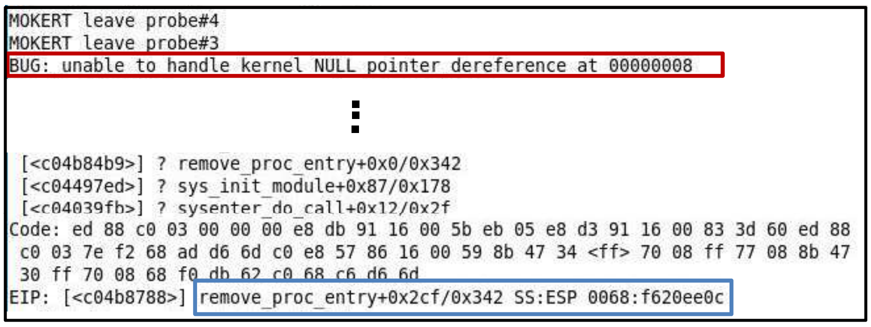
```
remove_proc_entry("dir1", null)
755a: spin_lock(&proc_subdir_lock);
756a: for(p=&parent->subdir; *p; p=&(*p)->next) {
        ...
764b: spin_unlock(&proc_subdir_lock) ;
        ...
807a: if ( de->subdir != NULL) {

808a:   printk(KERN_WARNING "%s: removing
        non-empty directory ..." ,de->parent->name,
        de->name, de->subdir->name) ;
```

```
remove_proc_entry("dir1/dir1_1",null)
755b: spin_lock(&proc_subdir_lock) ;
756b: for(p=&parent->subdir; *p; p=&(*p)->next) {
          /* if the linked list pointed by parent->subdir
             becomes empty, subdir is set as NULL */
          ...
763b: }
764b: spin_unlock(&proc_subdir_lock) ;
```

Fig. 10. Data race scenario between two `remove_proc_entry()`'s

The translation script and the environment model for the case study 1 (see Section 4) were reused to generate a Promela model (81 lines) from the proc file system code. The verification environment (73 lines long) created two proc directory entries, namely `dir1` and `dir1/dir1_1`, and two processes for `remove_proc_entry()`; one `remove_proc_entry()` removed `dir1` while the other `proc_readdir()` removed `dir1/dir1_1` at the same time.

The requirement specification to detect NULL pointer dereferences was described as follows. Since Promela does not provide pointer variables, pointers were modeled by index variables for arrays. NULL value was defined as 255 in this verification task and the sizes of the arrays in the model were less than 255. Therefore, if a NULL pointer was dereferenced in the target C program, the corresponding Promela model would access the corresponding array through an index variable whose value was beyond the bound of the array. Spin raised an array index violation error and generated a counter example. When the counter example was replayed, the kernel panic message reporting the NULL pointer dereference in `remove_proc_entry()` appeared as shown in Figure 11.



```
MOKERT leave probe#4
MOKERT leave probe#3
BUG: unable to handle kernel NULL pointer dereference at 00000008

                              ⋮

 [<c04b84b9>] ? remove_proc_entry+0x0/0x342
 [<c04497ed>] ? sys_init_module+0x87/0x178
 [<c04039fb>] ? sysenter_do_call+0x12/0x2f
Code: ed 88 c0 03 00 00 00 e8 db 91 16 00 5b eb 05 e8 d3 91 16 00 83 3d 60 ed 88
 c0 03 7e f2 68 ad d6 6d c0 e8 57 86 16 00 59 8b 47 34 <ff> 70 08 ff 77 08 8b 47
 30 ff 70 08 68 f0 db 62 c0 68 c6 d6 6d
EIP: [<c04b8788>] remove_proc_entry+0x2cf/0x342 SS:ESP 0068:f620ee0c
```

Fig. 11. Snapshot of a kernel panic message due to the data race between two `remove_proc_entry()`'s

Note that it took only one day for one graduate student to perform this whole debugging task, since the previous experimental setting including the translation script and the test driver could be reused. Although construction of a target model, its environment, and the test driver cost time and manual effort, those initial setting can be reused for other debugging tasks, thus lowering the average cost.

7 Related Works

Research on testing concurrent programs utilizes both dynamic and static analyses. Eraser [12] proposes an algorithm to detect data race candidates at run-time by observing the operations of locks that protect shared resources. RacerX [2] aims for the same goal, but it detects such data race candidates by analyzing the target program statically. These approaches try to discover which locks protect shared data. In general, this knowledge is, however, hard to obtain due to the high complexity of the target program. Engler et al. [1] detect the statements in the target program that match user-specified bug patterns by scanning the target program code. For example, pairing operations such as lock/unlock can be checked by this approach.

In addition, software model checking is utilized for debugging concurrent programs. Yang et al. [6] modified the kernel to run on top of a model checker and found several errors in the file system. However, this approach is not readily applied to general target systems, since it requires intrusive and non-portable modification of the kernel to a large degree. VeriSoft [10] and CHESS [8] provide a more general framework based on stateless search techniques, which can test a large number of scenarios systematically. However, these approaches require that the target component as well as its environment be completely executable, and thus suffer from similar problems to those of the kernel testing. Furthermore, these frameworks depend on the underlying execution platform (e.g., WIN32API) and cannot directly test the kernel operating on bare hardware.

Other software model checkers extract a formal model from a given target C program through predicate abstraction or SAT encoding, and then apply model checking. These software model checkers were developed for sequential programs and have been extended for concurrent programs. These extended software model checkers apply new abstraction techniques or explore only partial concurrent executions to reduce state space. In [14], instead of considering interleaved executions among all concrete thread executions, the model checker considers the interleaving of one concrete thread execution and the abstracted executions of the other threads. In [13], an extended version of Blast abstracts not only data but also control structure of each thread. TCBMC [5] checks a part of the target program's concurrent executions with a context-switching bound. KISS [11] transforms a given concurrent program into a sequential program that simulates the concurrent executions with only one context-switch. The goal of these software model checking techniques is to alleviate the state space explosion problem caused by numerous interleaved executions among the concurrent threads. But these software model checkers are still not applicable to automatically analyze the kernel due to the complexity of the kernel. In contrast, MOKERT does not aim at a fully automatic analysis of the target program, but utilizes human expertise to build an abstract model and environment for a real-world application (see [9]). We believe that MOKERT can provide kernel developers a practical platform for the kernel analysis.

8 Conclusion and Future Work

This paper proposed the MOKERT framework to detect concurrency bugs in the kernel. The framework applies the analysis result of model checking (i.e., a counter example) to the actual kernel code. Thus, the framework increases the level of user confidence, as it confirms the exhaustive model checking result on the real kernel code. We have demonstrated the effectiveness of the framework through three case studies on the the file systems of the Linux 2.6 kernel and found a hidden data race bug in the proc file system.

Through these case studies, it was observed that the model extraction technique should be automated further, since current model extraction techniques, including Modex, still depend on human expertise much. We also plan to continue analyzing the file systems of the Linux kernel through the MOKERT framework.

Acknowledgments

We would like to thank Junhee Lee and Yunho Kim at KAIST for the valuable discussions on the kernel debugging techniques and the data structure of the Linux 2.6 kernel.

References

[1] D.Engler, B.Chelf, A.Chou, and S.Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Operating System Design and Implementation (OSDI)*, 2000.

[2] D.Engler and K.Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Symposium on Operating Systems Principles*, 2003.

[3] G.J.Holzmann and R.Joshi. Model-driven software verification. In *Spin Workshop*, April 2004.

[4] G. J. Holzmann. *The Spin Model Checker.* Wiley, New York, 2003.

[5] I.Rabinovitz and O.Grumberg. Bounded model checking of concurrent programs. In *Computer Aided Verification (CAV)*, 2005.

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

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

[8] M.Musuvathi, S.Qadeer, and T. Ball. Chess: A systematic testing tool for concurrent software. Technical report, Microsoft Research, 2007.

[9] P.Camara, M.Gallardo, and P.Merino. Model extraction for arinc 653 based avionics software. In *Spin workshop*, 2007.

[10] P.Godefroid. The VeriSoft Approach. *Formal Methods in System Design*, 26(2):77–101, 2005.

[11] S.Qadeer and D.Wu. Kiss: keep it simple and sequential. In *Programming Language Design and Implementation (PLDI)*, 2004.

[12] S.Savage, M.Burrows, G.Nelson, P.Sobalvarro, and T.Anderson. Eraser: A dynamic data race detector for multithreaded programming. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[13] T.A.Henzinger, R.Jhala, and R.Majumdar. Race checking by context inference. In *Programming Language Design and Implementation (PLDI)*, 2004.

[14] T.A.Henzinger, R.Jhala, R.Majumdar, and S.Qadeer. Thread-modular abstraction refinement. In *Computer Aided Verification (CAV)*, 2003.