COBET:

Effective COncurrency Bug dETector Framework for OS Kernel

Hong,Shin (advised by Prof. Moonzoo Kim)

Introduction

✓ Motivation

As multi-core hardware becomes increasingly powerful and popular, operating systems (OSes) such as Linux utilize the cutting-edge multithreaded techniques heavily to enhance performance.

However, current analysis techniques and tools for concurrent programs are not yet mature enough to support OS developers in a practical manner due to the unique characteristics of the kernel programming. In particular, the following three obstacles hinder analysis of the concurrent behavior of OSes.

- Various customized synchronization primitives

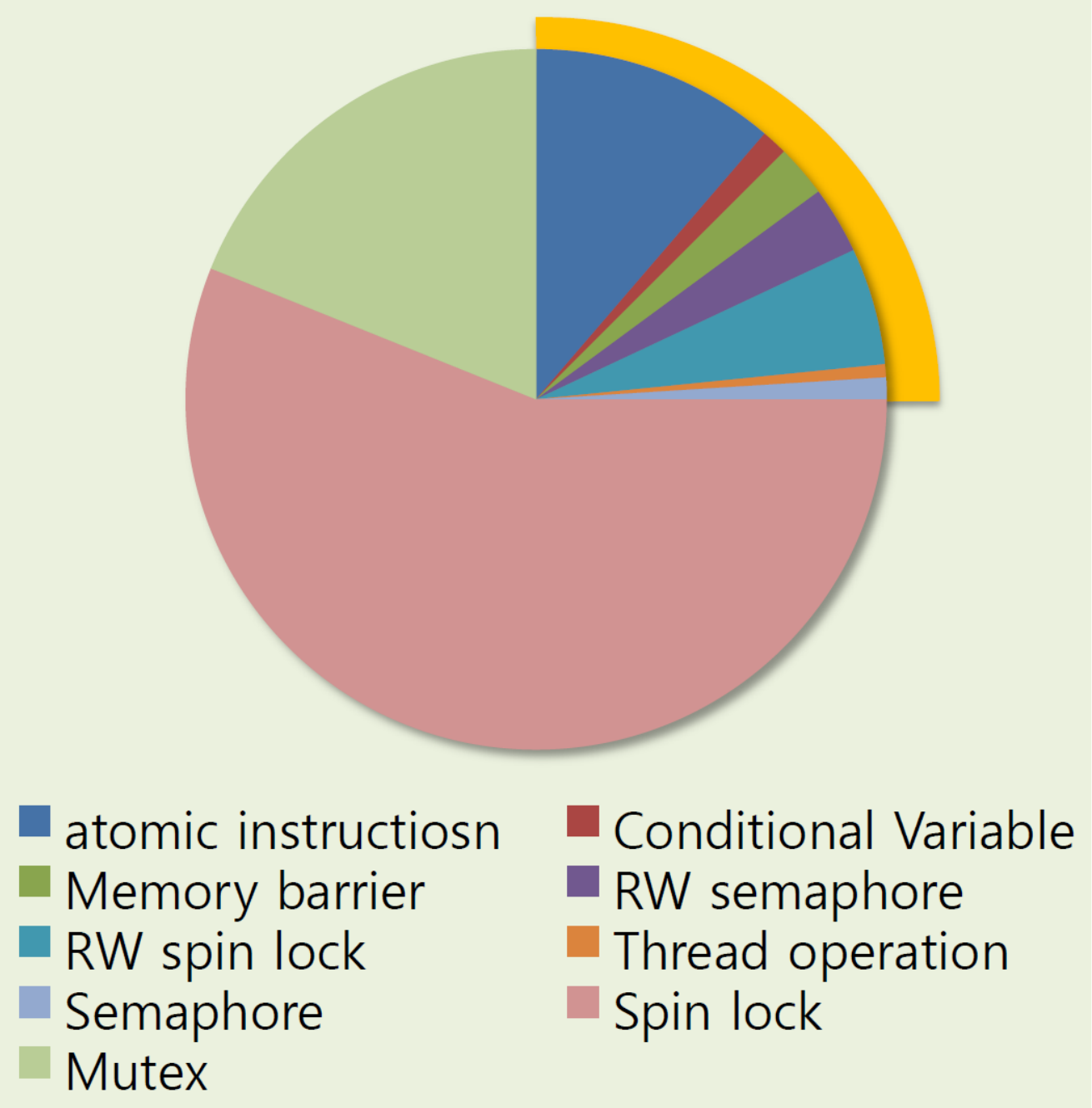
OS developers sometimes implement their own synchronization primitives. Concurrency bug detection tools for standard synchronization mechanisms do not recognize these customized synchronization mechanisms and produce imprecise results.

- Limitations of lock-based bug detections

Most available bug detection techniques focus on low−level data races through the analysis of binary lock usages. However, OSes exploit various synchronization mechanisms for performance. In addition, high−level data race and atomicity violations are more difficult to detect than low−level data races.

- Lack of scalability

A dynamic analysis often fails to uncover hidden concurrency bugs due to the exponential number of possible interleaving scenarios.



Statistics on synchronization statments in the Linux kernel 2.6.30.4

A static analysis, on the other hand, cannot analyze the OS code accurately due to its complex code and data structure. the monolithic structure of OSes severely hinder modular analyses.

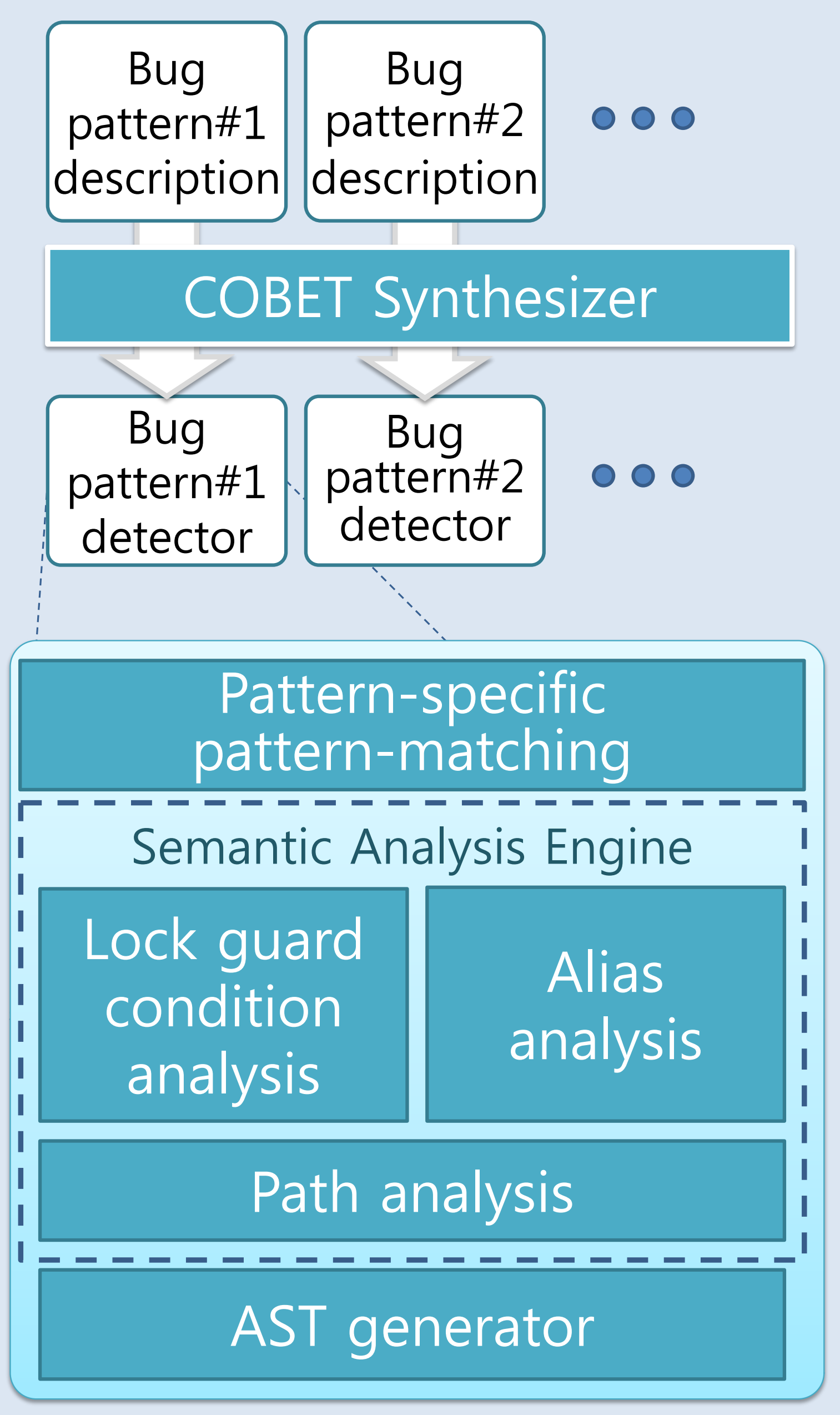
✓ Approach

We developed the COncurrency Bug dETector (COBET) framework that utilizes bug patterns augmented with semantic conditions.

A salient contribution of COBET is that it utilizes semantic information to define and detect bug patterns in a more precise manner while sustaining scalability, as previous research works utilizing syntactic patterns often raise many false alarms and examine relatively simple patterns.

As a user can define various concurrent bug patterns in a precise manner, COBET can detect real concurrency bugs that cannot be detected with conventional lock-based concurrency bug detection tools.

COBET Framework



✓ Overview

Most concurrency errors are caused by unintended interferences among concurrently executing multiple threads. Therefore, to detect a concurrency bug accurately, the concurrency bug pattern should be modeled with multiple code patterns each of which captures a specific code to be executed on each thread. Furthermore, it is necessary to check whether these multiple code patterns are possible to be executed concurrently at the same time, or not.

For this purpose, COBET provide the pattern description language (PDL) to describe the syntactic features of multiple code patterns. And the COBET semantic analysis engine enables the feasibility test, which is a salient feature of COBET compared to other pattern-based bug detection techniques.

✓ Pattern Detector Construction

The construction process has the three steps:

1. A user specifies the syntactic characteristics of a bug pattern in PDL.
2. The COBET synthesizer translates the PDL description to the corresponding bug pattern detector template code. The template code performs the syntactic pattern matching and calls `sem_cond_checking()` to check the semantic conditions.
3. The user fills out `sem_cond_checking()` function with the semantic condition checking routine to complete the bug pattern detector.

```
Bug-pattern ::= Sub-pattern+
Sub-pattern ::= pattern constant { Function+ }
Function  ::= fun Identifier { Stmt+ }
Stmt ::= if $cond { Stmt* }
       | if $cond { Stmt* } else { Stmt* }
       | loop $cond { Stmt* }       | break ;
       | lock Identifier;           | unlock Identifier;
       | read Identifier;           | write Identifier ;
       | call Identifier $args;     | \ { Stmt+ }
       | …
Identifier ::= constant | $<name>
```

Brief Grammar of PDL

PDL description

```
1a: pattern 1 {              1b: pattern 2 {
2a:  fun $f1 {               2b:  fun $f2 {
3a:   if $cond {             3b:   write $w ;
4a:    lock $l;              4b: }}
5a:    \{ if $cond { }}
6a:    unlock $l;
7a: }}}
```

Augmented semantic conditions

```
1:BOOL sem_cond_checking(bug_instance bi) {
2: if(is_lockset_exclusive(bi._3a,bi._3b) == FALSE)
3:  return FALSE;
4: if(is_shared_var(bi._w) == FALSE)
5:  return FALSE;
7: if(may_alias(bi._cond, bi._w) == FALSE)
8:  return FALSE;
9: return TRUE; }
```

COBET bug description of “Misused Test and Test-and-Set” pattern

✓ Five bug patterns with semantic conditions

Misused Test and Test-and-Set

```
1: void func1() {                        8: void func2() {
2:   if(condition(data)) {               9:   write data ;
3:     lock(m);                          10: }
4:     /* no if(condition(data))…*/
5:     ...
6:     unlock(m);
7:   }
8: }
```

- lock guard conditions at `line2` and `line9` must be exclusive.
- `data` at `line3` and `data` at `line9` may be aliasing.

Error scenario

```
2: if(condition(data))
                              6: write data
3: lock($m)
```

`condition(data)` might not be true

Unsynchronized communication at thread creation

```
1: void init_thread() {          5: void child(arg) {
2:  thread_run(child,arg);       6:   parameter = arg->data ;
3:  arg->data = setting ;        7: }
4: }
```

- lock guard conditions at `line2`, `line3` and `line6` must be exclusive
- `arg->data` at `line3` and `arg->data` at `line6` may be aliasing

Error scenario

```
2: thread_run(child,arg)
                              6: paramter = arg->data
3: arg->data = setting
```

`arg->data` is invalid

Busy-waiting without barrier

```
1: void wait_func() {            6: void notify_func() {
2:  loop(flag != true) {         8:  flag = true ;
3:   /* no barrier */            9: }
4:  }
5: }
```

- lock guard conditions at `line2` and `line5` must be exclusive
- `flag` at `line2` and `flag` at `line8` may be aliasing

Error scenario

```
2: (flag != true)
2: (flag != true)
                              8: flag = true
2: (flag != true)
2: (flag != true)
2: (flag != truee)
     :
```

unnecessary spinning

Using atomic instructions in non-atomic ways

```
1: void func1() {                6: void func2() {
                                 7:  atomic_inc(count) ;
2:  atomic_inc(count) ;          8: }
3:  if (atomic_read(count) > c){
     ...
4:  }
5: }
```

- lock guard conditions at `line2` and `line7` must be exclusive
- `count` at `line3` and `count` at `line7` may be aliasing

Error scenario

```
2: atomic_inc(count)
                              7: atomic_inc(count)
3:(atomic_read(count) > c)
```

`count` might be unexpected value

Waiting already terminated thread

```
1: void child() {                          8:  void parent() {
2:  loop(!kthread_should_stop()){          9:    $th = kthread_run(child);
3:   if (err)                              10:   kthread_stop($th) ;
4:    break ;                              11: }
5:  }
6:  /*no kthread_should_stop() checking*/
7: }
```

Error scenario

```
                                    9: th = kthread_run(child...)
2: (!kthread_should_stop())                  :
        :                                    :
3: (err == true)                             :
4: break;
  --- child terminates ---
                                    10: kthread_stop(th)
```

Deadlock

Experiments

✓ Bug Detection Result on File Systems

We applied the five COBET bug pattern detectors to the seven Linux file systems. The result shows the number of alarms was modest. (5 bug patterns X 7 file systems result 42 warnings) A relatively new file systems **btrfs** have several bugs and COBET approach effectively detected these bugs.

	btrfs (41KL)	ext4 (28KL)	nfs (29KL)	proc (8KL)	reiserfs (27KL)	sysfs (3KL)	udf (9KL)	vfs (48KL)	total (193KL)
Busy-waiting without barrier	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0
Misused test and test-and-set	3/0	3/0	4/0	2/0	3/0	1/0	2/0	10/0	28/0
Unsync. communication at thread creation	2/1	0/0	0/0	0/0	0/0	0/0	0/0	0/0	2/1
Using atomic inst. in non-atomic ways	4/0	0/0	2/0	0/0	2/0	0/0	0/0	1/0	9/0
Waiting already terminated thread	3/3	0/0	0/0	0/0	0/0	0/0	0/0	0/0	3/3
Total	12/4	3/0	6/0	3/0	4/0	1/0	2/0	11/0	42/4

✓ Evaluation of Semantic Analysis Techniques

To demonstrate the effectiveness of COBET semantic analyses, we measured the false alarm reduction rate through the semantic analyses and the additional time cost.

The result shows additional analyses reduce false alarms and the time cost were not burdensome.

	Syntactic analysis (single sub-pattern)		Syntactic analysis (multiple sub-patterns)		Syntactic analysis + path analysis + lock analysis		Syntactic analysis + path analysis + lock analysis + alias analysis	
	Detections	Time (sec)	Detections	Time (sec)	Detections	Time (sec)	Detections	Time (sec)
Busy-waiting without barrier	7	0.82	2	0.92	0	1.2	0	1.22
Misused test and test-and-set	51	0.67	36	2.58	32	4.51	28	4.37
Unsync. communication at thread creation	2	0.86	2	1.00	2	1.28	2	1.31
Using atomic inst. in non-atomic ways	12	0.70	9	0.86	9	1.44	9	1.45
Waiting already terminated thread	3	0.64	3	0.74	3	1.01	3	1.21
Total	75	3.69	52	6.10	46	9.44	42	9.56

✓ Bug Detection Results on Device Drivers and Network Modules

To demonstrate applicability, we applied 5 pattern detectors to Linux device drivers and network modules

	Device drivers			Network modules				
	bluetooth (11KL)	ieee1394 (25KL)	mtd (15KL)	atm (8KL)	ax25 (7KL)	netfilter (27KL)	rds(ib) (9KL)	Total (100KL)
Busy-waiting without barrier	1/0/0	0/0/0	0/0/0	0/0/0	0/0/0	12/0/0	1/0/0	14/0/0
Misused test test-and-set	0/0/0	4/1/0	0/0/0	1/1/1	6/3/1	5/1/1	5/1/0	21/7/3
Unsync. communication at thread creation	0/0/0	0/0/0	1/1/1	0/0/0	0/0/0	0/0/0	0/0/0	1/1/1
Using atomic inst. in non-atomic ways	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	2/1/1	4/2/1	6/3/2
Waiting already terminated thread	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Total	1/0/0	4/1/0	1/1/1	1/1/1	6/3/1	19/2/2	11/3/1	42/11/6

Detected bug of “Misused Test and Test-and-Set” pattern

```
// Matching with pattern 1
1f:void br2684_push(atm_vcc *atmvcc, sk_buff *skb) {
2f:  ...
3f:  if (list_empty(&brdev->brvccs)) {
4f:    write_lock_irq(&devs_lock) ;
5f:    list_del(&brdev->br2684_devs) ;
6f:    write_unlock_irq(&devs_lock) ;
// Matching with pattern 2
1g:int br2684_regvcc(atm_vcc *atmvcc, __user *arg) {
2g:  ...
3g:  write_lock_irq(&devs_lock) ;
4g:  ...
5g:  list_add(&brvcc->brvccs, &brdev->brvccs) ;
6g:  write_unlock_irq(&devs_lock);
```

Further work

✓Patternize conventional data race and atomicity violations

Many bug definitions used for standard lock-based concurrency bug detection techniques such as data race or atomicity violations can be represented or approximated to COBET patterns.

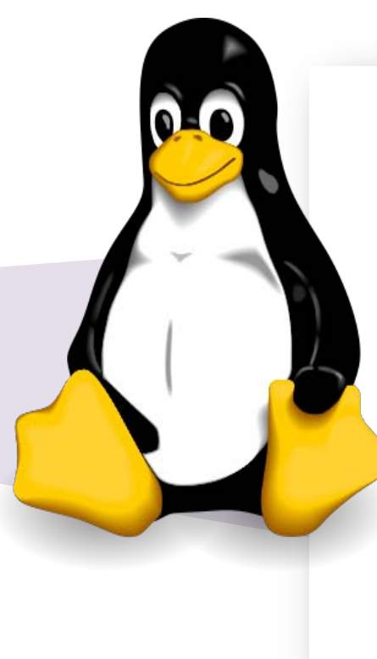
✓Apply to Application-level programs

Large-size application programs such as HTTP daemon or DBMS systems suffer similar difficulties to Linux kernel programming. We plan to apply COBET approach and COBET bug patterns to find bugs in these domain and compare the result to the case of Linux kernel.

✓Enhance PDL to include semantic conditions specification

Currently, PDL only specifies syntactic aspects of bug patterns. We plan to extend PDL to associate essential semantic conditions to improve usability of COBET framework.

Answer to the reporting detected bug
from the corresponding Linux maintainer



> *… that seems pretty reasonable. i dont believe you would see this race in practice given the serialization from the userspace code for startup and shutdown. however, this isnt a reason not do things correctly. …* [5]

KAIST
Computer
Science

PROVABLE
SOFTWARE LAB

hongshin@kaist.ac.kr
http://pswlab.kaist.ac.kr