# Meta-level Compilation

Presentation by Tony Ling

# What Is This Presentation About?

- A problem in software systems
- Common solutions and their inefficiencies
- Introduction meta-level compilation and examples

# What is the problem?

**Software systems have rules they must be obeyed**

**Examples of System Rules:**

- **Variable A must be protected by lock B before being accessed**
- **Message handlers should be free buffers ASAP**
- **Interrupts must be enabled after being disabled**
- **Shared variables not modified should be protected with read locks**

# Common Methods of Rule Violations Testing

- Formal Verification
- Test Cases
- Manual Inspection

# Formal Verification

**Using theorem provers or model checkers by building abstract formal specification**

# Formal Verification

**Using theorem provers or model checkers by building abstract formal specification**

**Some Problems:**

- **Cost Effectiveness**
- **Specifications are not easy to build**
- **What about agile software development?**

# Test Cases

**Creating test cases to test for rules dynamically**

# Test Cases

**Creating test cases to test for rules dynamically**

**Issues:**

- **Time consuming**
- **Narrowing down the cause of bug**
- **Practicality**

# Manual Inspection

**Programmers looking through code checking for inconsistencies or errors**

# Manual Inspection

**Programmers looking through code checking for inconsistencies or errors**

**Problems:**

- **Large code base**
- **Humans, can’t trust them**

# Rule Template

**System rules follow certain templates:**

- **Never/always do X**
- **Do X rather than Y**
- **Always do X before/after Y**
- **Never do X before/after Y**
- **In situation X, do/not do Y**
- **In situation X, do Y rather than Z**

# Rule Template

- Never/always do X
- Do X rather than Y
- Always do X before/after Y
- Never do X before/after Y

- In situation X, do/not do Y
- In situation X, do Y rather

- Enable interrupts after disabling them
- Never read before obtaining lock
- If a shared variable is not modified, protect with read lock

# Static Compiler Analysis

**Compilers are good at analyzing source code.**

**Removes need to tests program by running it.**

**Can detect sequence order of code execution.**

# Static Compiler Analysis

**Compilers are good at analyzing source code.**

**Removes need to tests program by running it.**

**Can detect sequence order of code execution.**

**Problem:**

- **Compilers ignorant of the ‘meta’ semantics of the software system**

# Meta-level Compilation

- **Allow developers to extend compilers with system specific checkers**
- **Compiler extensions are written in a high level state-machine language called *metal***
- **These extensions check for specific system rules**

# How does it work?

- A *metal* extensions is a collection of state machines
- Syntax pattern matching triggers state transitions
- Patterns are written in an extended version of the base language
- Extensions can run in either flow insensitive or flow sensitive modes
- Extensions goes through all flow paths of source code while keep tracking states

# Example 1

```
sm check_interrupts{
    decl {unsigned} any_flag;

    pat enable = { sti(); }
               | { restore_flags(any_flag);};
    pat disable = { cli(); };

    is_enabled: disabled ==> is_disabled
              | enable ==> { err(“double enable”); };
    is_disabled: enable ==> is_enabled
               | disabled == > { err(“double disable”); }
               | $end_of_path$ ⇒ { err(“exiting w/intr disabled!”); };
}
```

**Engler, Dawson et al. "Checking system rules using system-specific, programmer-written compiler extensions."** ***Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*** **22 Oct. 2000: 1-1.**

# Example 1

```
sm check_interrupts{
    decl {unsigned} any_flag;

    pat enable = { sti(); }
               | { restore_flags(any_flag);};
    pat disable = { cli(); };

    is_enabled: disabled ==> is_disabled
              | enable ==> { err("double enable"); };
    is_disabled: enable ==> is_enabled
               | disabled == > { err("double disable"); }
               | $end_of_path$ ⇒ { err("exiting w/intr disabled!"); };
}
```

Wild card variable that matches any expression of type “unsigned”

**Engler, Dawson et al. "Checking system rules using system-specific, programmer-written compiler extensions."** ***Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*** **22 Oct. 2000: 1-1.**

# Example 1

```
sm check_interrupts{
    decl {unsigned} any_flag;

    pat enable = { sti(); }
               | { restore_flags(any_flag);};
    pat disable = { cli(); };

    is_enabled: disabled ==> is_disabled
              | enable ==> { err("double enable"); };
    is_disabled: enable ==> is_enabled
               | disabled == > { err("double disable"); }
               | $end_of_path$ ⇒ { err("exiting w/intr disabled!"); };
}
```

Wild card variable that matches any expression of type "unsigned"

Defines pattern in source code to match

**Engler, Dawson et al. "Checking system rules using system-specific, programmer-written compiler extensions."** ***Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*** **22 Oct. 2000: 1-1.**

# Example 1

```
sm check_interrupts{
    decl {unsigned} any_flag;

    pat enable = { sti(); }
               | { restore_flags(any_flag);};
    pat disable = { cli(); };

    is_enabled: disabled ==> is_disabled
              | enable ==> { err("double enable"); };
    is_disabled: enable ==> is_enabled
               | disabled == > { err("double disable"); }
               | $end_of_path$ ⇒ { err("exiting w/intr disabled!"); };
}
```

Wild card variable that matches any expression of type "unsigned"

Defines pattern in source code to match

Defines state and transition edges and actions to take if a pattern is matched

**Engler, Dawson et al. "Checking system rules using system-specific, programmer-written compiler extensions."** ***Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*** **22 Oct. 2000: 1-1.**

# Example 1

```
sm check_interrupts{
    decl {unsigned} any_flag;

    pat enable = { sti(); }
               | { restore_flags(any_flag);};
    pat disable = { cli(); };

    is_enabled: disabled ==> is_disabled
              | enable ==> { err("double enable"); };
    is_disabled: enable ==> is_enabled
               | disabled == > { err("double disable"); }
               | $end_of_path$ ⇒ { err("exiting w/intr disabled!"); };
}
```

Wild card variable that matches any expression of type "unsigned"

Defines pattern in source code to match

matched if exiting out of scope

Defines state and transition edges and actions to take if a pattern is matched

**Engler, Dawson et al. "Checking system rules using system-specific, programmer-written compiler extensions."** ***Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*** **22 Oct. 2000: 1-1.**

# Example 1

```
sm check_interrupts{
  decl {unsigned} any_flag;

  pat enable = { sti(); }
    | { restore_flags(any_flag); };
  pat disable = { cli(); };

  is_enabled:
       disabled ==> is_disabled
    | enable ==> { err("double
enable");};
  is_disabled: enable ==> is_enabled
    | disabled ==> { err("double
disable");}
    | $end_of_path$ ==> { err("exiting
w/intr disabled!"); };
}
```

```
/* From Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct strip_head *sh,
                    int b_size) {
     struct buffer_head *bh;
     unsigned long flags;

     save_flags(flags);
     cli();
     if ((bh = sh->buffer_pool) == NULL)
          return NULL;
     sh->buffer_pool = bh->b_next;
     bh->b_size = b_size;
     restore_flags(flags);
     return bh;
}
```

sm: is_enabled

**Engler, Dawson et al. "Checking system rules using system-specific, programmer-written compiler extensions."** ***Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*** **22 Oct. 2000: 1-1.**

# Example 1

```
sm check_interrupts{
  decl {unsigned} any_flag;

  pat enable = { sti(); }
    | { restore_flags(any_flag); };
  pat disable = { cli(); };

  is_enabled:
     disabled ==> is_disabled
    | enable ==> { err("double
enable");};
  is_disabled: enable ==> is_enabled
    | disabled ==> { err("double
disable");}
    | $end_of_path$ ==> { err("exiting
w/intr disabled!"); };
}
```

```
/* From Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct strip_head *sh,
                    int b_size) {
    struct buffer_head *bh;
    unsigned long flags;

    save_flags(flags);
    cli();
    if ((bh = sh->buffer_pool) == NULL)
        return NULL;
    sh->buffer_pool = bh->b_next;
    bh->b_size = b_size;
    restore_flags(flags);
    return bh;
}
```

sm: is_enabled

Engler, Dawson et al. "Checking system rules using system-specific, programmer-written compiler extensions." *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4* 22 Oct. 2000: 1-1.

# Example 1

```
sm check_interrupts{
  decl {unsigned} any_flag;

  pat enable = { sti(); }
    | { restore_flags(any_flag); };
  pat disable = { cli(); };

  is_enabled:
     disabled ==> is_disabled
    | enable ==> { err("double
enable");};
  is_disabled: enable ==> is_enabled
    | disabled ==> { err("double
disable");}
    | $end_of_path$ ==> { err("exiting
w/intr disabled!"); };
}
```

```
/* From Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct strip_head *sh,
                    int b_size) {
    struct buffer_head *bh;
    unsigned long flags;

    save_flags(flags);
    cli();
    if ((bh = sh->buffer_pool) == NULL)
        return NULL;
    sh->buffer_pool = bh->b_next;
    bh->b_size = b_size;
    restore_flags(flags);
    return bh;
}
```

sm: is_disabled

**Engler, Dawson et al. "Checking system rules using system-specific, programmer-written compiler extensions."** ***Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*** **22 Oct. 2000: 1-1.**

# Example 1

```
sm check_interrupts{
  decl {unsigned} any_flag;

  pat enable = { sti(); }
    | { restore_flags(any_flag); };
  pat disable = { cli(); };

  is_enabled:
       disabled ==> is_disabled
    | enable ==> { err("double
enable");};
  is_disabled: enable ==> is_enabled
    | disabled ==> { err("double
disable");}
    | $end_of_path$ ==> { err("exiting
w/intr disabled!"); };
}
```

```
/* From Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct strip_head *sh,
                    int b_size) {
     struct buffer_head *bh;
     unsigned long flags;

     save_flags(flags);
     cli();
     if ((bh = sh->buffer_pool) == NULL)
          return NULL;
     sh->buffer_pool = bh->b_next;
     bh->b_size = b_size;
     restore_flags(flags);
     return bh;
}
```

sm: is_disabled

**Engler, Dawson et al. "Checking system rules using system-specific, programmer-written compiler extensions."** ***Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*** **22 Oct. 2000: 1-1.**

# Example 1

```
sm check_interrupts{
  decl {unsigned} any_flag;

  pat enable = { sti(); }
    | { restore_flags(any_flag); };
  pat disable = { cli(); };

  is_enabled:
       disabled ==> is_disabled
    | enable ==> { err("double
enable");};
  is_disabled: enable ==> is_enabled
    | disabled ==> { err("double
disable");}
    | $end_of_path$ ==> { err("exiting
w/intr disabled!"); };
}
```

```
/* From Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct strip_head *sh,
                    int b_size) {
    struct buffer_head *bh;
    unsigned long flags;

    save_flags(flags);
    cli();
    if ((bh = sh->buffer_pool) == NULL)
        return NULL;
    sh->buffer_pool = bh->b_next;
    bh->b_size = b_size;
    restore_flags(flags);
    return bh;
}
```

sm: is_disabled

$end_of_oath$ ==> error

**Engler, Dawson et al. "Checking system rules using system-specific, programmer-written compiler extensions."** ***Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*** **22 Oct. 2000: 1-1.**

# Example 1

```
sm check_interrupts{
  decl {unsigned} any_flag;

  pat enable = { sti(); }
    | { restore_flags(any_flag); };
  pat disable = { cli(); };

  is_enabled:
       disabled ==> is_disabled
    | enable ==> { err("double
enable");};
  is_disabled: enable ==> is_enabled
    | disabled ==> { err("double
disable");}
    | $end_of_path$ ==> { err("exiting
w/intr disabled!"); };
}
```

```
/* From Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct strip_head *sh,
                    int b_size) {
     struct buffer_head *bh;
     unsigned long flags;

     save_flags(flags);
     cli();
     if ((bh = sh->buffer_pool) == NULL)
          return NULL;
     sh->buffer_pool = bh->b_next;
     bh->b_size = b_size;
     restore_flags(flags);
     return bh;
}
```

sm: is_disabled

**Engler, Dawson et al. "Checking system rules using system-specific, programmer-written compiler extensions."** ***Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*** **22 Oct. 2000: 1-1.**

# Example 1

```
sm check_interrupts{
  decl {unsigned} any_flag;

  pat enable = { sti(); }
    | { restore_flags(any_flag); };
  pat disable = { cli(); };

  is_enabled:
       disabled ==> is_disabled
    | enable ==> { err("double
enable");};
  is_disabled: enable ==> is_enabled
    | disabled ==> { err("double
disable");}
    | $end_of_path$ ==> { err("exiting
w/intr disabled!"); };
}
```

```
/* From Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct strip_head *sh,
                    int b_size) {
    struct buffer_head *bh;
    unsigned long flags;

    save_flags(flags);
    cli();
    if ((bh = sh->buffer_pool) == NULL)
        return NULL;
    sh->buffer_pool = bh->b_next;
    bh->b_size = b_size;
    restore_flags(flags);
    return bh;
}
```

sm: is_disabled

Variable 'flag' matches 'any_flags' since it is of type unsigned

**Engler, Dawson et al. "Checking system rules using system-specific, programmer-written compiler extensions."** ***Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*** **22 Oct. 2000: 1-1.**

# Example 1

```
sm check_interrupts{
  decl {unsigned} any_flag;

  pat enable = { sti(); }
    | { restore_flags(any_flag); };
  pat disable = { cli(); };

  is_enabled:
       disabled ==> is_disabled
    | enable ==> { err("double
enable");};
  is_disabled: enable ==> is_enabled
    | disabled ==> { err("double
disable");}
    | $end_of_path$ ==> { err("exiting
w/intr disabled!"); };
}
```

```
/* From Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct strip_head *sh,
                    int b_size) {
     struct buffer_head *bh;
     unsigned long flags;

     save_flags(flags);
     cli();
     if ((bh = sh->buffer_pool) == NULL)
          return NULL;
     sh->buffer_pool = bh->b_next;
     bh->b_size = b_size;
     restore_flags(flags);
     return bh;
}
```

sm: is_enabled

**Engler, Dawson et al. "Checking system rules using system-specific, programmer-written compiler extensions."** ***Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*** **22 Oct. 2000: 1-1.**

# Example 1

```
sm check_interrupts{
  decl {unsigned} any_flag;

  pat enable = { sti(); }
    | { restore_flags(any_flag); };
  pat disable = { cli(); };

  is_enabled:
       disabled ==> is_disabled
    | enable ==> { err("double
enable");};
  is_disabled: enable ==> is_enabled
    | disabled ==> { err("double
disable");}
    | $end_of_path$ ==> { err("exiting
w/intr disabled!"); };
}
```

```
/* From Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct strip_head *sh,
                    int b_size) {
     struct buffer_head *bh;
     unsigned long flags;

     save_flags(flags);
     cli();
     if ((bh = sh->buffer_pool) == NULL)
          return NULL;
     sh->buffer_pool = bh->b_next;
     bh->b_size = b_size;
     restore_flags(flags);
     return bh;
}
```

sm: is_enabled

**Engler, Dawson et al. "Checking system rules using system-specific, programmer-written compiler extensions."** ***Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*** **22 Oct. 2000: 1-1.**

# Example 2

```
1: state decl any_pointer v;
2:
3: start: { kfree(v) } ==> v.freed;
4:
5: v.freed: { *v } ==> v.stop,
6:     { err(“using %s after free!”, mc_identifier(v)); }
7: | { kfree(v) } ==> v.stop,
8:     { err(“double free of %s!”, mc_identifier(v)); }
9: ;
```

**Hallem, Seth et al. *A system and language for building system-specific, static analyses*. ACM, 2002.**

# Example 2

```
1: state decl any_pointer v;
2:
3: start: { kfree(v) } ==> v.freed;
4:
5: v.freed: { *v } ==> v.stop,
6:     { err("using %s after free!", mc_identifier(v)); }
7: | { kfree(v) } ==> v.stop,
8:     { err("double free of %s!", mc_identifier(v)); }
9: ;
```

Variable-specific state variable instances have their own states

Hallem, Seth et al. ***A system and language for building system-specific, static analyses***. **ACM, 2002.**

# Example 2

```
1: state decl any_pointer v;
2:
3: start: { kfree(v) } ==> v.freed;
4:
5: v.freed: { *v } ==> v.stop,
6:     { err("using %s after free!", mc_identifier(v)); }
7: | { kfree(v) } ==> v.stop,
8:     { err("double free of %s!", mc_identifier(v)); }
9: ;
```

Variable-specific state variable instances have their own states

Hallem, Seth et al. ***A system and language for building system-specific, static analyses***. **ACM, 2002.**

# Example 2

```
1: state decl any_pointer v;
2:
3: start: { kfree(v) } ==> v.freed;
4:
5: v.freed: { *v } ==> v.stop,
6:     { err("using %s after free!", mc_identifier(v)); }
7: | { kfree(v) } ==> v.stop,
8:     { err("double free of %s!", mc_identifier(v)); }
9: ;
```

Variable-specific state variable instances have their own states

Hallem, Seth et al. ***A system and language for building system-specific, static analyses***. **ACM, 2002.**

# Example 2

```
1: state decl any_pointer v;
2:
3: start: { kfree(v) } ==> v.freed;
4:
5: v.freed: { *v } ==> v.stop,
6:     { err("using %s after free!", mc_identifier(v)); }
7: | { kfree(v) } ==> v.stop,
8:     { err("double free of %s!", mc_identifier(v)); }
9: ;
```

Variable-specific state variable instances have their own states

v.stop is a special state that means that the instance of v is no longer tracked

Hallem, Seth et al. *A system and language for building system-specific, static analyses*. ACM, 2002.

# Example 2

```
1: state decl any_pointer v;
2:
3: start: { kfree(v) } ==> v.freed;
4:
5: v.freed: { *v } ==> v.stop,
6:       { err("using %s after
free!", mc_identifier(v)); }
7: | { kfree(v) } ==> v.stop,
8:       { err("double free of %s!",
mc_identifier(v)); }
9: ;
```

```
1: int contrived(int *p, int *w, int x) {
2:   int *q;
3:
4:   if(x)
5:   {
6:      kfree(w);
7:      q = p;
8:      p = 0;
9:   }
10:  if(!x)
11:     return *w;
12:  return *q;
13:}
14:int contrived_caller(int *w, int x, int *p {
15:  kfree(p);
16:  contrived (p, w, x);
17:  return *w;
18:}
```

**Hallem, Seth et al. *A system and language for building system-specific, static analyses*. ACM, 2002.**

# Example 2

```
1: state decl any_pointer v;
2:
3: start: { kfree(v) } ==> v.freed;
4:
5: v.freed: { *v } ==> v.stop,
6:       { err("using %s after
free!", mc_identifier(v)); }
7: | { kfree(v) } ==> v.stop,
8:       { err("double free of %s!",
mc_identifier(v)); }
9: ;
```

```
1: int contrived(int *p, int *w, int x) {
2:   int *q;
3:
4:   if(x)
5:   {
6:      kfree(w);
7:      q = p;
8:      p = 0;
9:   }
10:  if(!x)
11:     return *w;
12:  return *q;
13:}
14:int contrived_caller(int *w, int x, int *p {
15:  kfree(p);
16:  contrived (p, w, x);
17:  return *w;
18:}
```

start at contrived_caller

**Hallem, Seth et al. *A system and language for building system-specific, static analyses*. ACM, 2002.**

# Example 2

```
1: state decl any_pointer v;
2:
3: start: { kfree(v) } ==> v.freed;
4:
5: v.freed: { *v } ==> v.stop,
6:       { err("using %s after
free!", mc_identifier(v)); }
7: | { kfree(v) } ==> v.stop,
8:       { err("double free of %s!",
mc_identifier(v)); }
9: ;
```

```
1: int contrived(int *p, int *w, int x) {
2:   int *q;
3:
4:   if(x)
5:   {
6:      kfree(w);
7:      q = p;
8:      p = 0;
9:   }
10:  if(!x)
11:     return *w;
12:  return *q;
13:}
14:int contrived_caller(int *w, int x, int *p {
15:  kfree(p);
16:  contrived (p, w, x);
17:  return *w;
18:}
```

v:p-> freed

Hallem, Seth et al. ***A system and language for building system-specific, static analyses***. ACM, 2002.

# Example 2

```
1: state decl any_pointer v;
2:
3: start: { kfree(v) } ==> v.freed;
4:
5: v.freed: { *v } ==> v.stop,
6:       { err("using %s after
free!", mc_identifier(v)); }
7: | { kfree(v) } ==> v.stop,
8:       { err("double free of %s!",
mc_identifier(v)); }
9: ;
```

```
1: int contrived(int *p, int *w, int x) {
2:   int *q;
3:
4:   if(x)
5:   {
6:      kfree(w);
7:      q = p;
8:      p = 0;
9:   }
10:  if(!x)
11:     return *w;
12:  return *q;
13:}
14:int contrived_caller(int *w, int x, int *p {
15:  kfree(p);
16:  contrived (p, w, x);
17:  return *w;
18:}
```

v:p-> freed

Hallem, Seth et al. ***A system and language for building system-specific, static analyses***. ACM, 2002.

# Example 2

```
1: state decl any_pointer v;
2:
3: start: { kfree(v) } ==> v.freed;
4:
5: v.freed: { *v } ==> v.stop,
6:       { err("using %s after
free!", mc_identifier(v)); }
7: | { kfree(v) } ==> v.stop,
8:       { err("double free of %s!",
mc_identifier(v)); }
9: ;
```

```
1: int contrived(int *p, int *w, int x) {
2:   int *q;
3:
4:   if(x)
5:   {
6:      kfree(w);
7:      q = p;
8:      p = 0;
9:   }
10:  if(!x)
11:     return *w;
12:  return *q;
13:}
14:int contrived_caller(int *w, int x, int *p {
15:  kfree(p);
16:  contrived (p, w, x);
17:  return *w;
18:}
```

v:p-> freed

x = true

Hallem, Seth et al. *A system and language for building system-specific, static analyses*. ACM, 2002.

# Example 2

```
1: state decl any_pointer v;
2:
3: start: { kfree(v) } ==> v.freed;
4:
5: v.freed: { *v } ==> v.stop,
6:       { err("using %s after
free!", mc_identifier(v)); }
7: | { kfree(v) } ==> v.stop,
8:       { err("double free of %s!",
mc_identifier(v)); }
9: ;
```

```
1: int contrived(int *p, int *w, int x) {
2:   int *q;
3:
4:   if(x)
5:   {
6:      kfree(w);
7:      q = p;
8:      p = 0;
9:   }
10:  if(!x)
11:     return *w;
12:  return *q;
13:}
14:int contrived_caller(int *w, int x, int *p {
15:  kfree(p);
16:  contrived (p, w, x);
17:  return *w;
18:}
```

v:p-> freed
v:w-> freed

x = true

**Hallem, Seth et al. *A system and language for building system-specific, static analyses*. ACM, 2002.**

# Example 2

```
1: state decl any_pointer v;
2:
3: start: { kfree(v) } ==> v.freed;
4:
5: v.freed: { *v } ==> v.stop,
6:       { err("using %s after
free!", mc_identifier(v)); }
7: | { kfree(v) } ==> v.stop,
8:       { err("double free of %s!",
mc_identifier(v)); }
9: ;
```

```
1: int contrived(int *p, int *w, int x) {
2:   int *q;
3:
4:   if(x)
5:   {
6:      kfree(w);
7:      q = p;
8:      p = 0;
9:   }
10:  if(!x)
11:     return *w;
12:  return *q;
13:}
14:int contrived_caller(int *w, int x, int *p {
15:  kfree(p);
16:  contrived (p, w, x);
17:  return *w;
18:}
```

v:p-> freed
v:w-> freed
v:q-> freed

x = true

instance of v created due to assignment

**Hallem, Seth et al. *A system and language for building system-specific, static analyses*. ACM, 2002.**

# Example 2

```
1: state decl any_pointer v;
2:
3: start: { kfree(v) } ==> v.freed;
4:
5: v.freed: { *v } ==> v.stop,
6:       { err("using %s after
free!", mc_identifier(v)); }
7: | { kfree(v) } ==> v.stop,
8:       { err("double free of %s!",
mc_identifier(v)); }
9: ;
```

```
1: int contrived(int *p, int *w, int x) {
2:   int *q;
3:
4:   if(x)
5:   {
6:      kfree(w);
7:      q = p;
8:      p = 0;
9:   }
10:  if(!x)
11:     return *w;
12:  return *q;
13:}
14:int contrived_caller(int *w, int x, int *p {
15:  kfree(p);
16:  contrived (p, w, x);
17:  return *w;
18:}
```

x = true

v:w-> freed
v:q-> freed

v:p goes to stop state and removed

**Hallem, Seth et al. *A system and language for building system-specific, static analyses*. ACM, 2002.**

# Example 2

```
1: state decl any_pointer v;
2:
3: start: { kfree(v) } ==> v.freed;
4:
5: v.freed: { *v } ==> v.stop,
6:       { err("using %s after
free!", mc_identifier(v)); }
7: | { kfree(v) } ==> v.stop,
8:       { err("double free of %s!",
mc_identifier(v)); }
9: ;
```

```
1: int contrived(int *p, int *w, int x) {
2:   int *q;
3:
4:   if(x)
5:   {
6:      kfree(w);
7:      q = p;
8:      p = 0;
9:   }
10:  if(!x)
11:     return *w;
12:  return *q;
13:}
14:int contrived_caller(int *w, int x, int *p {
15:  kfree(p);
16:  contrived (p, w, x);
17:  return *w;
18:}
```

x = true

v:w-> freed
v:q-> freed

Hallem, Seth et al. ***A system and language for building system-specific, static analyses***. ACM, 2002.

# Example 2

```
1: state decl any_pointer v;
2:
3: start: { kfree(v) } ==> v.freed;
4:
5: v.freed: { *v } ==> v.stop,
6:       { err("using %s after
free!", mc_identifier(v)); }
7: | { kfree(v) } ==> v.stop,
8:       { err("double free of %s!",
mc_identifier(v)); }
9: ;
```

```
1: int contrived(int *p, int *w, int x) {
2:   int *q;
3:
4:   if(x)
5:   {
6:      kfree(w);
7:      q = p;
8:      p = 0;
9:   }
10:  if(!x)
11:     return *w;
12:  return *q;
13:}
14:int contrived_caller(int *w, int x, int *p {
15:  kfree(p);
16:  contrived (p, w, x);
17:  return *w;
18:}
```

x = true

v:w-> freed

v:q goes to stop state and is removed

**Hallem, Seth et al. *A system and language for building system-specific, static analyses*. ACM, 2002.**

# Example 2

```
1: state decl any_pointer v;
2:
3: start: { kfree(v) } ==> v.freed;
4:
5: v.freed: { *v } ==> v.stop,
6:       { err("using %s after
free!", mc_identifier(v)); }
7: | { kfree(v) } ==> v.stop,
8:       { err("double free of %s!",
mc_identifier(v)); }
9: ;
```

```
1: int contrived(int *p, int *w, int x) {
2:   int *q;
3:
4:   if(x)
5:   {
6:      kfree(w);
7:      q = p;
8:      p = 0;
9:   }
10:  if(!x)
11:     return *w;
12:  return *q;
13:}
14:int contrived_caller(int *w, int x, int *p {
15:  kfree(p);
16:  contrived (p, w, x);
17:  return *w;
18:}
```

v:p-> freed

x = false

**Hallem, Seth et al. *A system and language for building system-specific, static analyses*. ACM, 2002.**

# Example 2

```
1: state decl any_pointer v;
2:
3: start: { kfree(v) } ==> v.freed;
4:
5: v.freed: { *v } ==> v.stop,
6:       { err("using %s after
free!", mc_identifier(v)); }
7: | { kfree(v) } ==> v.stop,
8:       { err("double free of %s!",
mc_identifier(v)); }
9: ;
```

```
1: int contrived(int *p, int *w, int x) {
2:   int *q;
3:
4:   if(x)
5:   {
6:      kfree(w);
7:      q = p;
8:      p = 0;
9:   }
10:  if(!x)
11:     return *w;
12:  return *q;
13:}
14:int contrived_caller(int *w, int x, int *p {
15:  kfree(p);
16:  contrived (p, w, x);
17:  return *w;
18:}
```

v:p-> freed

x = false

Hallem, Seth et al. ***A system and language for building system-specific, static analyses***. ACM, 2002.

# Example 2

```
1: state decl any_pointer v;
2:
3: start: { kfree(v) } ==> v.freed;
4:
5: v.freed: { *v } ==> v.stop,
6:       { err("using %s after
free!", mc_identifier(v)); }
7: | { kfree(v) } ==> v.stop,
8:       { err("double free of %s!",
mc_identifier(v)); }
9: ;
```

```
1: int contrived(int *p, int *w, int x) {
2:   int *q;
3:
4:   if(x)
5:   {
6:      kfree(w);
7:      q = p;
8:      p = 0;
9:   }
10:  if(!x)
11:     return *w;
12:  return *q;
13:}
14:int contrived_caller(int *w, int x, int *p {
15:  kfree(p);
16:  contrived (p, w, x);
17:  return *w;
18:}
```

v:p-> freed

x = false

Hallem, Seth et al. ***A system and language for building system-specific, static analyses***. ACM, 2002.

# Example 2

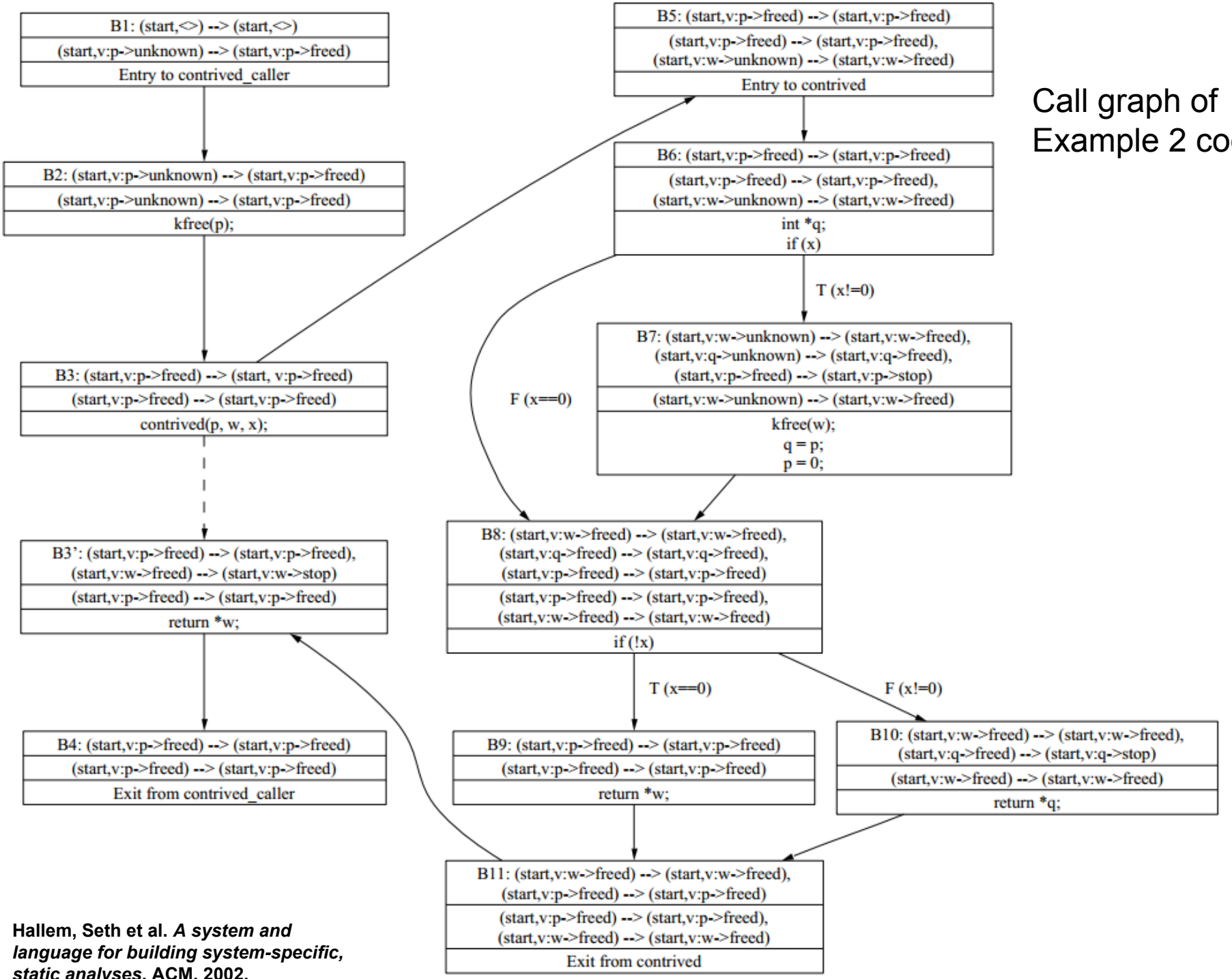
```
1: state decl any_pointer v;
2:
3: start: { kfree(v) } ==> v.freed;
4:
5: v.freed: { *v } ==> v.stop,
6:       { err("using %s after
free!", mc_identifier(v)); }
7: | { kfree(v) } ==> v.stop,
8:       { err("double free of %s!",
mc_identifier(v)); }
9: ;
```

```
1: int contrived(int *p, int *w, int x) {
2:   int *q;
3:
4:   if(x)
5:   {
6:      kfree(w);
7:      q = p;
8:      p = 0;
9:   }
10:  if(!x)
11:     return *w;
12:  return *q;
13:}
14:int contrived_caller(int *w, int x, int *p {
15:  kfree(p);
16:  contrived (p, w, x);
17:  return *w;
18:}
```

v:p-> freed
v:q-> freed

contrived returns with
v:p and v:w

**Hallem, Seth et al. *A system and language for building system-specific, static analyses*. ACM, 2002.**




Call graph of Example 2 code

**Hallem, Seth et al. *A system and language for building system-specific, static analyses*. ACM, 2002.**

# Some Problems with MC

**Extensions are not verifiers**

**False positive and false negatives occur**

**Underlying state machine**

# Coverity

**Coverity founded by Dawson Engler, Seth Hallem, Andy Chou, Benjamin Chelf in 2002**

**According to Crunchbase as of 10/20/2014:**

- **Received $22 million in round of investment in 2008**
- **Acquired by Synopsys for $350 million in February 19, 2014**

# References


Engler, Dawson et al. "Checking system rules using system-specific, programmer-written compiler extensions." *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4* 22 Oct. 2000: 1-1.

Hallem, Seth et al. *A system and language for building system-specific, static analyses*. ACM, 2002.

http://www.crunchbase.com/organization/coverity