

# Is Exception Handling an Aspect?

Tom Tourwé  
Software Engineering Cluster  
C.W.I. Amsterdam

joint work with Magiel Bruntink and Arie van Deursen

# Context

## Exception handling at ASML

```
void f(int a, int *b) {  
    g(a);  
    h(b);  
}
```

# Context

## Exception handling at ASML

```
int f(int a, int *b) {  
    int r = OK;  
    r = g(a);  
    if (r != OK) {  
        LOG(CCXA_error, r);  
        r = CCXA_error;  
    }  
    if (r == OK) {  
        r = h(b);  
    }  
    return r;  
}
```

# Context

## Exception handling at ASML

```
int f(int a, int *b) {  
    int r = OK; error variable  
    r = g(a);  
    if (r != OK) {  
        LOG(CCXA_error, r);  
        r = CCXA_error;  
    }  
    if (r == OK) {  
        r = h(b);  
    }  
    return r;  
}
```

# Context

## Exception handling at ASML

catch error returned

```
int f(int a, int *b) {  
    int r = OK;  
    r = g(a);  
    if (r != OK) {  
        LOG(CCXA_error, r);  
        r = CCXA_error;  
    }  
    if (r == OK) {  
        r = h(b);  
    }  
    return r;  
}
```

# Context

## Exception handling at ASML

```
int f(int a, int *b) {  
    int r = OK;  
    r = g(a);  
    if (r != OK) {  
        LOG(CCXA_error, r);  
        r = CCXA_error;  
    }  
    if (r == OK) {  
        r = h(b);  
    }  
    return r;  
}
```

if not OK, link to  
a new error  
and state context  
info

# Context

## Exception handling at ASML

```
int f(int a, int *b) {  
    int r = OK;  
    r = g(a);  
    if (r != OK) {  
        LOG(CCXA_error, r);  
        r = CCXA_error;  
    }  
    if (r == OK) {  
        r = h(b);  
    }  
    return r;  
}
```

only continue if OK

# Context

## Exception handling at ASML

```
int f(int a, int *b) {  
    int r = OK;  
    r = g(a);  
    if (r != OK) {  
        LOG(CCXA_error, r);  
        r = CCXA_error;  
    }  
    if (r == OK) {  
        r = h(b);  
    }  
    return r; return error  
}
```

# Exception Handling

scattered

- ✓ in  $\pm$  200 components
- ✓ in  $\pm$  15 MLoC

tangled

- ✓ mixed with "ordinary" functionality
- ✓ mixed with other concerns



Our goal ...

# Our goal ...

Improve the handling of crosscutting concerns in order to reduce development time/code size/errors/...

# Our goal ...

Improve the handling of crosscutting concerns in order to reduce development time/code size/errors/...

→ refactor exception handling into an aspect

# Our goal ...

Improve the handling of crosscutting concerns in order to reduce development time/code size/errors/...

- refactor exception handling into an aspect
- in an automatic way (because 15MLoC)

# Our goal ...

Improve the handling of crosscutting concerns in order to reduce development time/code size/errors/...

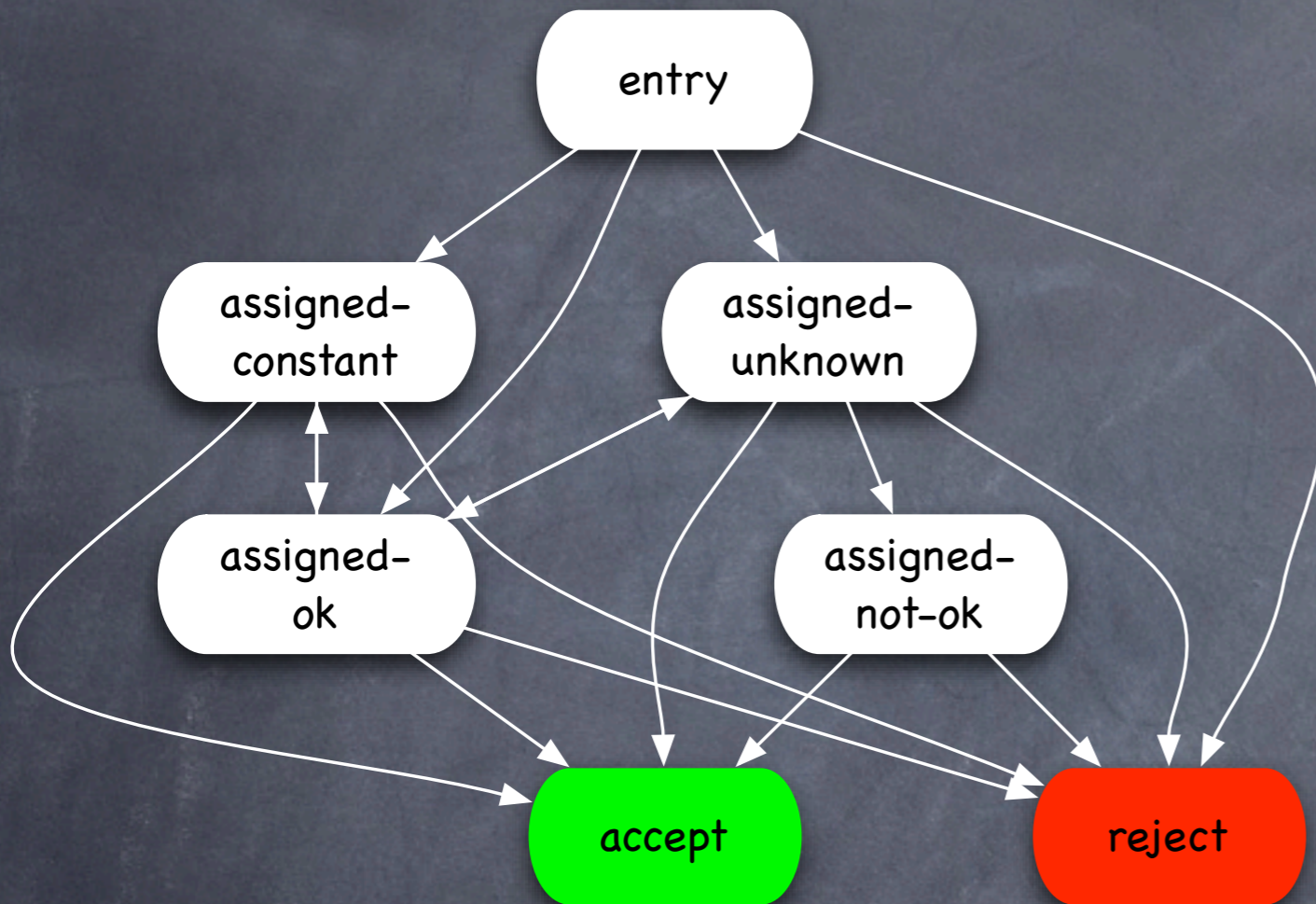
- refactor exception handling into an aspect
- in an automatic way (because 15MLoC)
- but this requires correct code!

# Our goal ...

Improve the handling of crosscutting concerns in order to reduce development time/code size/errors/...

- refactor exception handling into an aspect
- in an automatic way (because 15MLoC)
- but this requires correct code!
- so we wrote a checker ...

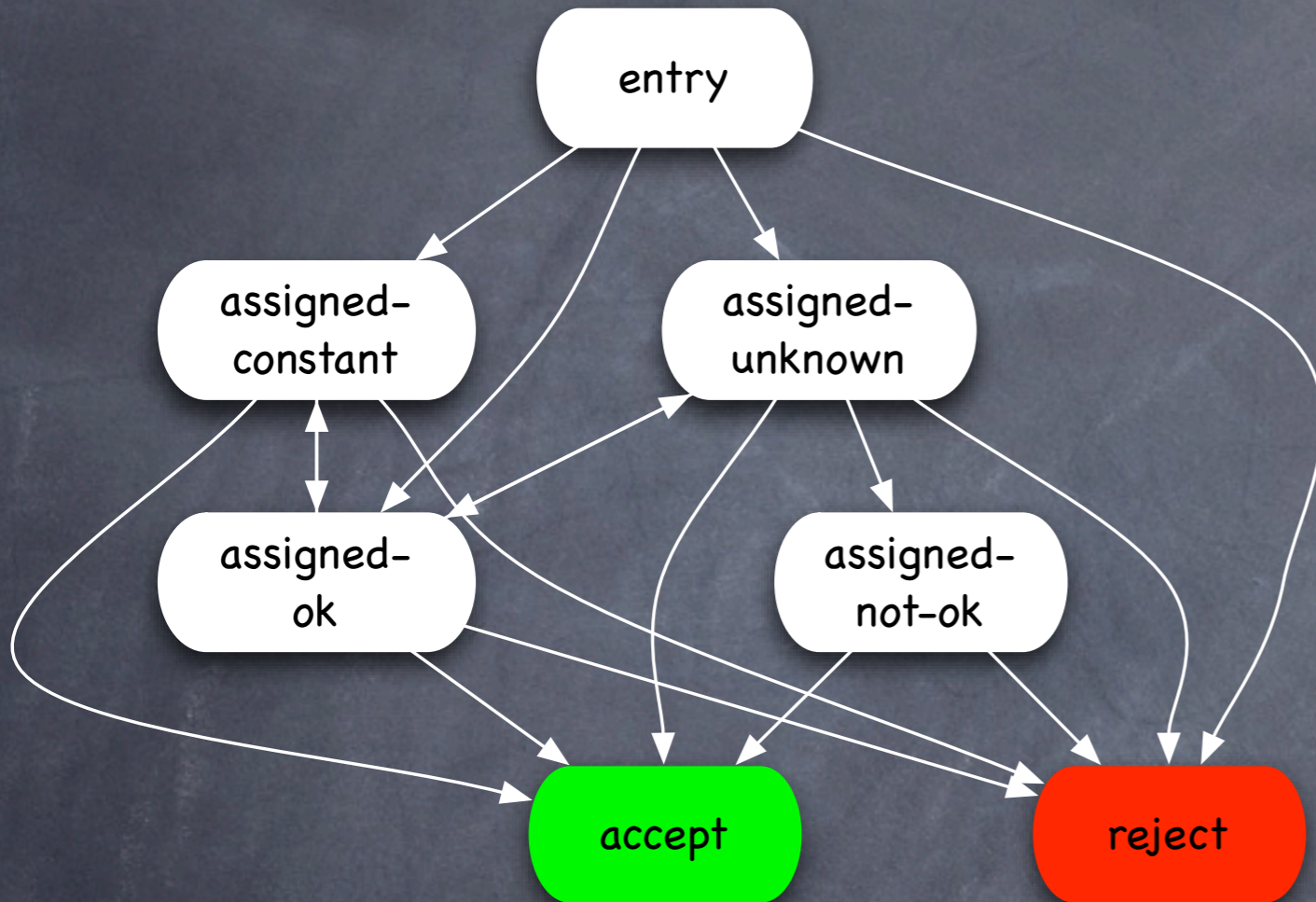
# State Machine for Exception Linking & Logging



Track the value of the error variable by

- writing the exception handling idiom as a DFA
- evaluating the DFA along all (possible) paths of a function

# State Machine for Exception Linking & Logging



Track the value of the error variable by

- writing the exception handling idiom as a DFA
- evaluating the DFA along all (possible) paths of a function

requires

1. error variable identification
2. path identification
3. state machine evaluation

# 1. Error variable identification

```
int f(int a, int *b) {  
    int r = OK;  
    ...  
    if (r != OK)  
    {  
        LOG(CCXA_error, r);  
        r = CCXA_error;  
    }  
    if (r == OK)  
    {  
        r = h(b);  
    }  
    ...  
}
```

An **error variable** is a declaration node, that has type 'int' and is initialised to 'OK'

# 1. Error variable identification

```
int f(int a, int *b) {  
    int r = OK;  
    ...  
    if (r != OK)  
    {  
        LOG(CCXA_error, r);  
        r = CCXA_error;  
    }  
    if (r == OK)  
    {  
        r = h(b);  
    }  
    ...  
}
```

**Killed set** contains only  
constant assignment and/or  
function call assignment  
vertices

# 1. Error variable identification

```
int f(int a, int *b) {  
    int r = OK;  
    ...  
    if (r != OK)  
    {  
        LOG(CCXA_error, r);  
        r = CCXA_error;  
    }  
    if (r == OK)  
    {  
        r = h(b);  
    }  
    ...  
}
```

**Used set** can not contain actual parameters, except for a LOG call

## 2. Path identification

```
int f(int a, int *b) {  
    int r = OK;  
    r = g(a);  
    if (r != OK) {  
        LOG(CCXA_error, r);  
        r = CCXA_error;  
    }  
    if (r == OK) {  
        r = h(b);  
    }  
    return r;  
}
```

## 2. Path identification

```
int f(int a, int *b) {
    int r = OK;
    r = g(a);
    if (r != OK) {
        LOG(CCXA_error, r);
        r = CCXA_error;
    }
    if (r == OK) {
        r = h(b);
    }
    return r;
}
```

```
int f(int a, int *b) {
    int r = OK;
    r = g(a);
    if (r != OK) {
        LOG(CCXA_error, r);
        r = CCXA_error;
    }
    if (r == OK) {
        r = h(b);
    }
    return r;
}
```

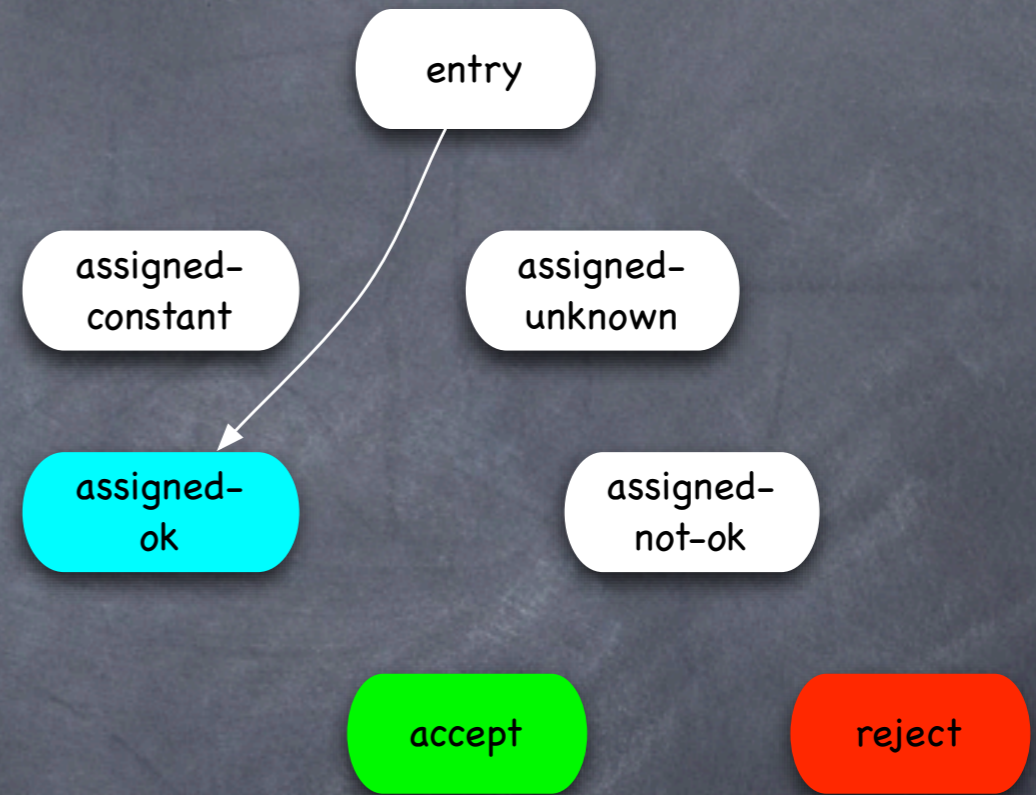
# 3. State machine evaluation

```
int f(int a, int *b) {  
    int r = OK;  
    r = g(a);  
    if (r != OK) {  
        LOG(CCXA_error, r);  
        r = CCXA_error;  
    }  
    if (r == OK) {  
        r = h(b);  
    }  
    return r;  
}
```



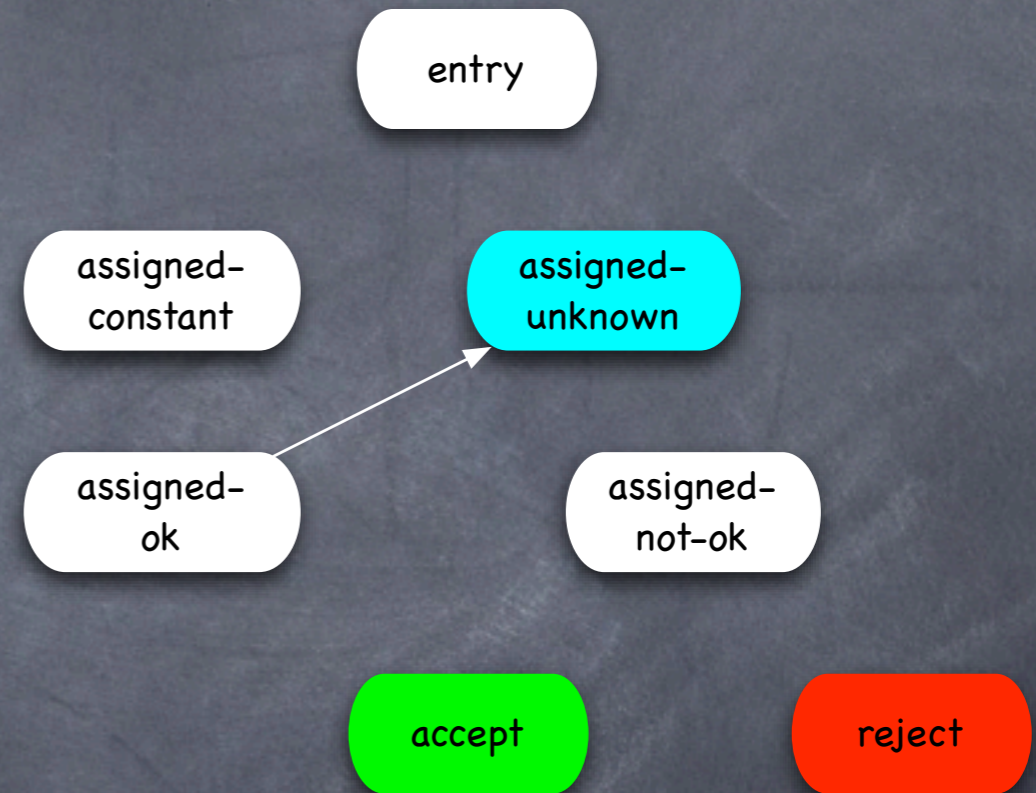
# 3. State machine evaluation

```
int f(int a, int *b) {  
  int r = OK;  
  r = g(a);  
  if (r != OK) {  
    LOG(CCXA_error, r);  
    r = CCXA_error;  
  }  
  if (r == OK) {  
    r = h(b);  
  }  
  return r;  
}
```



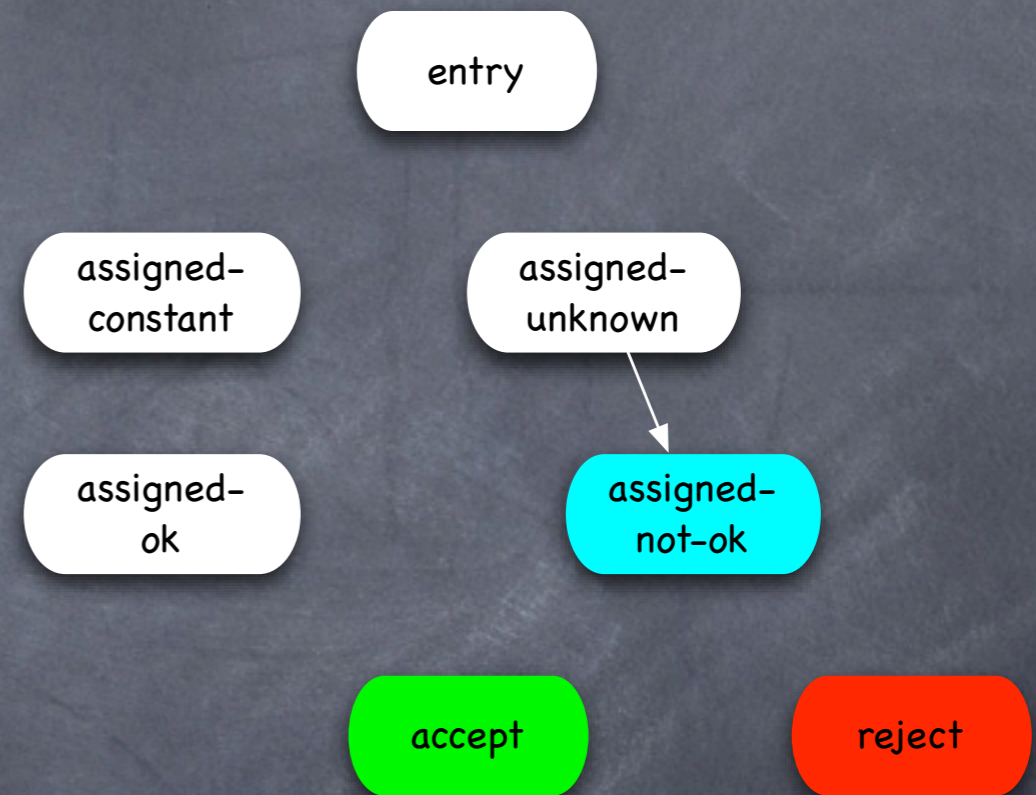
# 3. State machine evaluation

```
int f(int a, int *b) {  
  int r = OK;  
  r = g(a);  
  if (r != OK) {  
    LOG(CCXA_error, r);  
    r = CCXA_error;  
  }  
  if (r == OK) {  
    r = h(b);  
  }  
  return r;  
}
```



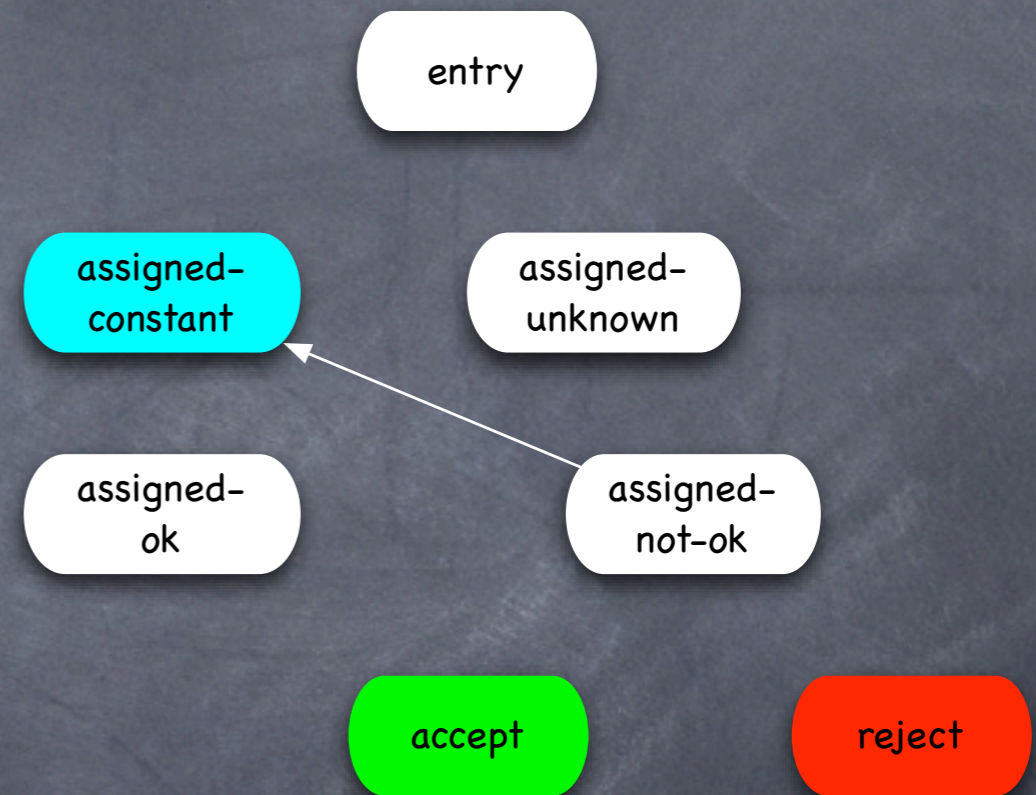
# 3. State machine evaluation

```
int f(int a, int *b) {  
  int r = OK;  
  r = g(a);  
  if (r != OK) {  
    LOG(CCXA_error, r);  
    r = CCXA_error;  
  }  
  if (r == OK) {  
    r = h(b);  
  }  
  return r;  
}
```



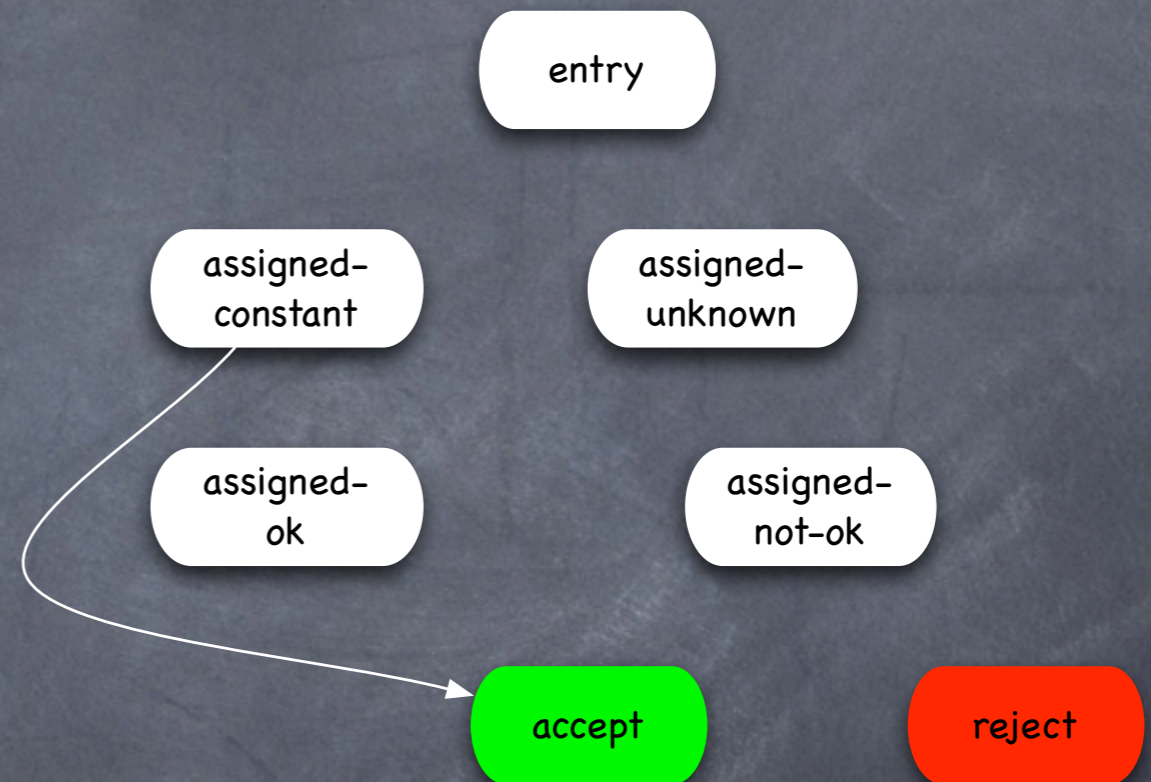
# 3. State machine evaluation

```
int f(int a, int *b) {  
  int r = OK;  
  r = g(a);  
  if (r != OK) {  
    LOG(CCXA_error, r);  
    r = CCXA_error;  
  }  
  if (r == OK) {  
    r = h(b);  
  }  
  return r;  
}
```



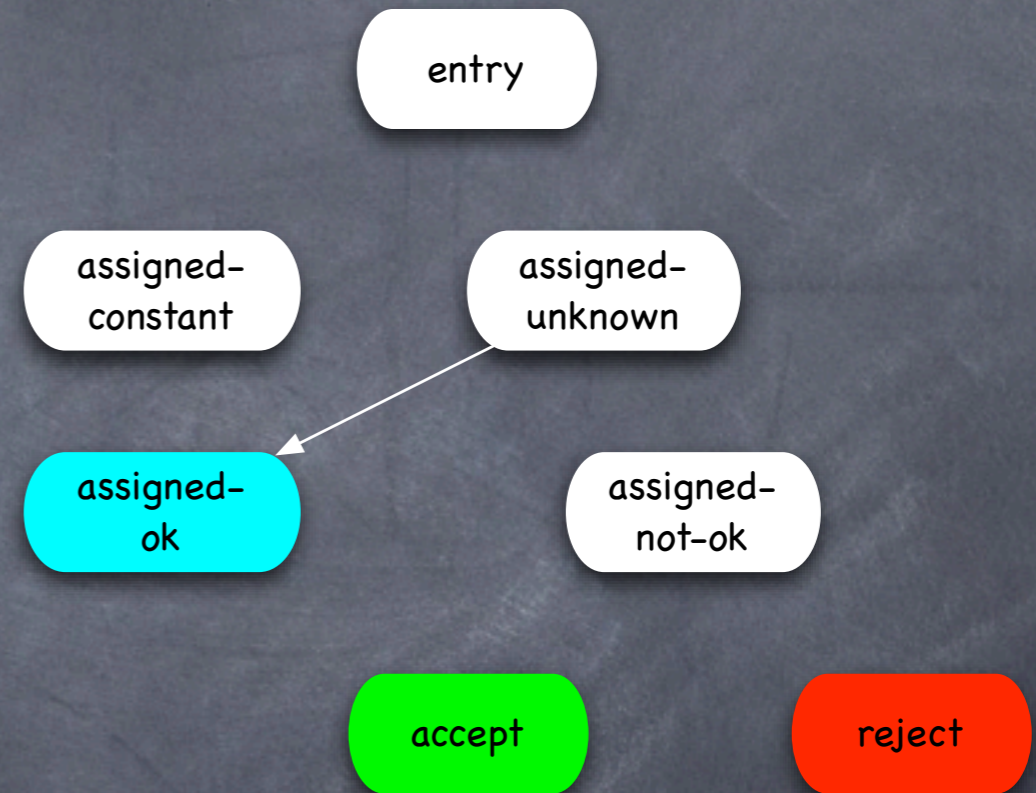
# 3. State machine evaluation

```
int f(int a, int *b) {  
  int r = OK;  
  r = g(a);  
  if (r != OK) {  
    LOG(CCXA_error, r);  
    r = CCXA_error;  
  }  
  if (r == OK) {  
    r = h(b);  
  }  
  return r;  
}
```



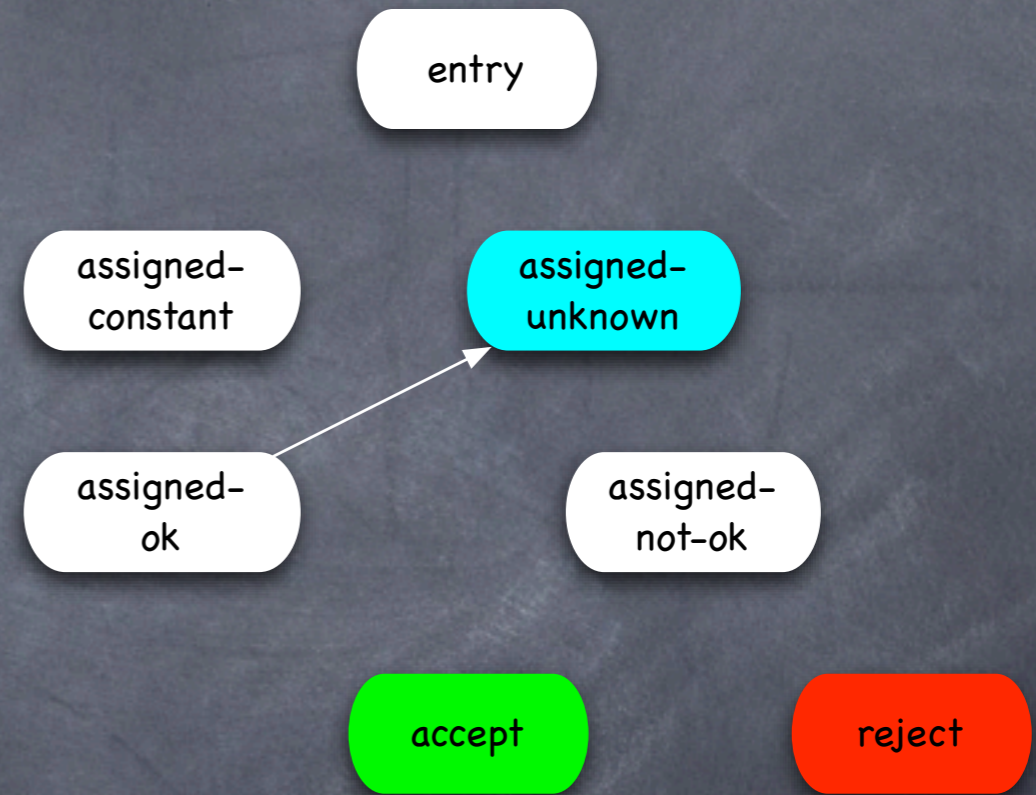
# 3. State machine evaluation

```
int f(int a, int *b) {  
  int r = OK;  
  r = g(a);  
  if (r != OK) {  
    LOG(CCXA_error, r);  
    r = CCXA_error;  
  }  
  if (r == OK) {  
    r = h(b);  
  }  
  return r;  
}
```



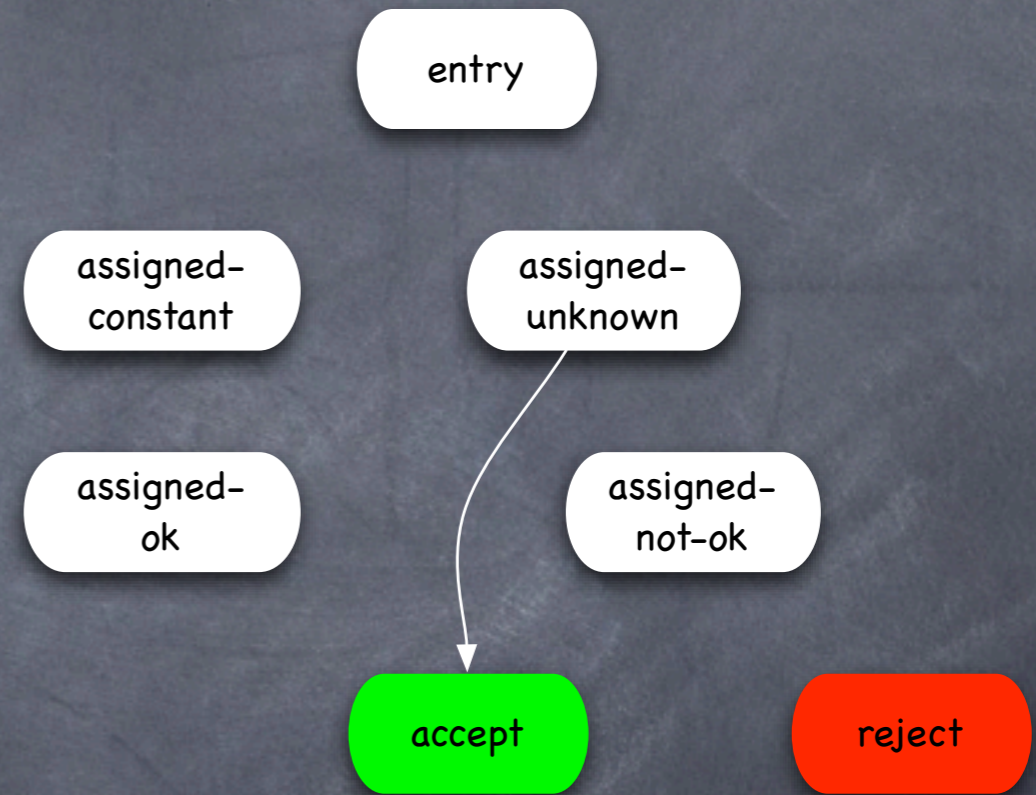
# 3. State machine evaluation

```
int f(int a, int *b) {  
  int r = OK;  
  r = g(a);  
  if (r != OK) {  
    LOG(CCXA_error, r);  
    r = CCXA_error;  
  }  
  if (r == OK) {  
    r = h(b);  
  }  
  return r;  
}
```



# 3. State machine evaluation

```
int f(int a, int *b) {  
    int r = OK;  
    r = g(a);  
    if (r != OK) {  
        LOG(CCXA_error, r);  
        r = CCXA_error;  
    }  
    if (r == OK) {  
        r = h(b);  
    }  
    return r;  
}
```



# Experimental results

	Reported deviations	False positives	unintended deviations	kLoC
CC1	138	27	111	15
CC2	38	15	23	21
CC3	11	7	4	15
CC4	22	5	17	4
CC5	74	14	60	14.5
CC6	17	4	13	5
Total	300	72	228	74.5

# Experimental results

only 66 found by  
manual inspection

	Reported deviations	False positives	unintended deviations	kLoC
CC1	138	27	111	15
CC2	38	15	23	21
CC3	11	7	4	15
CC4	22	5	17	4
CC5	74	14	60	14.5
CC6	17	4	13	5
Total	300	72	228	74.5

# Experimental results

only 66 found by  
manual inspection

	Reported deviations	False positives	unintended deviations	kLoC
CC1	138	27	111	15
CC2	38	15	23	21
CC3	11	7	4	15
CC4	22	5	17	4
CC5	74	14	60	14.5
CC6	17	4	13	5
Total	300	72	228	74.5

76% unintended  
deviations

# Experimental results

only 66 found by  
manual inspection

	Reported deviations	False positives	unintended deviations	kLoC
CC1	138	27	111	15
CC2	38	15	23	21
CC3	11	7	4	15
CC4	22	5	17	4
CC5	74	14	60	14.5
CC6	17	4	13	5
Total	300	72	228	74.5

76% unintended  
deviations

→  $\pm 3$  deviations/kLoc

# Our goal ...

Improve the handling of crosscutting concerns in order to reduce development time/code size/errors/...

- refactor exception handling into an aspect
- in an automatic way (because 15MLoC)
- but this requires correct code!
- so we wrote a checker ...

# Our goal ...

Improve the handling of crosscutting concerns in order to reduce development time/code size/errors/...

- refactor exception handling into an aspect
- in an automatic way (because 15MLoC)

# Typical faults

unallowed assignments

```
int g(int a) {  
    int r = OK;  
    ...  
    if (r != OK) {  
        LOG(some_error, r);  
        r = some_error;  
    }  
    ...  
    if (some_condition) {  
        r = h(b);  
    }  
    ...  
}
```

logged value != assigned value

```
int h(int *b) {  
    int r = OK;  
  
    if (r != OK) {  
        r = some_error;  
    }  
    ...  
    return r;  
}
```

# Typical faults

unallowed assignments

```
int g(int a) {  
    int r = OK;  
    ...  
    if (r != OK) {  
        LOG(some_error, r);  
        r = some_error;  
    }  
    ...  
    if (some_condition) {  
        r = h(b);  
    }  
    ...  
}
```

mostly due to forgotten  
guard when assigning to  
"r"

logged value != assigned value

```
int h(int *b) {  
    int r = OK;  
  
    if (r != OK) {  
        r = some_error;  
    }  
    ...  
    return r;  
}
```

mostly due to forgotten  
LOG call when assigning  
an error to "r"

# Introducing macro's

automatic logging when assigning

```
#define ROOT_LOG(error_value, error_var)\
    error_var = error_value;\
    LOG(error_value, OK);
```

```
#define LINK_LOG(function_call, error_value, error_var)\
    if(error_var == OK) {\
        int _internal_error_var = function_call;\
        if(_internal_error_var != OK) {\
            LOG(error_value, _internal_error_var);\
            error_var = error_value;\
        }\
    }
```

```
#define NO_LOG(function_call, error_var)\
    if(error_var == OK) \
        error_var = function_call;
```

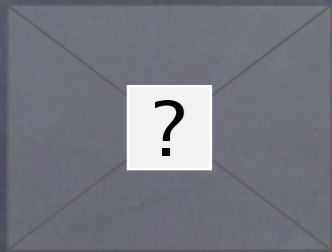
# Introducing macro's

automatic guarding when assigning

```
#define ROOT_LOG(error_value, error_var)\
    error_var = error_value;\
    LOG(error_value, OK);
```

```
#define LINK_LOG(function_call, error_value, error_var)\
    if(error_var == OK) {\
        int _internal_error_var = function_call;\
        if(_internal_error_var != OK) {\
            LOG(error_value, _internal_error_var);\
            error_var = error_value;\
        }\
    }
```

```
#define NO_LOG(function_call, error_var)\
    if(error_var == OK) \
        error_var = function_call;
```



# After migration

```
int f(int a, int *b) {  
    int r = OK;  
    r = g(a);  
    if (r != OK) {  
        LOG(CCXA_error, r);  
        r = CCXA_error;  
    }  
    if (r == OK) {  
        r = h(b);  
    }  
    return r;  
}
```



```
int f(int a, int *b) {  
    int r = OK;  
    LINK_LOG(g(a), CCXA_error, r);  
    NO_LOG(h(b), r);  
    return r;  
}
```



# One step further ...

```
int f(int a, int *b) {  
    int r = OK;  
    LINK_LOG(g(a), CCXA_error, r);  
    NO_LOG(h(b), r);  
    return r;  
}
```



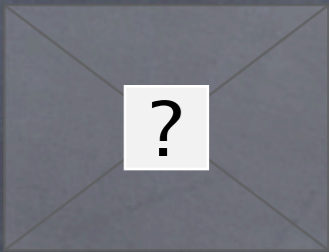
```
int f(int a, int *b) {  
    int r = OK;  
    g(a);  
    h(b);  
    return r;  
}
```

```
pointcut g-calls() :  
    call(g(int));  
around g-calls() {  
    LINK_LOG(ceed(), CCXA_error, r);  
}
```



```
pointcut h-calls() :  
    call(h(int *));  
around h-calls() {  
    NO_LOG(ceed(), r);  
}
```





# One step further ...

```
int f(int a, int *b) {  
    int r = OK;  
    LINK_LOG(g(a), CCXA_error, r);  
    NO_LOG(h(b), r);  
    return r;  
}
```



```
int f(int a, int *b) {  
    int r = OK;  
    g(a);  
    h(b);  
    return r;  
}
```

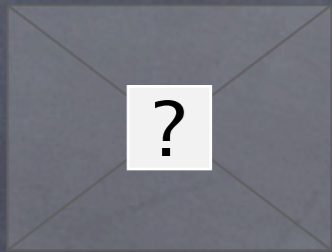
```
pointcut g-calls() :  
    call(g(int));
```

```
around g-calls() {  
    LINK_LOG(ceed(), CCXA_error, r);  
}
```

```
pointcut h-calls() :  
    call(h(int *));
```

```
around h-calls() {  
    NO_LOG(ceed(), r);  
}
```

does not hold for all g and h calls  
does only hold inside f!



# One step further ...

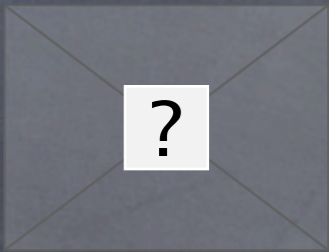
```
int f(int a, int *b) {  
    int r = OK;  
    LINK_LOG(g(a), CCXA_error, r);  
    NO_LOG(h(b), r);  
    return r;  
}
```



```
int f(int a, int *b) {  
    int r = OK;  
    g(a);  
    h(b);  
    return r;  
}
```

```
pointcut g-calls() :  
    call(g(int));  
around g-calls() {  
    LINK_LOG(ceed(), CCXA_error, r);  
}
```

```
pointcut h-calls() :  
    call(h(int *));  
around h-calls() {  
    NO_LOG(ceed(), r);  
}
```



# One step further ...

```
int f(int a, int *b) {  
    int r = OK;  
    LINK_LOG(g(a), CCXA_error, r);  
    NO_LOG(h(b), r);  
    return r;  
}
```



```
int f(int a, int *b) {  
    int r = OK;  
    g(a);  
    h(b);  
    return r;  
}
```

```
pointcut g-calls() :  
    call(g(int));  
around g-calls() {  
    LINK_LOG(ceed(), CCXA_error, r);  
}
```

```
pointcut h-calls() :  
    call(h(int *));  
around h-calls() {  
    NO_LOG(ceed(), r);  
}
```

→ difficult to abstract, we will end up  
with enumeration-based pointcuts

# What you should remember ...

- ✓ We use advanced program analysis techniques to verify/migrate complex idioms
- ✓ Our techniques are applicable to large-scale industrial applications, written in C
- ✓ We are always interested in applying these techniques to other idioms
- ✓ We know where we are, we don't know where we are going ...

# Thought-provoking questions

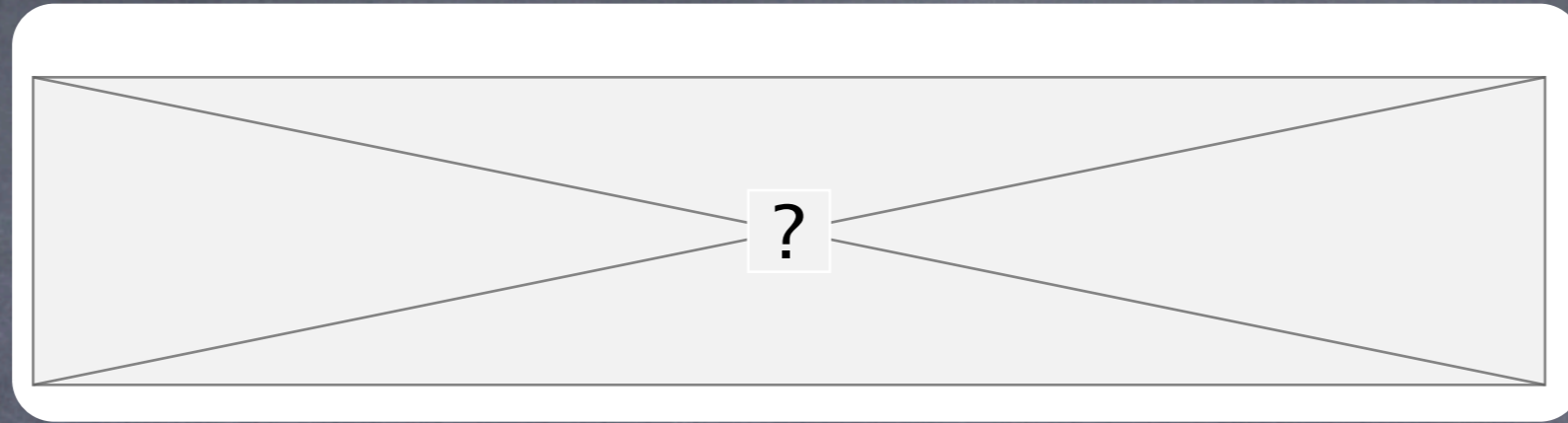
- ✓ What additional benefits/drawbacks would using AOSD bring?
- ✓ What is and what isn't an aspect/crosscutting concern?
- ✓ What can we learn from this experiment about current aspect language technology?
- ✓ Are there other concerns that exhibit the same behaviour?

# Do you want to know the details?

<http://www.cwi.nl/~tourwe>

- ✓ Magiel Bruntink, Arie van Deursen, Remco van Engelen, and Tom Tourwé, [On the Use of Clone Detection for Identifying Crosscutting Concern code](#), in IEEE Transactions on Software Engineering (TSE), 31(10), October, 2005.
- ✓ Magiel Bruntink, Arie van Deursen, and Tom Tourwé, [Isolating Idiomatic Crosscutting Concerns](#), in Proceedings of the International Conference on Software Maintenance (ICSM), September, 2005.
- ✓ Magiel Bruntink, Arie van Deursen, and Tom Tourwé, [Discovering Faults in Idiom-Based Exception Handling](#), submitted to ICSE 2006.

# Do you want to participate?



Linking **A**spect **T**echnology and **E**volution workshop

held in conjunction with the  
5th Aspect-Oriented Software Development Conference,  
Bonn, Germany, March 20-24, 2006

“Submit to LATE, but not too late!”