CSEP504: Advanced topics in software systems

- Tonight: 2nd of three lectures on software tools and environments – a few tools in some more depth
- February 22 (Reid Holmes): Future directions and areas for improvement – rationale behind the drive towards integration
  - Capturing latent knowledge
  - Task specificity / awareness
  - Supporting collaborative development
- The plan for the final two lectures

David Notkin ● Winter 2010 ● CSEP504 Lecture 5

Announcements

- The second state-of-the-research paper can be on *any approved topic* in software engineering research
  - That is, it needn't be focused on one of the core topics in the course
  - Everything else stays the same (due dates, groups, commenting, etc.)
- Comment away on the first state-of-the-research papers!

Announcements

- March 1:
  - Report from India (Microsoft Research, discussions about starting a software engineering center, etc.) [~30 minutes]
  - Different ways to evaluate and assess software engineering research [~60-90 minutes]
- March 8: SE economics (I will post readings soon)

Languages and tools

- In preparing for this lecture, one possible topic Reid and I discussed was “languages as tools”
  - The premise is that different programming languages support different development methodologies and have particular strengths
  - Another lightly related question is how to decide between placing something in a language or in a tool: as an example, consider lint vs. types
- But no deep discussion tonight

Tonight

- Concolic testing – in depth
- Continuous testing – not in depth
- Carving from system tests – even less in depth
- Speculation – discussion about the idea
- LSDiff – in depth
- Reflexion models – in some depth

Testing

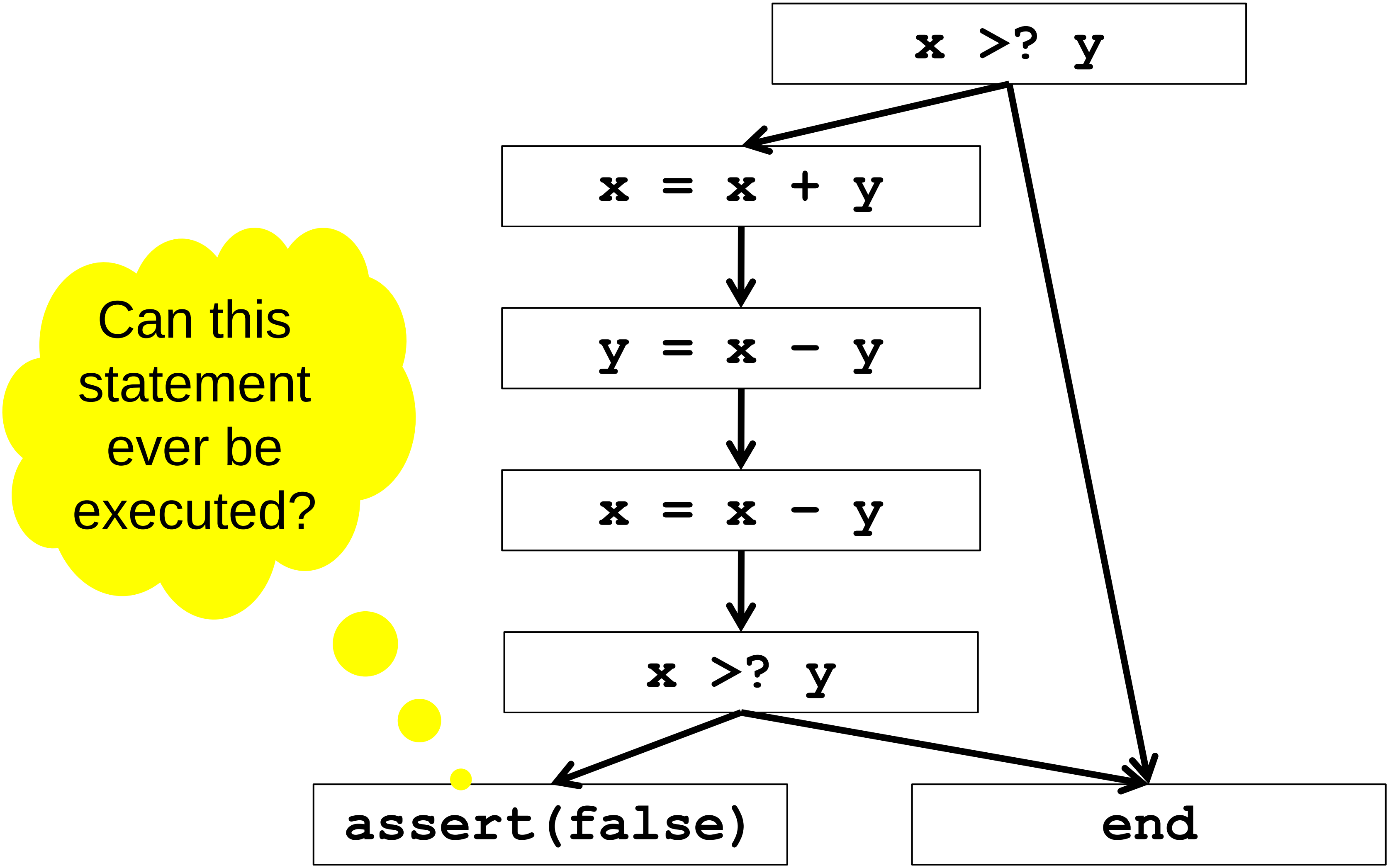
Not full-fledged testing lectures!

- What questions should testing – broadly construed – answer about this itsy-bitsy program?
- What criteria should we use to assess different approaches to testing it?

```
if (x > y) {
  x = x + y;
  y = x - y;
  x = x - y;
  if (x > y)
    assert(false)
}
```

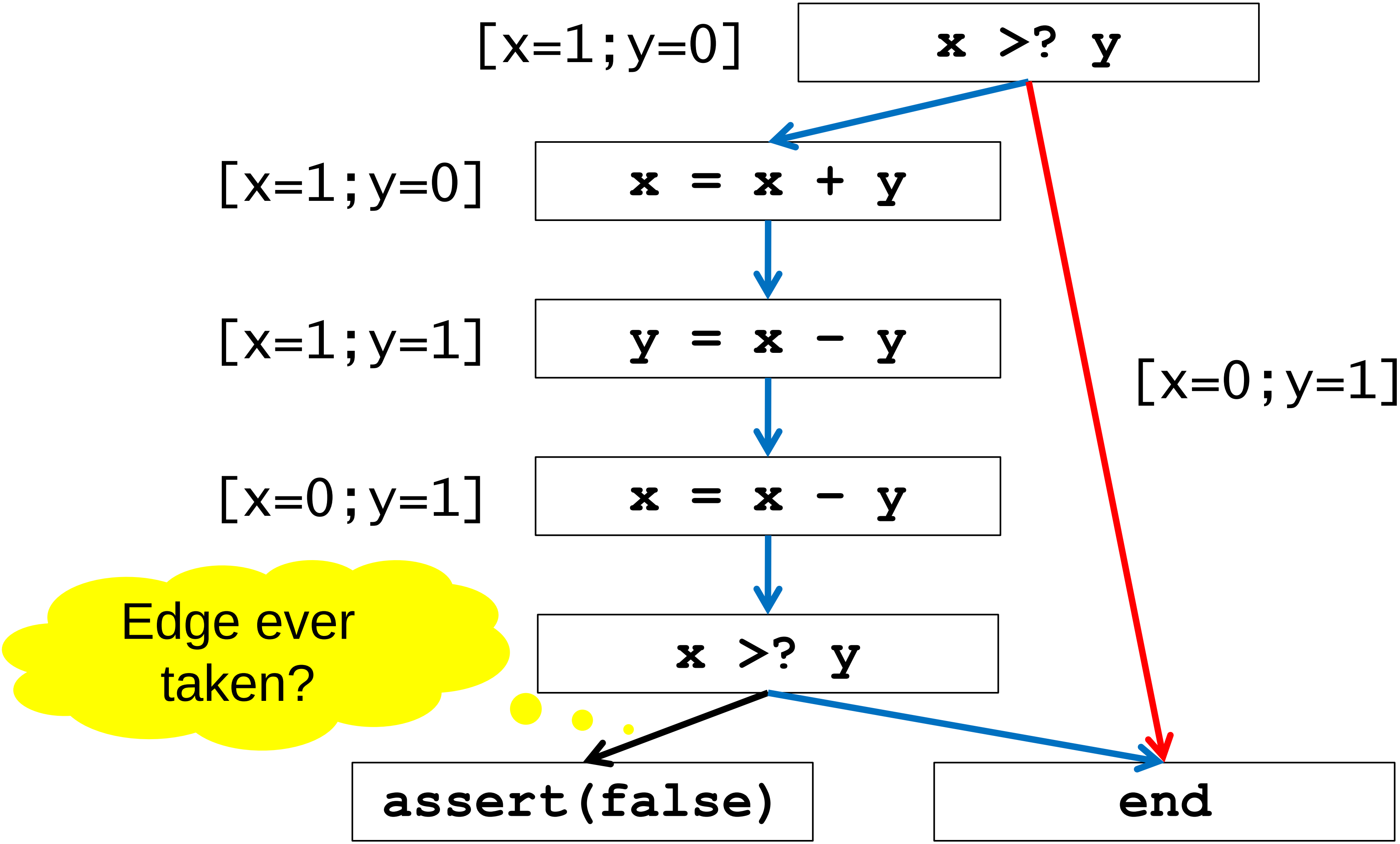
Example from
Visser, Pasareanu & Mehlitz

Control flow graph (CFG)

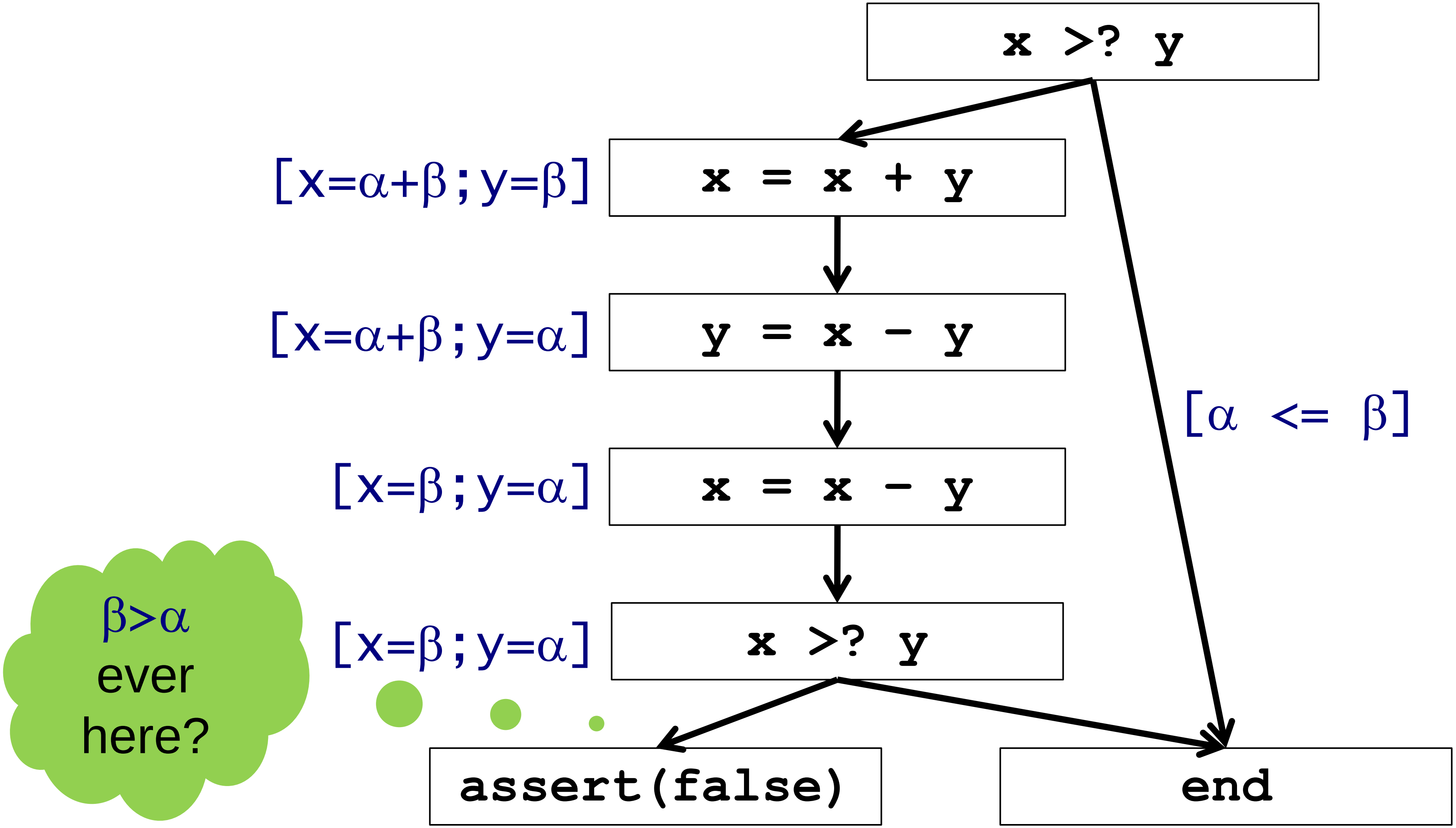


Can this statement ever be executed?

Edge coverage



Symbolic execution [x=α;y=β]



`x >? y`

[x=α+β;y=β] `x = x + y`

[x=α+β;y=α] `y = x - y`

[x=β;y=α] `x = x - y`

[x=β;y=α] `x >? y`

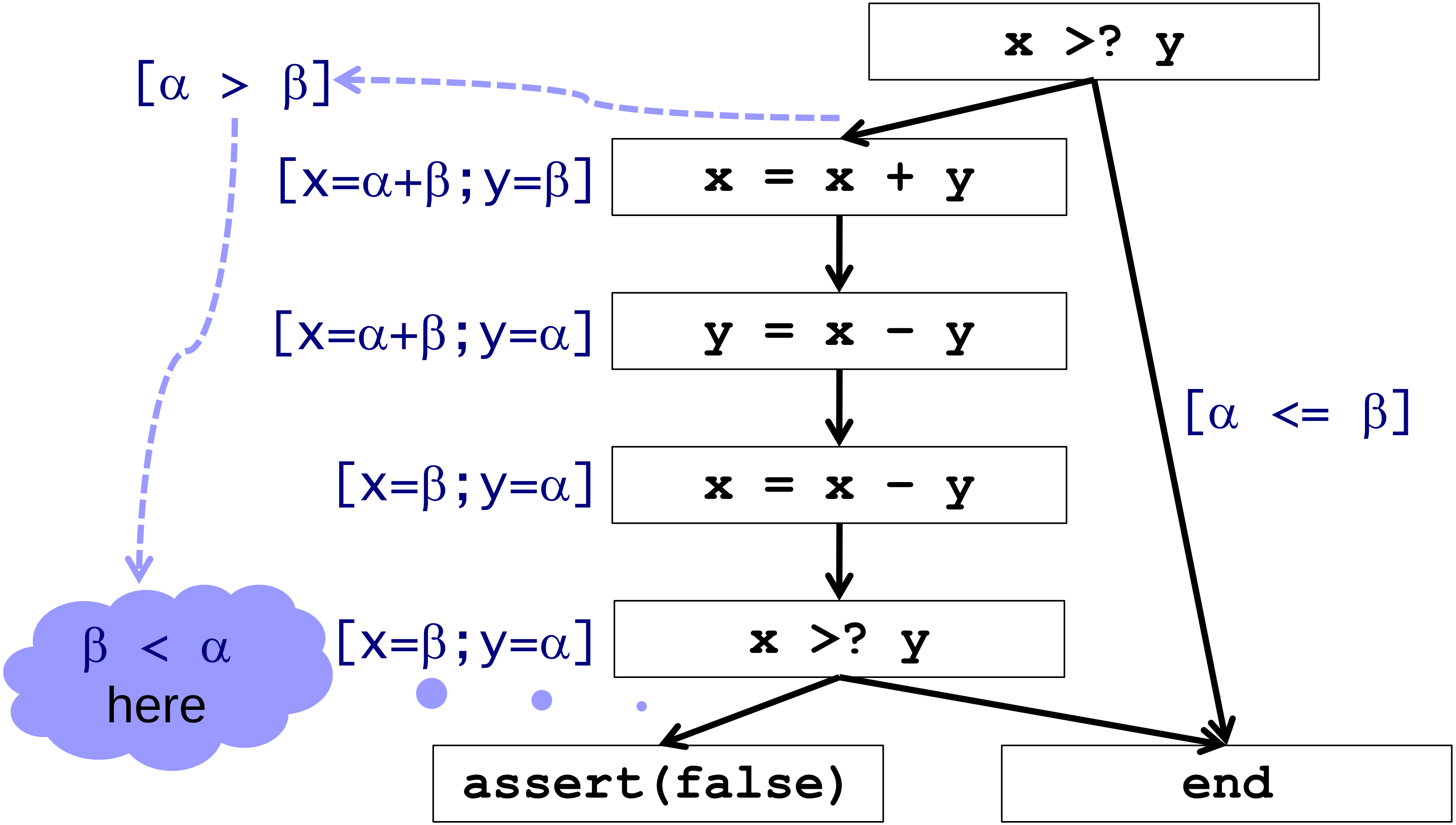
[α <= β]

`assert(false)`

`end`

β>α ever here?

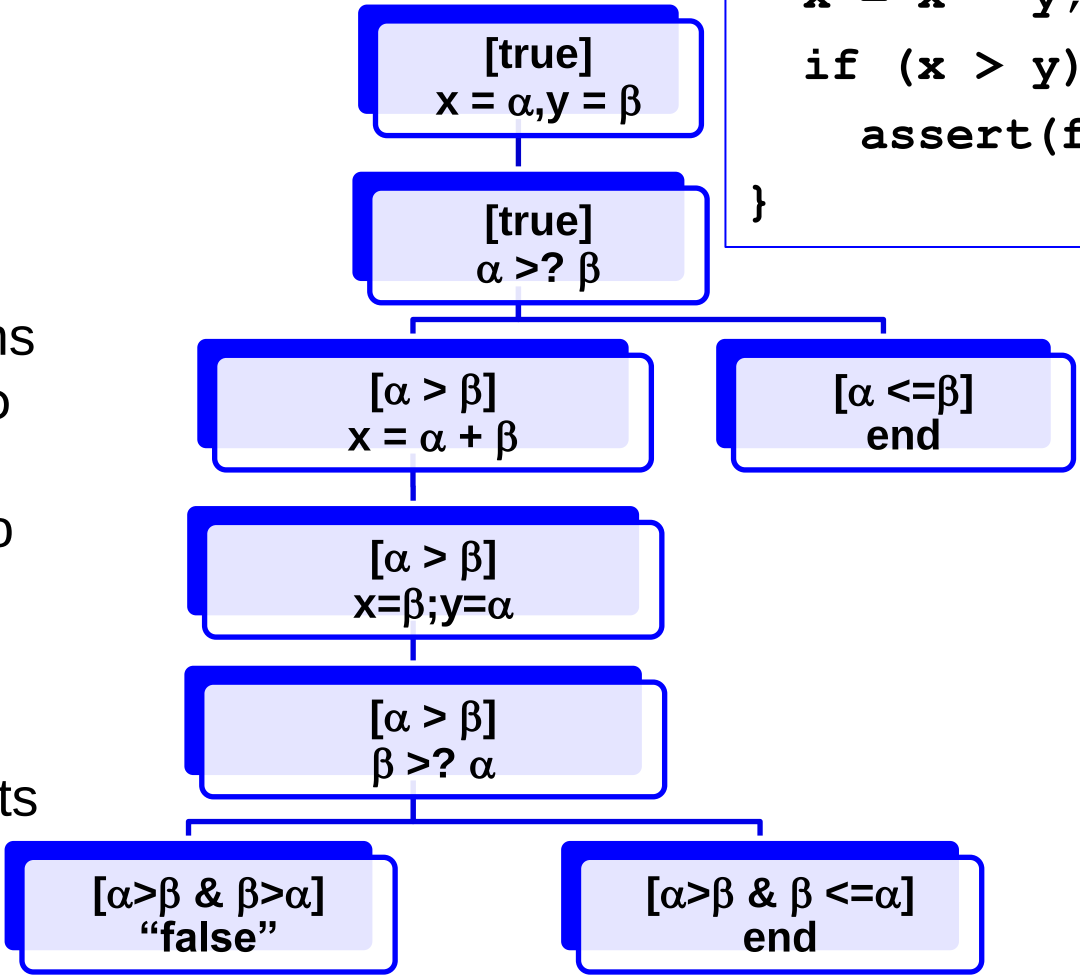
Symbolic execution



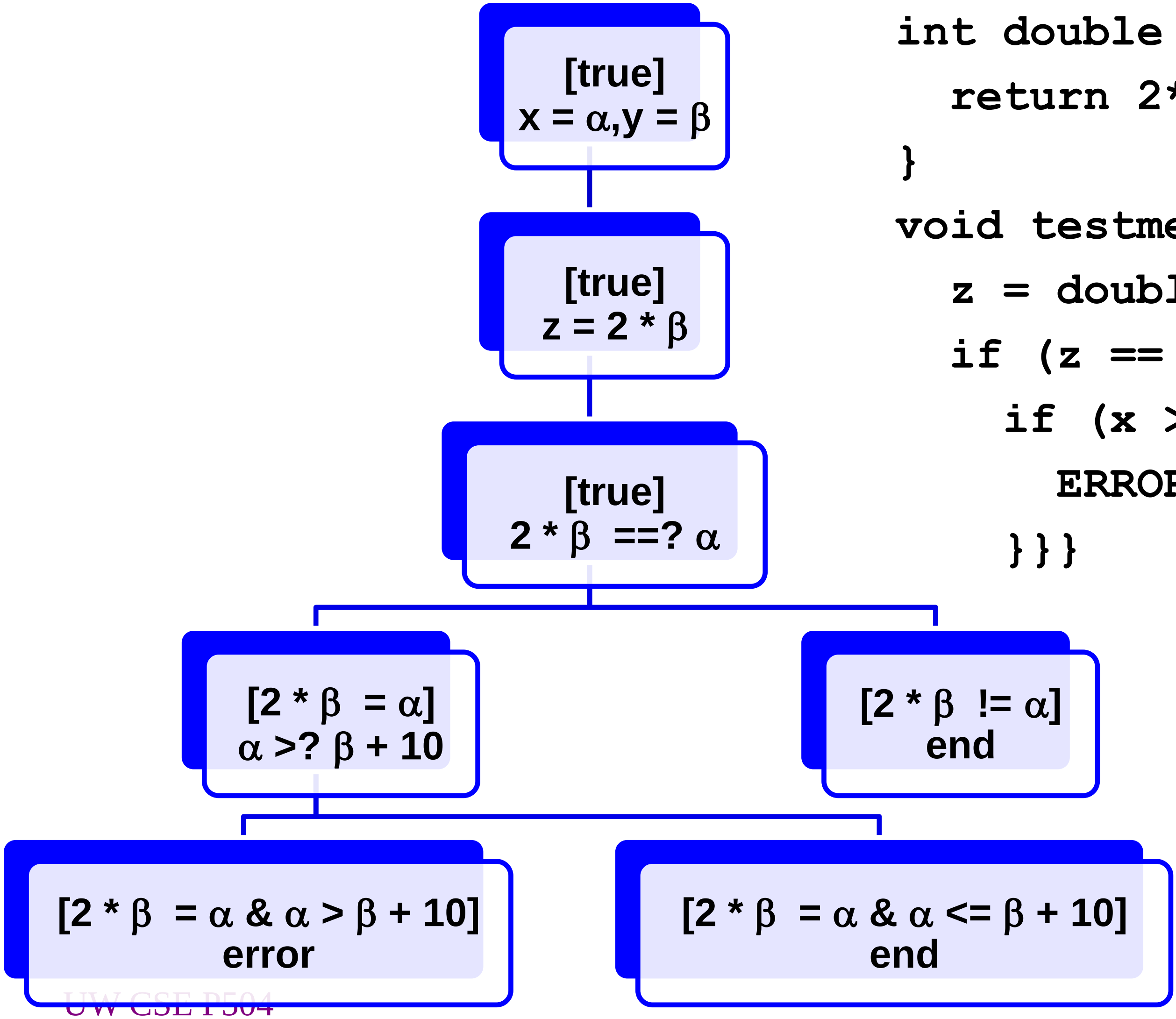
What’s really going on?

```
if (x > y) {
  x = x + y;
  y = x – y;
  x = x – y;
  if (x > y)
    assert(false)
}
```

- Create a symbolic execution tree
- Explicitly track path conditions
- Solve path conditions – “how do you get to this point in the execution tree?” – to defines test inputs
- Goal: define test inputs that reach all reachable statements



Another example (Sen and Agha)



```
int double (int v){
   return 2*v;
}
void testme (int x, int y){
   z = double (y);
   if (z == x) {
      if (x > y+10) {
         ERROR;
      }}}
```

Error: possible by solving equations

$[2 * \beta = \alpha \,\&\, \alpha > \beta + 10]$

$\equiv [2 * \beta > \beta + 10]$

$\equiv [\beta > 10]$

$\equiv [\beta > 10 \,\&\, 2 * \beta = \alpha\,]$

Way cool – we're done!

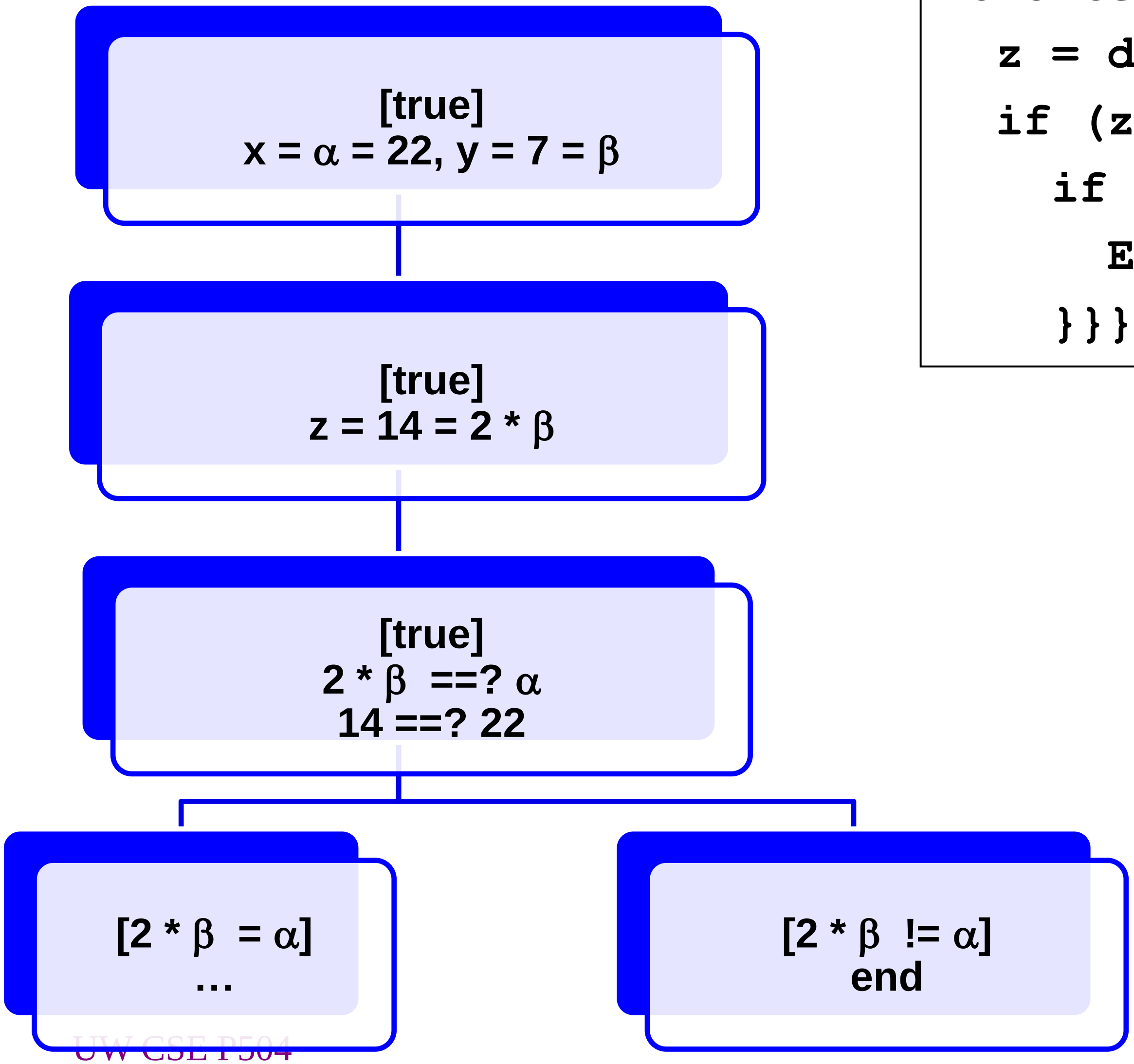
- First example can't reach **`assert(false)`**, and it's easy to reach **`end`** via both possible paths
- Second example: can reach **`error`** and **`end`** via both possible paths
- Well, what if we can't solve the path conditions?
  - Some arithmetic, some recursion, some loops, some pointer expressions, etc.
  - We'll see an example
- What if we want specific test cases?

Concolic testing: Sen et al.

- Basically, combine concrete and symbolic execution
- More precisely…
  - Generate a random concrete input
  - Execute the program on that input both concretely and symbolically simultaneously
  - Follow the concrete execution and maintain the path conditions along with the corresponding symbolic execution
  - Use the path conditions collected by this guided process to constrain the generation of inputs for the next iteration
  - Repeat until test inputs are produced to exercise all feasible paths

2nd example redux
1st iteration x=22, y=7

```
int double (int v){
  return 2*v;
}
void testme (int x, int y){
  z = double (y);
  if (z == x) {
    if (x > y+10) {
      ERROR;
    }}}
```



[true]
$x = \alpha = 22$, $y = 7 = \beta$

[true]
$z = 14 = 2 * \beta$

[true]
$2 * \beta$ ==? α
14 ==? 22

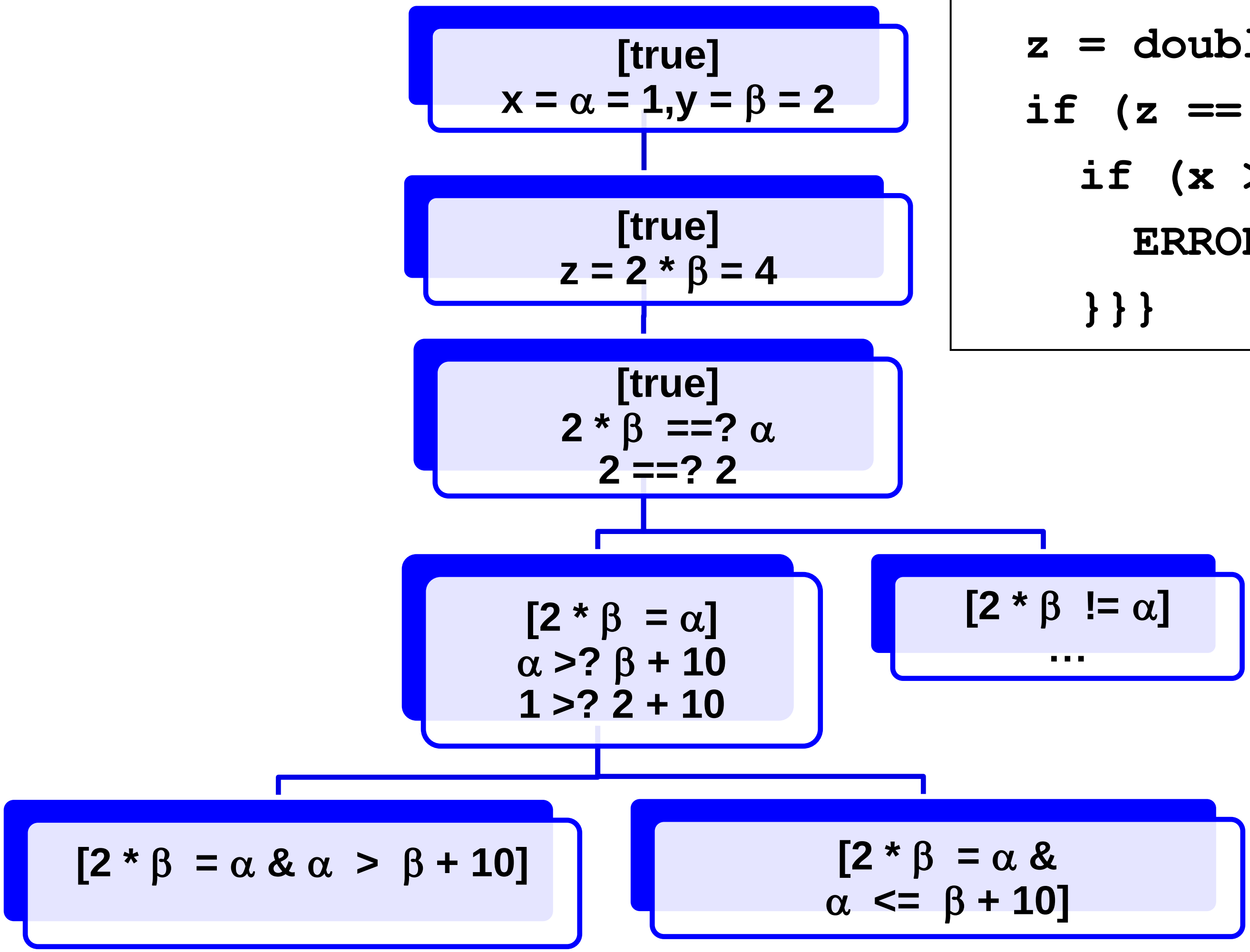
[$2 * \beta = \alpha$]
…

[$2 * \beta$!= α]
end

- Now solve $2 * \beta = \alpha$ to force the other branch
- `x = 1; y = 2` is one solution

2nd example
2nd iteration x=1, y=2

```
int double (int v){
  return 2*v;
}
void testme (int x, int y){
  z = double (y);
  if (z == x) {
    if (x > y+10) {
      ERROR;
    }}}
```



[true]
$x = \alpha = 1, y = \beta = 2$

[true]
$z = 2 * \beta = 4$

[true]
$2 * \beta$ ==? α
2 ==? 2

[$2 * \beta = \alpha$]
α >? $\beta + 10$
1 >? 2 + 10

[$2 * \beta$!= α]
…

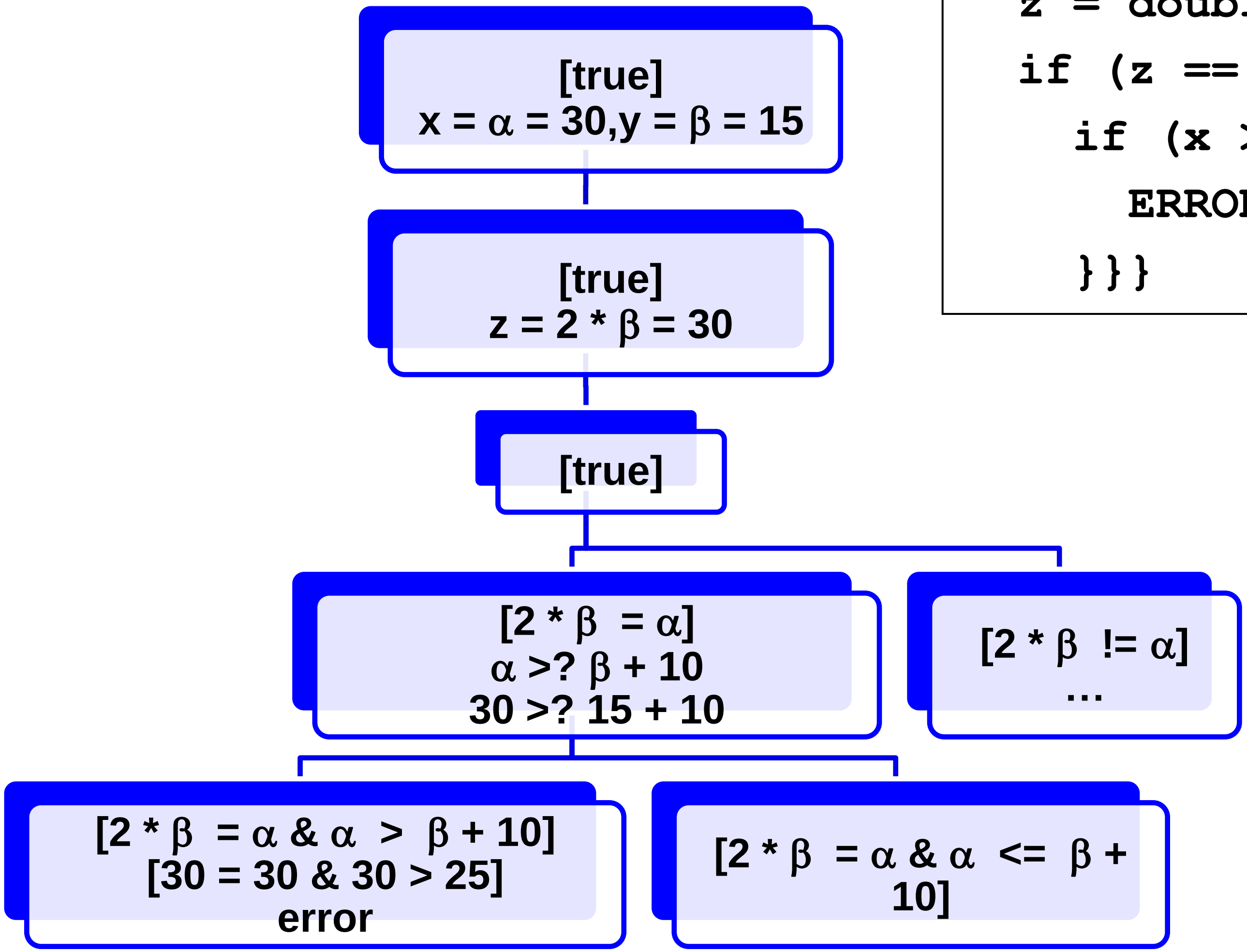
[$2 * \beta = \alpha$ & $\alpha > \beta + 10$]

[$2 * \beta = \alpha$ &
$\alpha <= \beta + 10$]

- **Now solve $2 * \beta = \alpha$ & $\alpha <= \beta + 10$ to force the other branch**
- **`x = 30; y = 15` is one solution**

2nd example
3nd iteration x=30, y=15

```
int double (int v){
  return 2*v;
}
void testme (int x, int y){
  z = double (y);
  if (z == x) {
    if (x > y+10) {
      ERROR;
    }}}
```



[true]
x = α = 30,y = β = 15

[true]
z = 2 * β = 30

[true]

[2 * β = α]
α >? β + 10
30 >? 15 + 10

[2 * β != α]
…

[2 * β = α & α > β + 10]
[30 = 30 & 30 > 25]
error

[2 * β = α & α <= β + 10]

- **Now solve $2 * \beta = \alpha$ & $\alpha <= \beta + 10$ to force the other branch**
- **`x = 30; y = 15` is one solution**

Three concrete test cases

```
int double (int v){ return 2*v;}
void testme (int x, int y){
  z = double (y);
  if (z == x) {
    if (x > y+10) {
      ERROR;
    }
  }
}
```

x	y	
22	7	Takes first else
2	1	Takes first then and second else
30	15	Takes first and second then

Concolic testing example: P. Sağlam

- Random seed
  - **x = -3; y = 7**
- Concrete
  - **z = 9**
- Symbolic
  - $z = x^3+3x^2+9$
- Take **then** branch with constraint $x^3+3x^2+9 \; != \; y$

```
void test_me(int x,int y){
    z = x*x*x + 3*x*x + 9;
    if(z != y){
        printf("Good branch");
    } else {
        printf("Bad branch");
        abort();
    }
}
```

- Take **else** branch with constraint $x^3+3x^2+9 = y$

Concolic testing example: P. Sağlam

- Solving is hard for
  $x^3+3x^2+9 = y$
- So use **z**'s concrete value, which is currently **9**, and continue concretely
- **9 != 7** so **then** is **good**
- Symbolically solve **9 = y** for **else** clause
- Execute next run with
  x = -3; y = 9
  so **else** is **bad**

```
void test_me(int x,int y){
   z = x*x*x + 3*x*x + 9;
   if(z != y){
        printf("Good branch");
   } else {
        printf("Bad branch");
        abort();
   }
}
```

- When symbolic expression becomes unmanageable (e.g., non-linear) replace it by concrete value

Concolic testing example: P. Sağlam

- Random
  - Random memory graph reachable from **`p`**
  - Random value for **`x`**
  - Probability of reaching **`abort( )`** is extremely low
- (Why is this a somewhat misleading motivation?)

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

Let’s try it

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

Concrete	Symbolic	Constraints

p=NULL;
x=236		

Let’s try it

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

Concrete

Symbolic

Constraints

```
p=[634,NULL];
x=236
```

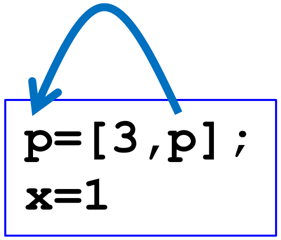
Let’s try it

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

Concrete

Symbolic

Constraints



```
p=[3,p];
x=1
```

Let’s try it

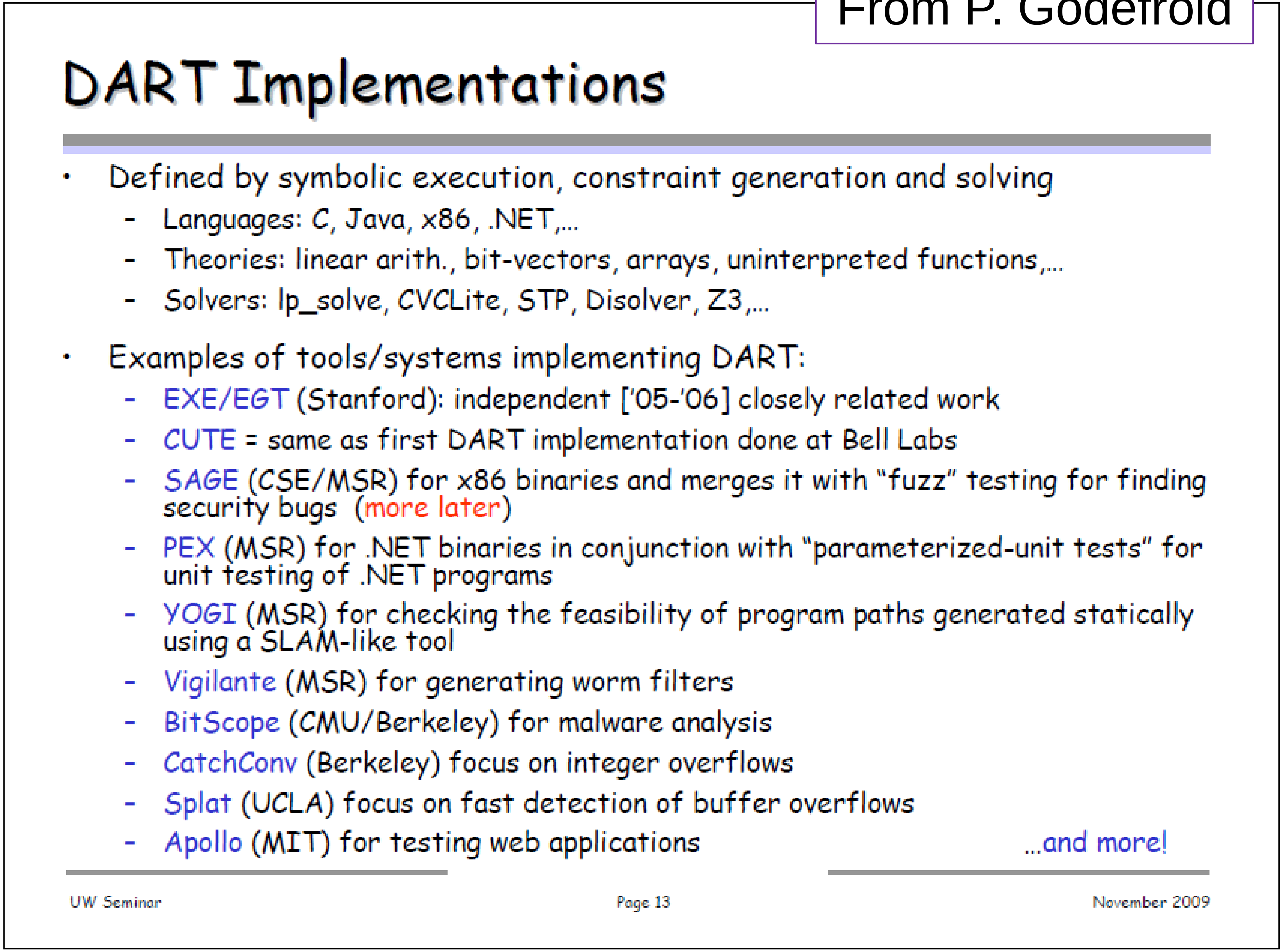
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

Concrete	Symbolic	Constraints

Concolic: status

- The jury is still out on concolic testing – but it surely has potential
- There are many papers on the general topic
- Here's one that is somewhat high-level Microsoft-oriented
  - Godefroid et al. Automating Software Testing Using Program Analysis *IEEE Software* (Sep/Oct 2008)
  - They tend to call the approach DART – Dynamic Automated Random Testing

From P. Godefroid



DART Implementations

- Defined by symbolic execution, constraint generation and solving
  - Languages: C, Java, x86, .NET,...
  - Theories: linear arith., bit-vectors, arrays, uninterpreted functions,...
  - Solvers: lp_solve, CVCLite, STP, Disolver, Z3,...
- Examples of tools/systems implementing DART:
  - EXE/EGT (Stanford): independent ['05-'06] closely related work
  - CUTE = same as first DART implementation done at Bell Labs
  - SAGE (CSE/MSR) for x86 binaries and merges it with "fuzz" testing for finding security bugs (more later)
  - PEX (MSR) for .NET binaries in conjunction with "parameterized-unit tests" for unit testing of .NET programs
  - YOGI (MSR) for checking the feasibility of program paths generated statically using a SLAM-like tool
  - Vigilante (MSR) for generating worm filters
  - BitScope (CMU/Berkeley) for malware analysis
  - CatchConv (Berkeley) focus on integer overflows
  - Splat (UCLA) focus on fast detection of buffer overflows
  - Apollo (MIT) for testing web applications

...and more!

My take

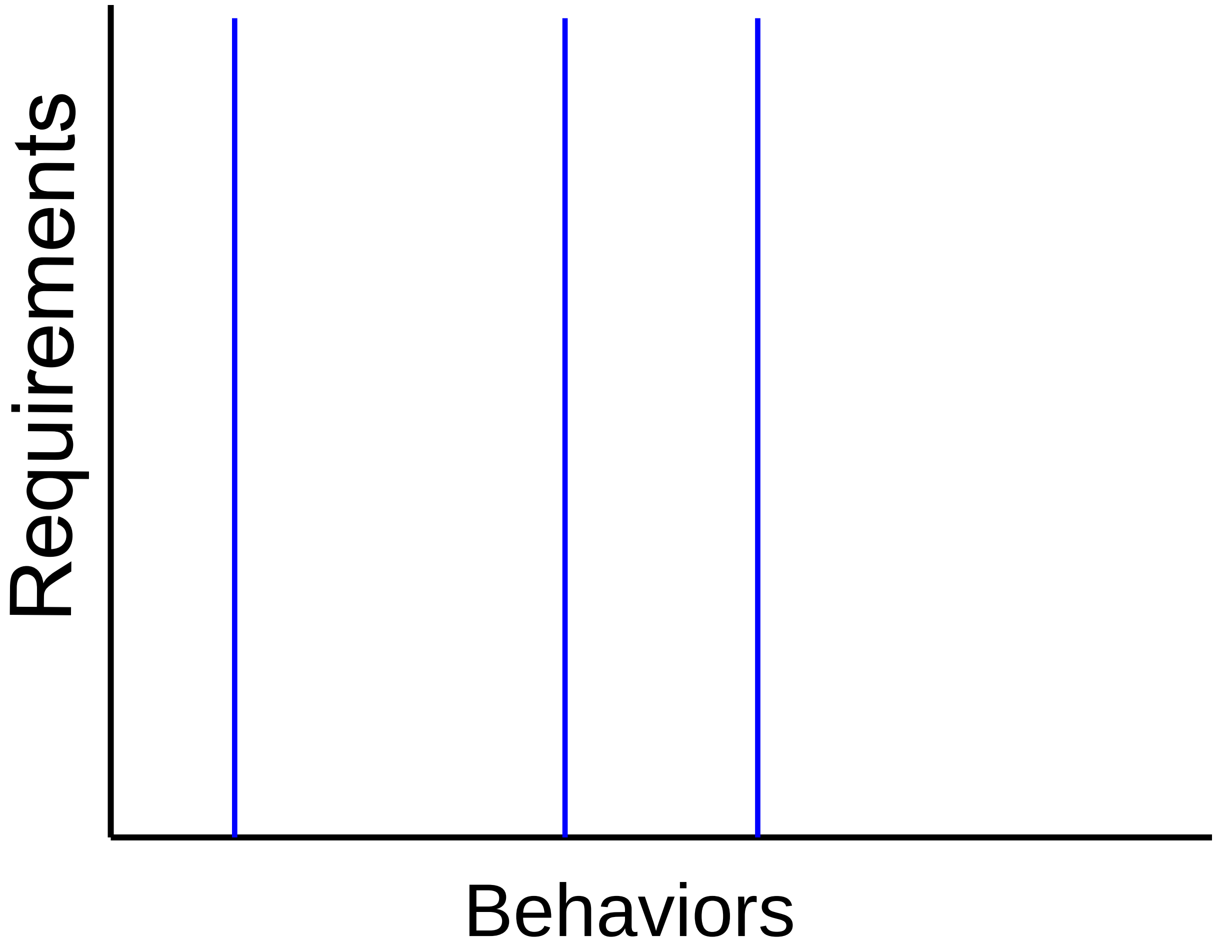
- The real story is the combination of symbolic evaluation, model checking, automated theorem proving, concrete testing, etc.
- These are being used and combined in ways that were previously not considered and/or were previously infeasible

- One other point: few if any of these systems actually help produce test suites with oracles – they rather help produce sets of test inputs that provide some kind of structural coverage
- This is fine, but it is not the full testing story – making sure the program computes what is wanted is also crucial

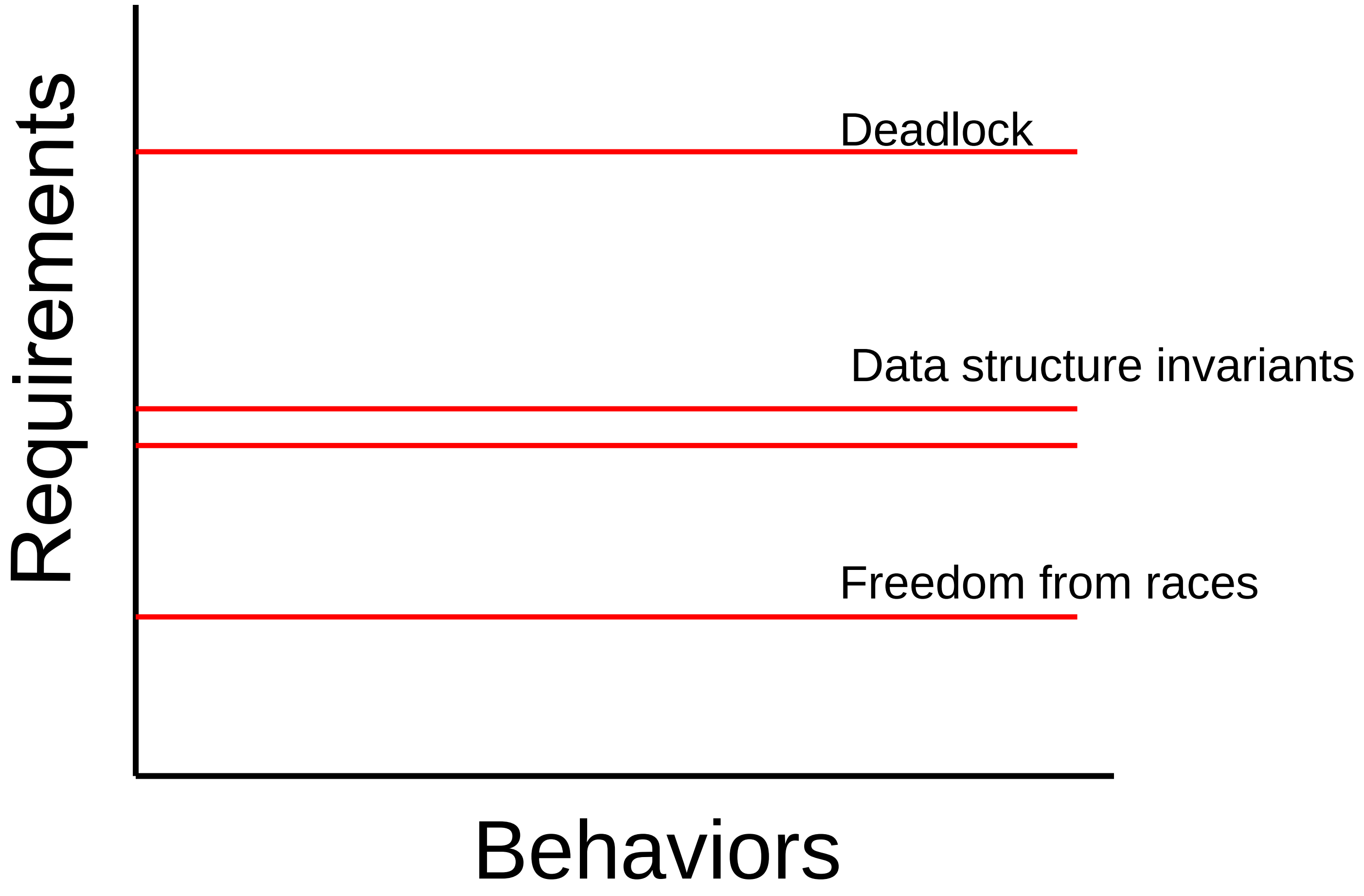
An aside: sources of unsoundness

- Matt Dwyer and colleagues have observed that in any form of analyzing a program (including analysis, testing, proving, …) there is a degree of unsoundness
- How do we know that
  - every desired property (correctness, performance, reliability, security, usability, …) is achieved in
  - every possible execution?
- We don’t – so we need to know what we know, and what we don’t know

Sample across executions



Sample across requirements



Continuous testing: Ernst et al.

- Run regression tests on every keystroke/save, providing rapid feedback about test failures as source code is edited
- Objectives: reduce the time and energy required to keep code well-tested, and prevent regression errors from persisting uncaught for long periods of time

Key results include

- Developers using continuous testing were three times more likely to complete the task before the deadline than those without (in a controlled experiment)
- Most participants found continuous testing to be useful and believed that it helped them write better code faster, and 90% would recommend the tool to others.
- Experimental supporting evidence that reducing the time between the introduction of an error and its discovery by a developer can lead to improvements in overall development time.

Test factoring

- “Expensive” tests (taking a long time to run, most often) are hard to handle “continuously” when they begin to fail
- Test factoring, given a large test, produces one or more smaller tests
- Each of these smaller tests is unlikely to fail unless the large test fails, and likely to regress (start to fail) when the large test regresses due to a particular kind of program change.

More details…

- Clever engineering, clever evaluation, and more
- http://www.cs.washington.edu/homes/mernst/research/#Testing (including continuous testing – old page at MIT)

Carving differential unit test cases from system test cases: Elbaum et al. FSE TSE

- Unit test cases are focused and efficient
- System tests are effective at exercising complex usage patterns
- Differential unit tests (DUT) are a hybrid of unit and system tests that exploits their strengths
- DUTs are generated by carving the system components, while executing a system test case, that influence the behavior of the target unit, and then re-assembling those components so that the unit can be exercised as it was by the system test
- Architecture, framework, implementation and empirical assessment of carving and replaying DUTs on three software artifacts

From FSE paper

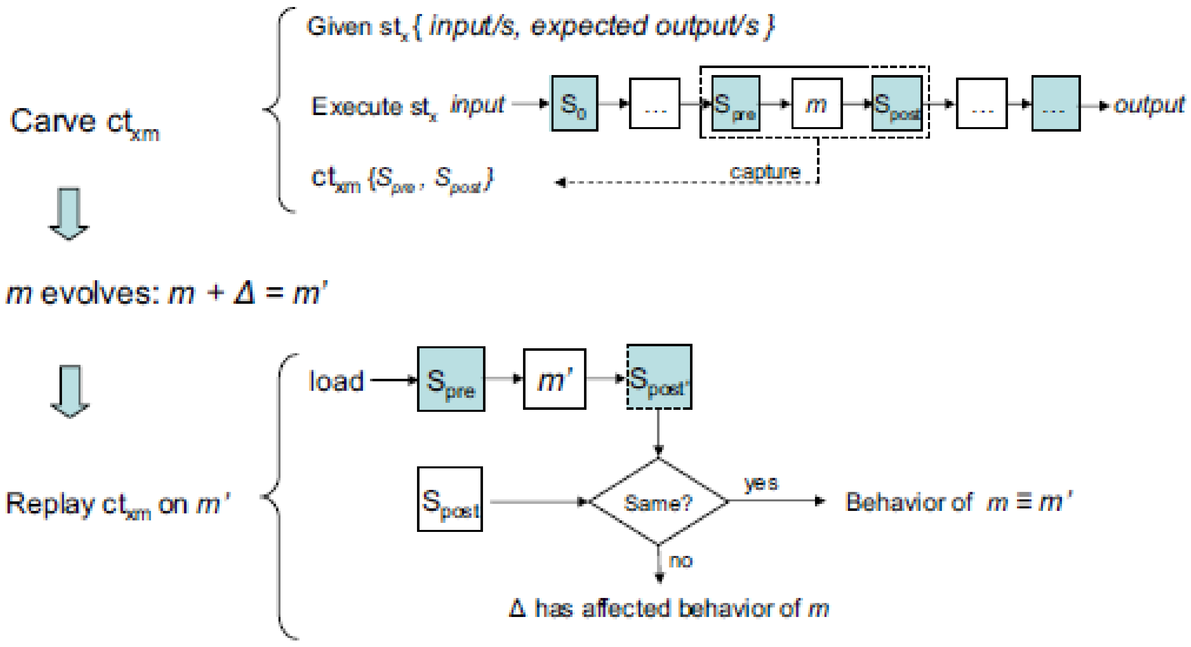


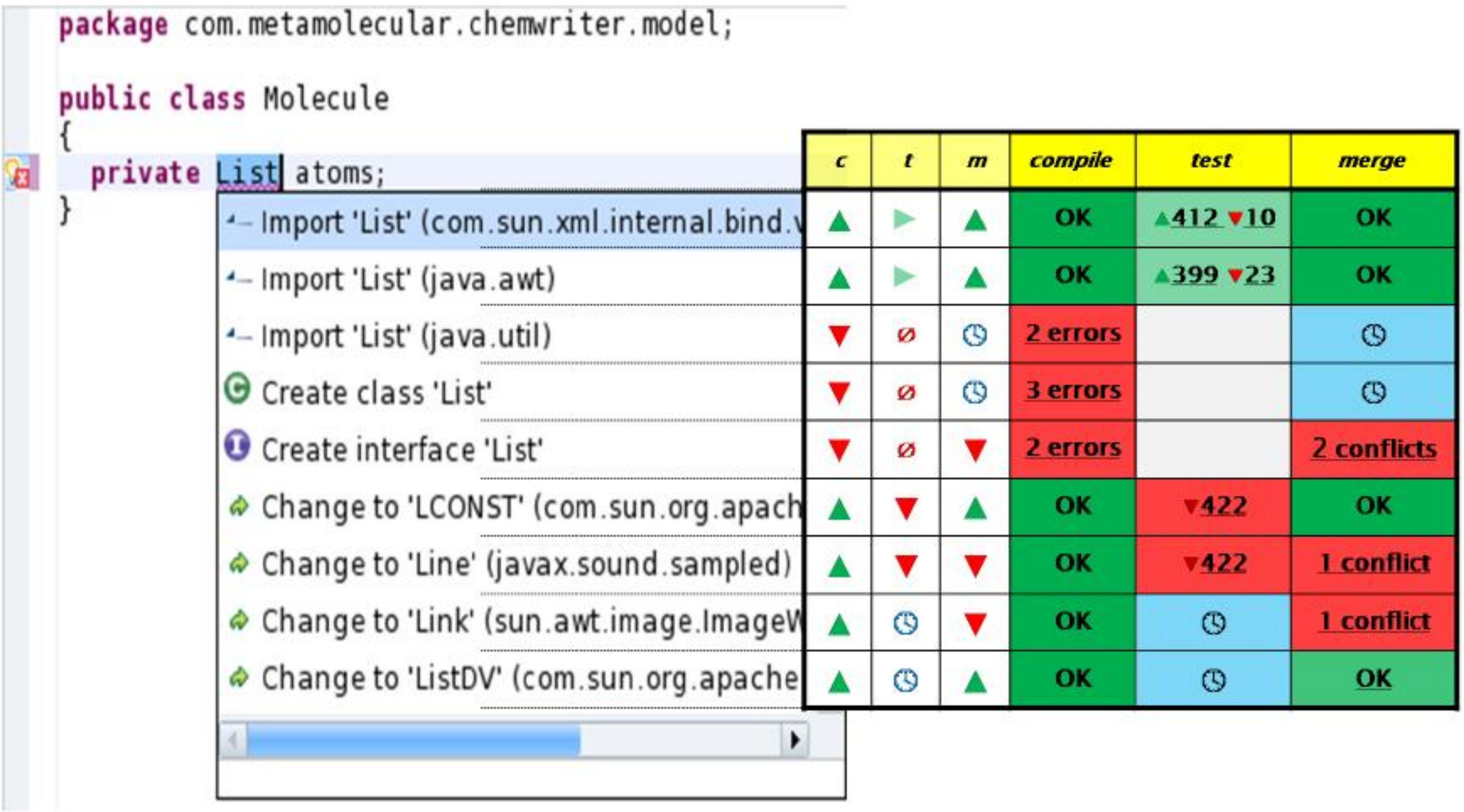
Figure 1: Carving and replay process.

"The Carving project is now a part of the new, bigger, and more ambitious T2T: Test-to-Test Transformation Project"

Speculation: again

- Continuous testing – in essence, trying to keep everything as up-to-date as possible
  - Using cycles for quality (not primarily for performance)
- Same two speculation slides, same motivation
- What if we had infinite cycles for quality and could provide up-to-date information about a set of possible actions?
  - This would also provide instantaneous transition to a new program state once an action was selected
- Discussion

Speculation: ongoing research @ UW



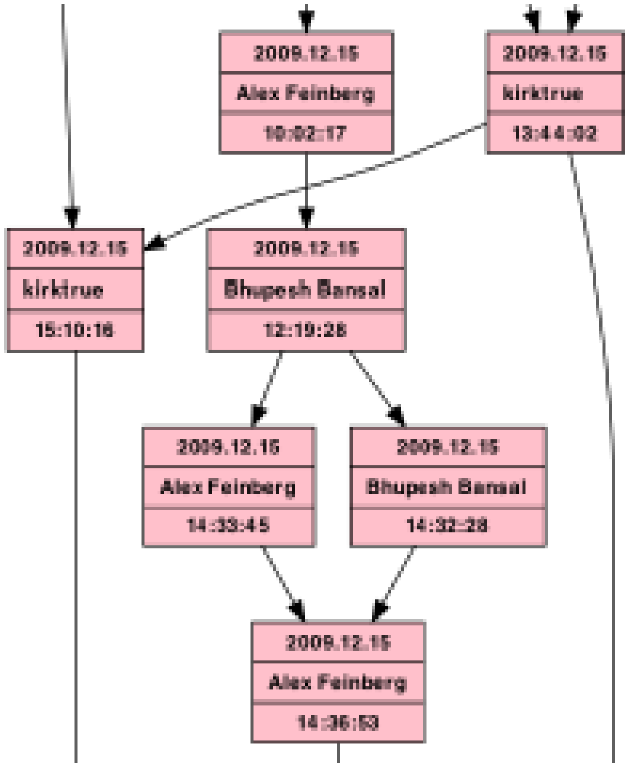
```
package com.metamolecular.chemwriter.model;

public class Molecule
{
  private List atoms;
}
```

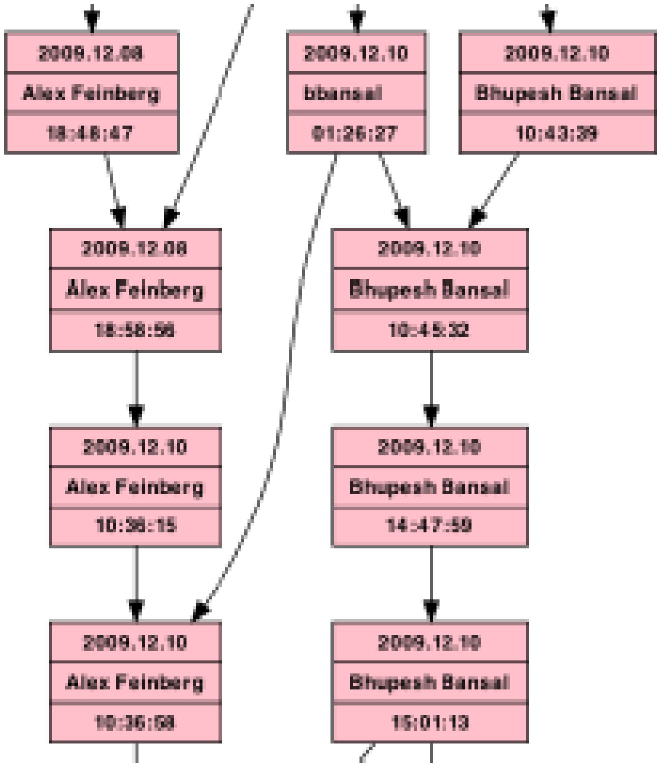
	c	t	m	compile	test	merge
Import 'List' (com.sun.xml.internal.bind.v	▲	▶	▲	OK	▲412 ▼10	OK
Import 'List' (java.awt)	▲	▶	▲	OK	▲399 ▼23	OK
Import 'List' (java.util)	▼	⌀	◷	2 errors		◷
Create class 'List'	▼	⌀	◷	3 errors		◷
Create interface 'List'	▼	⌀	▼	2 errors		2 conflicts
Change to 'LCONST' (com.sun.org.apach	▲	▼	▲	OK	▼422	OK
Change to 'Line' (javax.sound.sampled)	▲	▼	▼	OK	▼422	1 conflict
Change to 'Link' (sun.awt.image.ImageW	▲	◷	▼	OK	◷	1 conflict
Change to 'ListDV' (com.sun.org.apache	▲	◷	▲	OK	◷	OK

Figure 1: Mockup of the user interface for displaying contingent validation results.

Modified from http://depth-first.com/articles/2008/01/11/my-favorite-eclipse-shortcut-quick-fix © 2006-2007 Richard L. Apodaca. Original content licensed under the Creative Commons Attribution-Share Alike 3.0 United States License.

Speculation over merging?





LSDiff (M. Kim et al.): Help answer questions like …

Did Steve implement the intended changes correctly?

There's a merge conflict. What did Sally change?

Check-in comment
(revision 429 of carol open source project)

"Common methods go in an abstract class. Easier to extend/maintain/fix"

What changed?

What changed?

File Name	*Status*	*#Lines*
`DummyRegistry`	New	20
`AbsRegistry`	New	133
`JRMPRegistry`	Modified	123
`JeremieRegistry`	Modified	52
`JacORBCosNaming`	Modified	133
`IIOPCosNaming`	Modified	50
`CmiRegistry`	Modified	39
`NameService`	Modified	197
`NameServiceManager`	Modified	15

Changed code: 9 files, 723 lines

Was it really an *extract superclass* refactoring? Was any part of the refactoring missed? Did Steve make any other changes?

Try diff

File Name	*Status*	*#Lines*
`DummyRegistry`	New	20
`AbsRegistry`	New	133
`JRMPRegistry`	Modified	123
`JeremieRegistry`	Modified	52
`JacORBCosNaming`	Modified	133
`IIOPCosNaming`	Modified	50
`CmiRegistry`	Modified	39
`NameService`	Modified	197
`NameServiceManager`	Modified	15

Changed code: 9 files, 723 lines

Try diff

File Name	*Status*	*#Lines*
`DummyRegistry`	New	20
`AbsRe`		
`JRMPR`		
`Jerem`		
`JacOR`		
`IIOPC`		
`CmiRe`		
`NameS`		
`NameServiceManager`	Modified	15

```
- public class CmiRegistry implements NameService {
+ public class CmiRegistry extends AbsRegistry implements NameService {
-    private int port = ...
-    private String host = null
-    public void setPort (int p) {
-       if (TraceCarol. isDebug()) { ...
-       }
-     }
-     public int getPort() {
-       return port;
-      }
-     public void setHost(String host) { ...
```

Changed code: 9 files, 723 lines

Related diff-like approaches

- Syntactic Diff (Cdiff), Semantic Diff, Jdiff, BMAT, Eclipse diff, UMLdiff, Change Distiller, …
- They individually compare code elements at specific granularities using various similarity measures
  - Code elements may be lines, abstract syntax trees, control flow graphs, etc.
  - Similarity is usually based on names and structure
- These tools provide information that is accurate and useful but not well-suited to helping engineers and managers answer the kinds of questions we want

Use systematic change

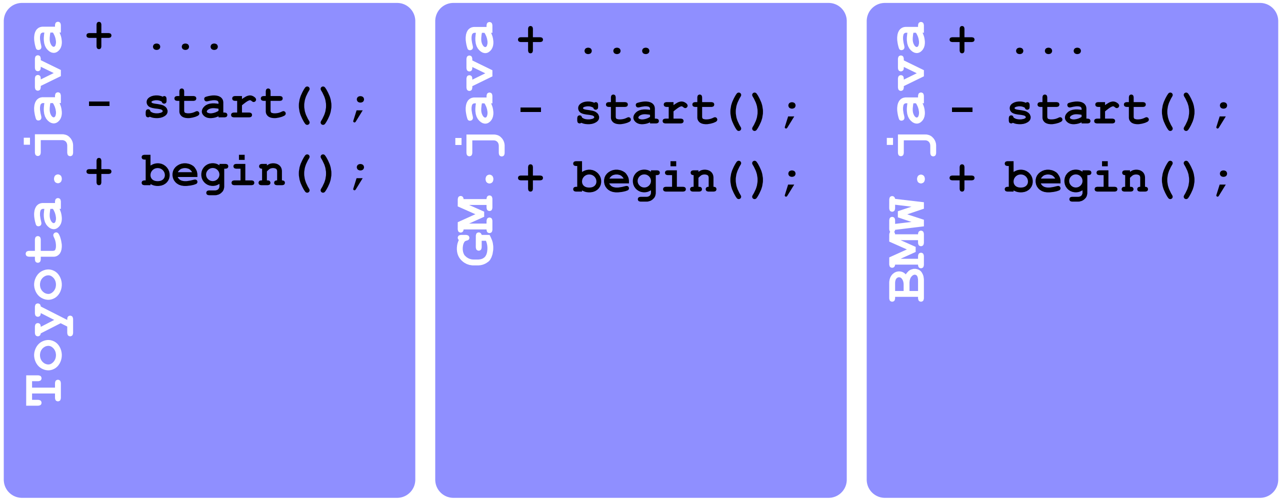
- Existing diff-based tools do not exploit the fact that programmers often make high-level changes in part by systematically applying lower-level changes
- Systematic changes are widespread; examples include
  - Refactoring [Opdyke 92, Griswold 92, Fowler 99...]
  - API update [Chow & Notkin 96, Henkel & Diwan 05, Dig & Johnson 05...]
  - Crosscutting concerns [Kiczales et. al. 97, Tarr et. al. 99, Griswold 01...]
  - Consistent updates on code clones [Miller & Myers 02, Toomim et. al. 04, Kim et. al. 05, …]

Limitations of diff-based approaches

- These approaches do not group related changes with respect to a high-level change – but rather by structural program units such as files
- In part because of this first limitation, they do not make it easy to identify incomplete or missed parts of high-level changes
- They leave it to the programmer to discover any useful contextual information surrounding the low-level changes

- In other words, these approaches are program-centric but not *change-centric*

Ex: No change-based grouping



Toyota.java

```
+ ...
- start();
+ begin();
```

GM.java

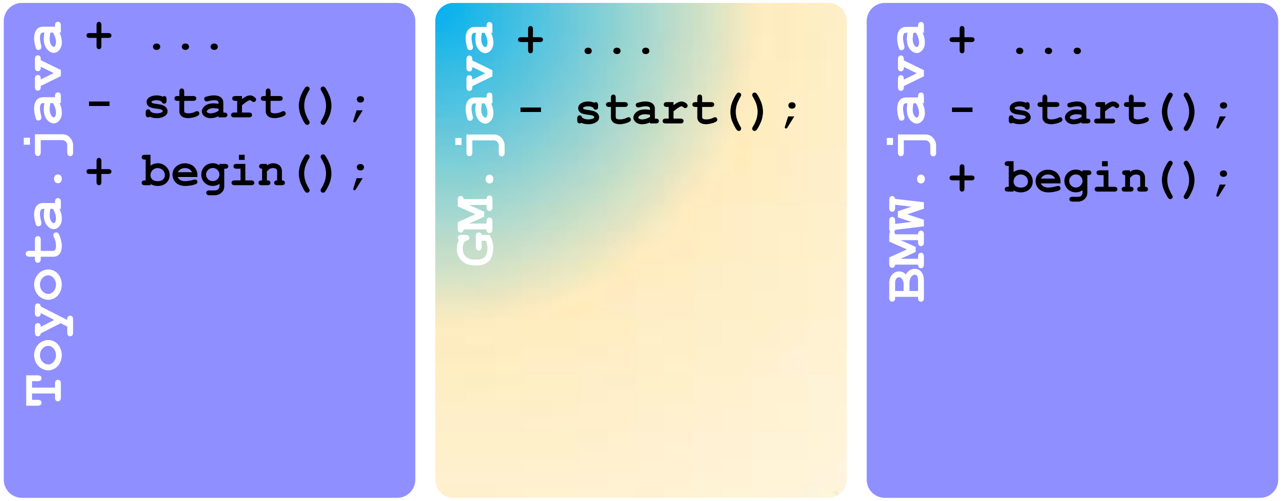
```
+ ...
- start();
+ begin();
```

BMW.java

```
+ ...
- start();
+ begin();
```

- The programmer must determine that the same changes have been made in these three related classes – if they even choose to think about this

Ex: Hard to see missed changed



Toyota.java

```
+ ...
- start();
+ begin();
```

GM.java

```
+ ...
- start();
```

BMW.java

```
+ ...
- start();
+ begin();
```

- The programmer must decide to look for a missing or inconsistent change – there is no help from the tool

Ex: Lack of contextual information

```
class Car
   ...
   run () {
   ...
   }
```

```
class Toyota
extends Car

+ run(){
+ ...
+ }
```

```
class GM
extends Car

+ run(){
+ ...
+ }
```

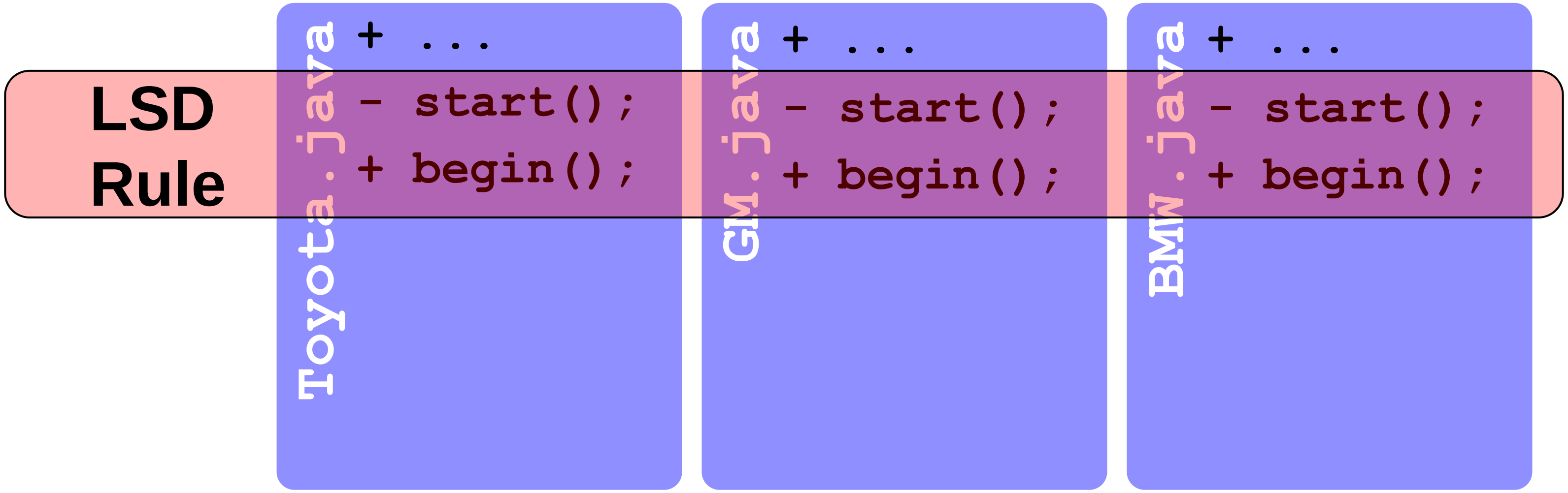
```
class BMW
extends Car

+ run(){
+ ...
+ }
```

- Three subclasses of a class changed in the same way would not be identified by the tools themselves

The Logical Structural Diff Approach

- LSDiff computes structural differences between two versions using logic rules and facts
- Each rule represents a group of transformations that share similar structural characteristics – a systematic change
- Our inference algorithm automatically discovers these rules

Conciseness



LSD Rule

Toyota.java

+ ...

- start();

+ begin();

GM.java

+ ...

- start();

+ begin();

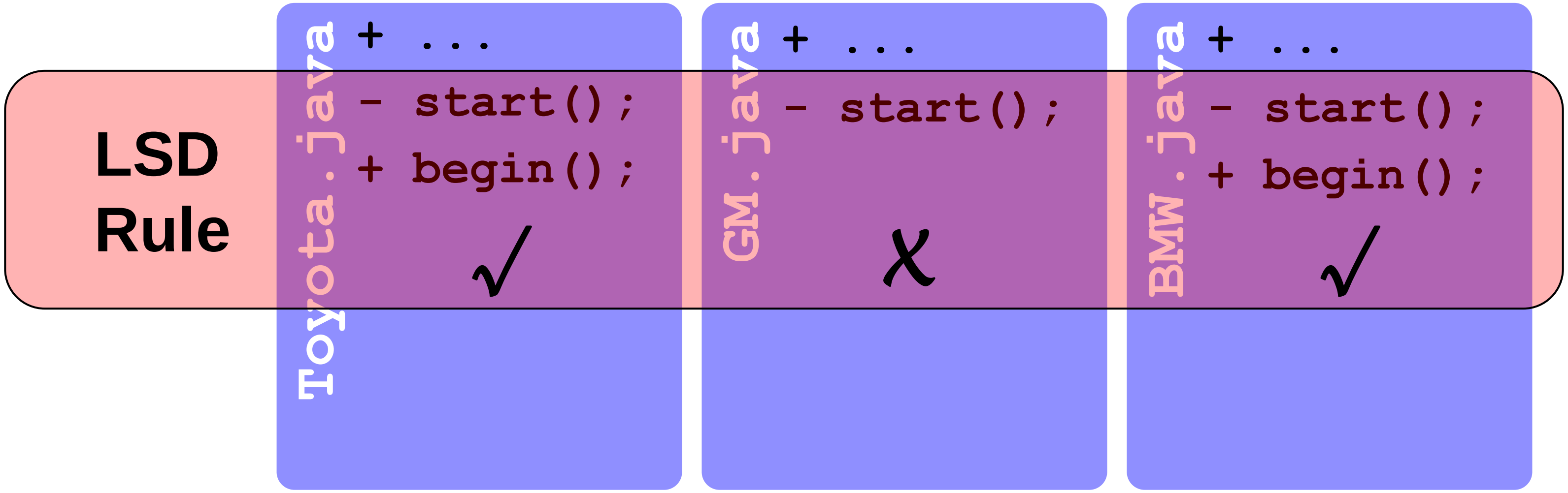
BMW.java

+ ...

- start();

+ begin();

Explicit exceptions



LSD Rule

Toyota.java

```
+ ...
- start();
+ begin();
```

✓

GM.java

```
+ ...
- start();
```

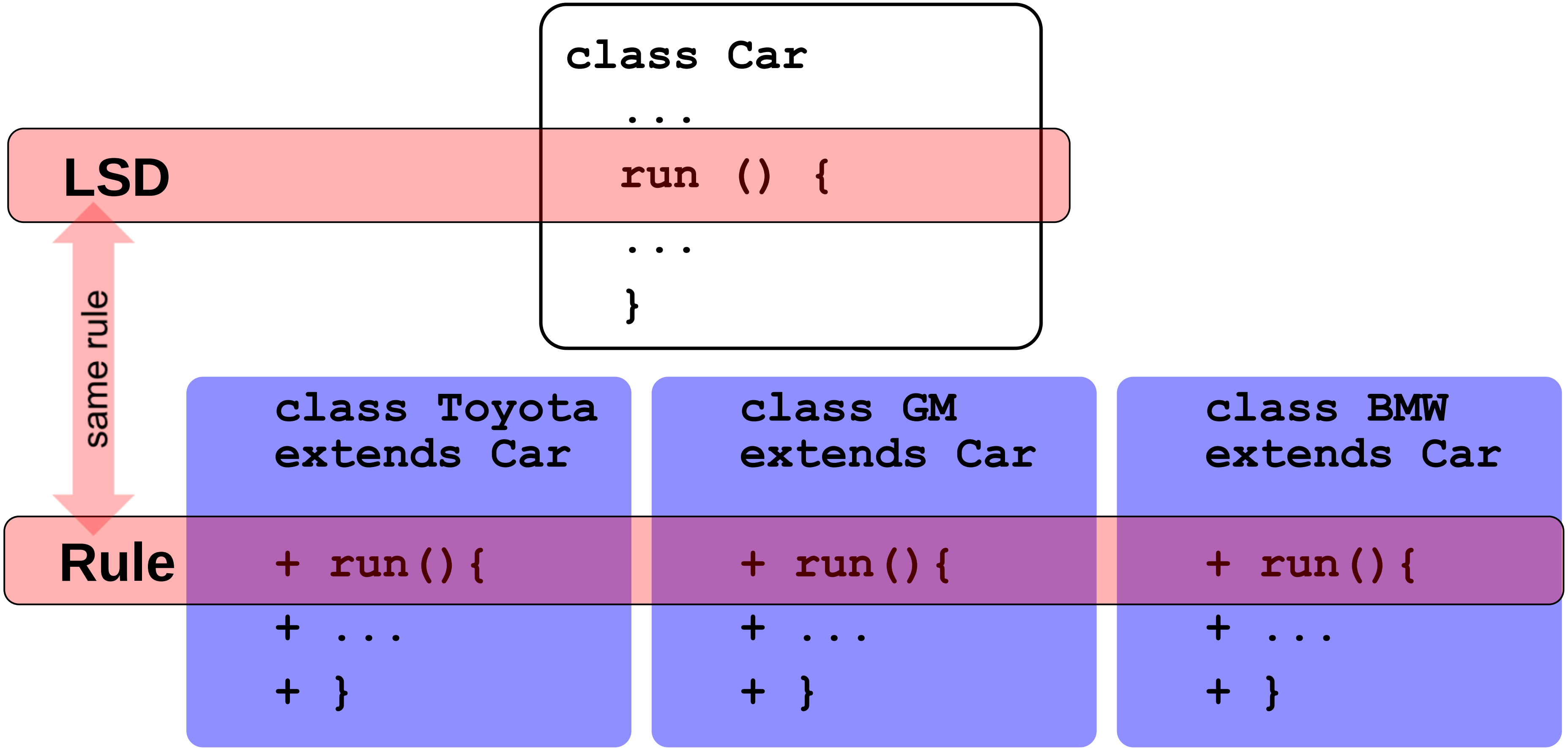
✗

BMW.java

```
+ ...
- start();
+ begin();
```

✓

Additional context



```
class Car
   ...
   run () {
   ...
   }
```

```
class Toyota
extends Car

+ run(){
+ ...
+ }
```

```
class GM
extends Car

+ run(){
+ ...
+ }
```

```
class BMW
extends Car

+ run(){
+ ...
+ }
```

Program representation

- We abstract Java programs at the level of code elements and structural dependencies
- Predicates represent package, type, method, field, sub-typing, overriding, method calls, field accesses and containment relationships

- ➢ `package`
- ➢ `type`
- ➢ `method`
- ➢ `field`
- ➢ `return`
- ➢ `fieldoftype`
- ➢ `typeintype`
- ➢ `accesses`
- ➢ `calls`
- ➢ `subtype`
- ➢ `inheritedfield`
- ➢ `inheritedmethod`

Fact-based representation

- Analyze a program’s abstract syntax tree and return a fact-base of these predicates (using JQuery [Jensen & DeVolder 03])
- Repeat for the modified program

```
type(“Bus”,..)
method(“Bus.start”,”start”,”Bus”)
access(“Key.on”,”Bus.start”)
method(“Key.out”,”out”,”Key”)...
```

Old program

FB_o

past_

```
type(“Bus”,..)
method(“Bus.start”,”start”,”Bus”)
calls(“Bus.start”,”log”)
method(“Key.output”,”output”,”Key”)...
```

New program

FB_n

current_

Compute $\Delta FB = FB_o - FB_n$

```
deleted_access(“Key.on”,”Bus.start”)
added_calls(“Bus.start”,”log”)
deleted_method(“Key.out”,”out”,”Key”)
added_method(“Key.output”,”output”,”Key”)
...
```

LSDiff Rule Quantification

- Rules represent systematic structural differences that relates groups of facts from the three fact-bases – FB_o, FB_n, ΔFB
- Universally quantified variables allow rules to represent a group of similar facts at once
  - For example, **∀m∀t method(m,"setHost",t)** refers to all methods named **setHost** in all types
  - Ex: **∀t subtype("Service", t)**
  - Ex: **∀m calls(m, "SQL.exec")**

LSD Rules

- Rules are Horn clauses where a conjunction of logic literals implies a single consequent literal

-
  ```
  ∀m ∀t method(m, “setHost”, t) ∧
        subtype(“Service”, t)
        ⇒ calls(m, “SQL.exec”)
  ```

Rules across versions

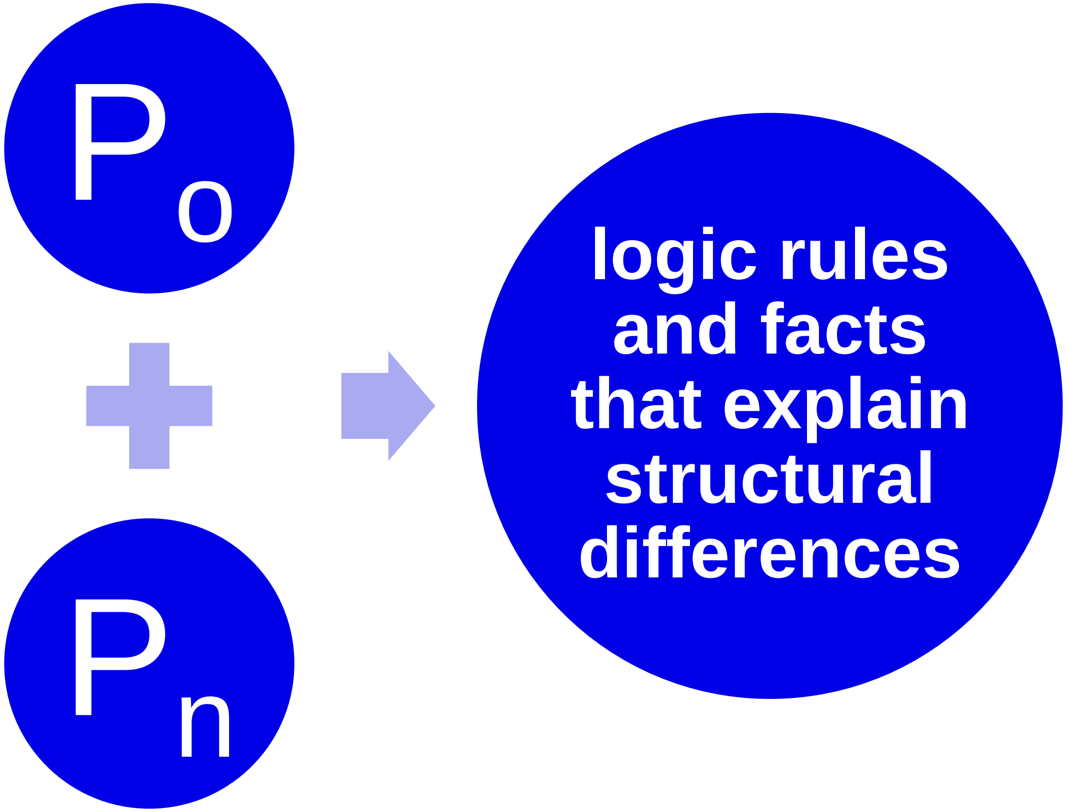
-
  ```
  ∀m ∀t past_method(m, “setHost”, t) ∧
        past_subtype(“Service”, t)
        ⇒ deleted_calls(m, “SQL.exec”)
  ```

Rules note exceptions

```
• ∀m ∀t past_method(m, “setHost”, t) ∧
        past_subtype(“Service”, t)
        ⇒ deleted_calls(m, “SQL.exec”)
          except t=“NameSvc”,
                m=”NameSvc.setHost”
```

- “All **`setHost`** methods in **`Service`**’s subclasses in the old version deleted calls to **`SQL.exec`** except the **`setHost`** method in the **`NameSvc`** class.”
- A parameter defines when exceptions are found and reported

Algorithm Overview



logic rules and facts that explain structural differences

1. Extract logic facts from programs and compute fact-level differences
2. Learn rules using a customized inductive logic programming algorithm
3. Select a subset of rules and then remove the facts in ΔFB using the learned rules

Learn rules

- Inductive logic programming with a bounded depth search based on beam search heuristics
- Input parameters determine the validity of a rule
  - **m**: the minimum # of facts a rule must match – enough evidence for a rule?
  - **a**: the minimum accuracy of a rule – enough evidence for an exception?
  - **k**: the maximum # of literals in an antecedent
  - **β**: the window size for beam search
- A sequential covering algorithm that iteratively finds rules and removes covered facts
- Generate rules starting with an empty antecedent and adding literals (e.g., from general to specific)
- Learn partially grounded rules by substituting variables of ungrounded rules with constants

Learn rules

```
R := {}     // a set of ungrounded rules
L := {}     // a set of valid learned rules
D := reduced ΔFB using default winnowing rules
for each antecedent size, i = 0...k :
    R := extend all rules in R by adding
         all possible literals
    for each ungrounded rule, r:
        for each possible grounded rule g of r:
            if (g is valid) L:= L U g
    R := select the best β rules in R
    D := D - { facts covered by L }
```

Select rules

- Some rules explain the same set of facts in ΔFB
- So we use a set cover algorithm to select a subset of learned rules

- Return the selected rules, remove the facts that those rules cover, and return any remaining uncovered facts in `ΔFB`

LSD Example

- To prevent an injection attack, a programmer replaced all calls to **SQL.exec** to **SafeSQL.exec**
- LSD infers the following rule

  -
    ```
    deleted_calls(m,“SQL.exec”) ⇒
         added_calls(m,“SafeSQL.exec”)
    ```
- And another rule we’ve seen before, suggesting a deletion was not done

  -
    ```
    past_subtype(“Service”, t) ∧
    past_method(m, “setHost”, t) ⇒
        deleted calls(m, “SQL.exec”)
        except t=“NameSvc”
    ```

Quantitative evaluation

- How often do individual changes form systematic change patterns?
  - Measure coverage, # of facts in ΔFB matched by inferred rules
- How concisely does LSD describe structural differences in comparison to existing differencing approach at the same abstraction level?
  - Measure conciseness, ΔFB / (# rules + # facts)
- How much contextual information does LSD find from unchanged code fragments?
  - Measure the number of facts mentioned by rules but are not contained in ΔFB

Quantitative evaluation

a=0.75, m=3, k=2, β=100

	FB_o/FB_n	ΔFB	Rule	Fact	Cover-age	Concise-ness	Context facts
carol 10 revisions	3080 ~ 10746	15 ~ 1812	1 ~ 36	3 ~ 71	59 ~ 98%	2.3 ~ 27.5	0 ~ 19
dnsjava 29 releases	3109 ~ 7204	4 ~ 1500	0 ~ 36	2 ~ 201	0 ~ 98%	1.0 ~ 36.1	0 ~ 91
LSdiff 10 versions	8315 ~ 9042	2 ~ 396	0 ~ 6	2 ~ 54	0 ~ 97%	1.0 ~ 28.9	0 ~ 12

Quantitative evaluation

a=0.75, m=3, k=2, β=100

	FB_o/FB_n	ΔFB	Rule	Fact	Cover-age	Concise-ness	Context facts
carol 10 revisions	3080 ~ 10740	15 ~ 1812	1 ~ 36	3 ~ 71	59 ~ 98%	2.3 ~ 27.5	0 ~ 19
dnsjava 29 releases	3109 ~ 7204	4 ~ 1500	0 ~ 36	2 ~ 201	0 ~ 98%	1.0 ~ 36.1	0 ~ 91
LSdiff 10 versions	8315 ~ 9042	2 ~ 396	0 ~ 6	2 ~ 54	0 ~ 97%	1.0 ~ 28.9	0 ~ 12

On average, 75% coverage,

9.3 times conciseness improvement,

9.7 additional contextual facts

Textual Delta vs. LSD

a=0.75, m=3, k=2, β=100

	Textual Delta				LSD	
	Changed Files	Changed Lines	Hunks	% Touched	Rule	Fact
carol 10 revisions	1 ~ 35	67 ~ 4313	9 ~ 132	1 ~ 19	1 ~ 36	3 ~ 71
dnsjava 29 releases	1 ~ 117	5 ~ 15915	1 ~ 344	2 ~ 100	0 ~ 36	2 ~ 201
LSdiff 10 versions	2 ~ 11	9 ~ 747	2 ~ 39	2 ~ 9	0 ~ 6	2 ~ 54

Textual Delta vs. LSD

a=0.75, m=3, k=2, β=100

	Textual Delta				LSD	
	Changed Files	Changed Lines	Hunks	% Touched	Rule	Fact
carol 10 revisions	1 ~ 35	67 ~ 4313	9 ~ 132	1 ~ 19	1 ~ 36	3 ~ 71
dnsjava 29 releases	1 ~ 117	5 ~ 15915	[illegible] ~ 344	2 ~ 100	0 ~ 36	2 ~ 201
LSdiff 10 versions	2 ~ 11	9 ~ 747	2 ~ 39	2 ~ 9	0 ~ 6	2 ~ 54

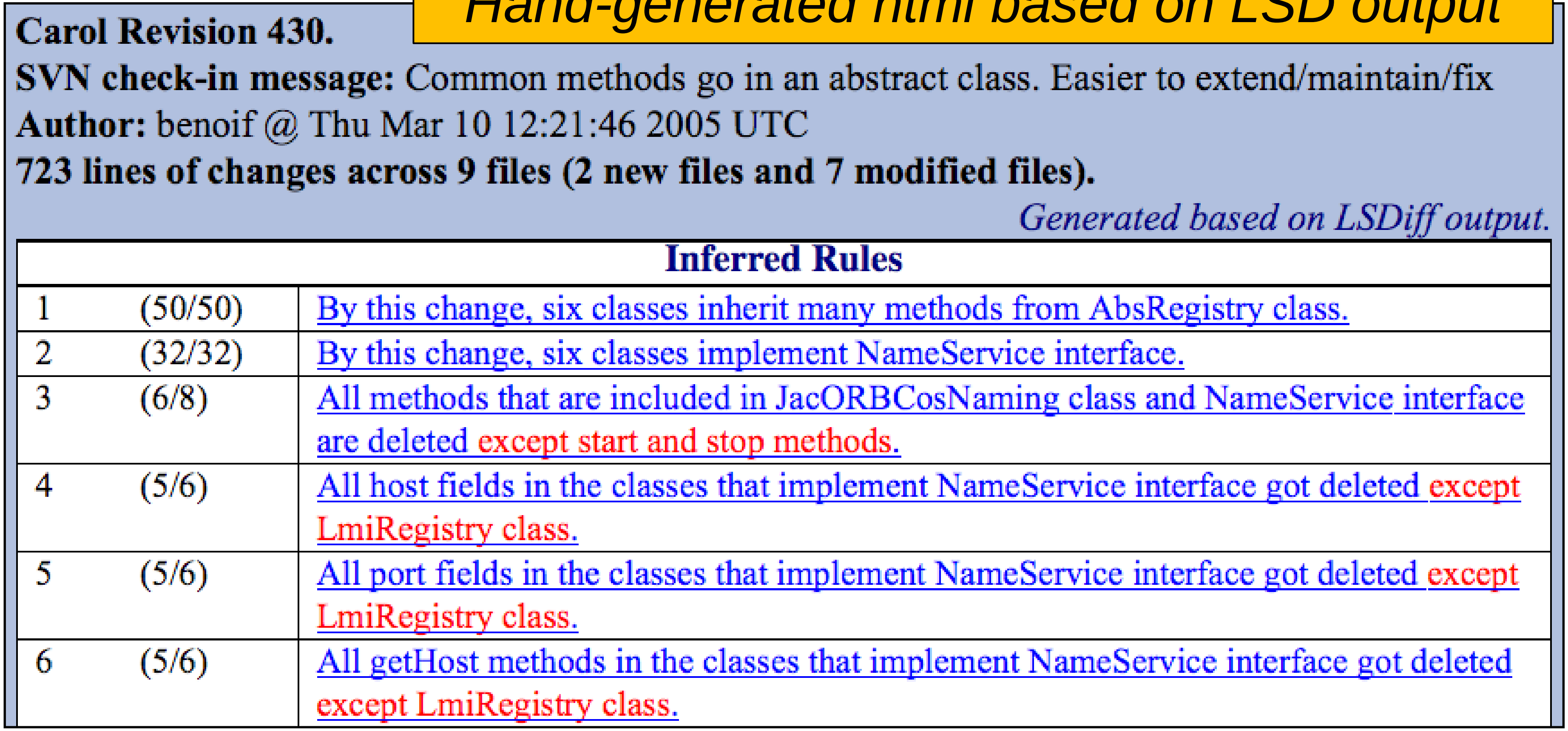
When an average text delta consists of 997 lines across 16 files, LSD outputs an average of 7 rules and 27 facts

Focus group: e-commerce company

- Pre-screener survey
- Participants: five professional software engineers
  - industry experience ranging from six to over 30 years
  - use diff and diff-based version control system daily
  - review code changes daily except one who did weekly
- One hour structured discussion
  - Professor Kim worked as the moderator
  - There was also a note-taker and the discussion was audio-taped and transcribed

Focus Group Hands-On Trial

Hand-generated html based on LSD output



Carol Revision 430.

SVN check-in message: Common methods go in an abstract class. Easier to extend/maintain/fix

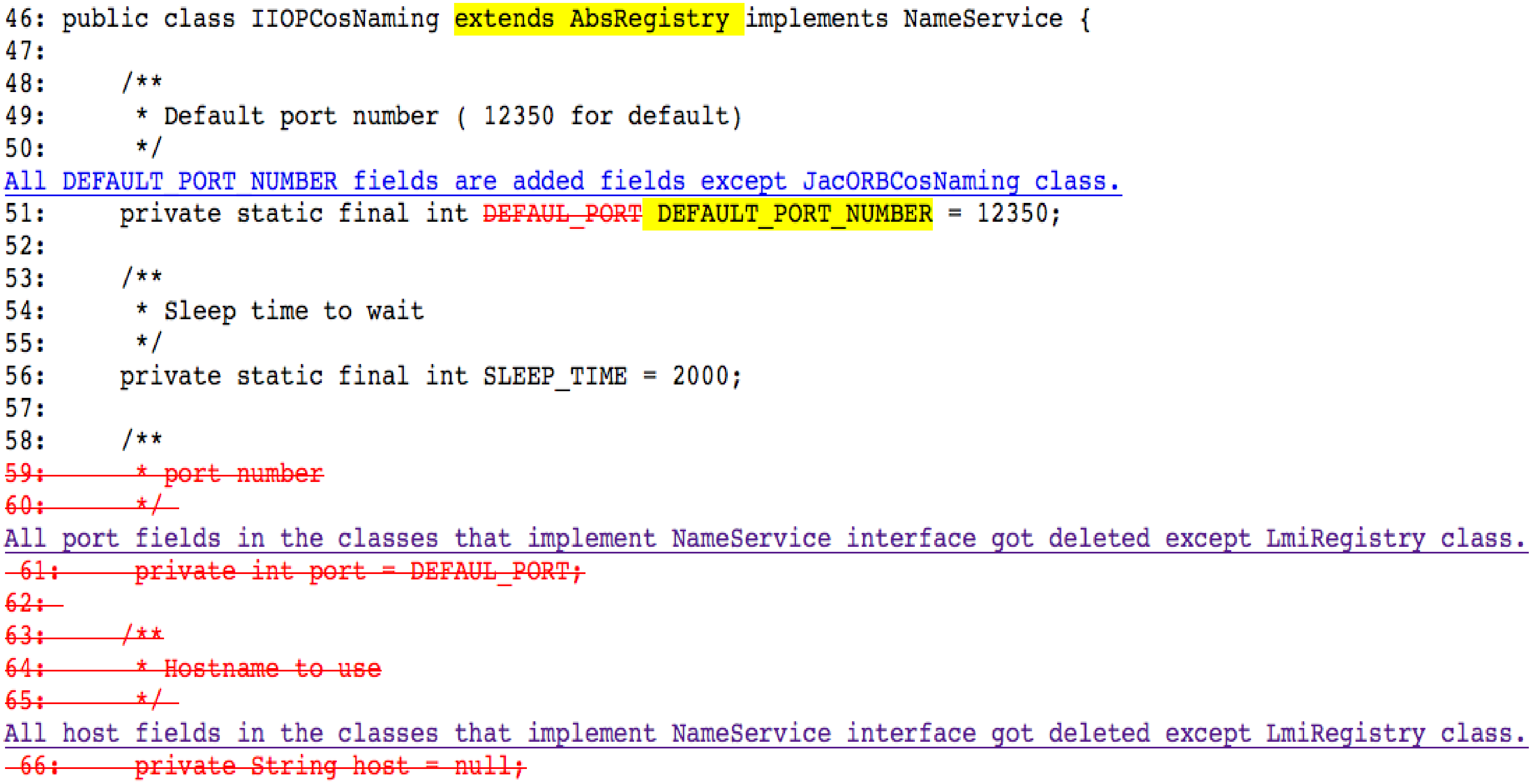
Author: benoif @ Thu Mar 10 12:21:46 2005 UTC

723 lines of changes across 9 files (2 new files and 7 modified files).

Generated based on LSDiff output.

		Inferred Rules
1	(50/50)	By this change, six classes inherit many methods from AbsRegistry class.
2	(32/32)	By this change, six classes implement NameService interface.
3	(6/8)	All methods that are included in JacORBCosNaming class and NameService interface are deleted except start and stop methods.
4	(5/6)	All host fields in the classes that implement NameService interface got deleted except LmiRegistry class.
5	(5/6)	All port fields in the classes that implement NameService interface got deleted except LmiRegistry class.
6	(5/6)	All getHost methods in the classes that implement NameService interface got deleted except LmiRegistry class.

http://users.ece.utexas.edu/~miryung/LSDiff/carol429-430.htm



```
public class IIOPCosNaming extends AbsRegistry implements NameService {

    /**
     * Default port number ( 12350 for default)
     */
```

All DEFAULT PORT NUMBER fields are added fields except JacORBCosNaming class.

```
    private static final int ~~DEFAUL_PORT~~ DEFAULT_PORT_NUMBER = 12350;

    /**
     * Sleep time to wait
     */
    private static final int SLEEP_TIME = 2000;

    /**
~~     * port number~~
~~     */~~
```

All port fields in the classes that implement NameService interface got deleted except LmiRegistry class.

```
~~    private int port = DEFAUL_PORT;~~

~~    /**~~
~~     * Hostname to use~~
~~     */~~
```

All host fields in the classes that implement NameService interface got deleted except LmiRegistry class.

```
~~    private String host = null;~~
```

Focus Group Comments (some)

- “You can’t infer the intent of a programmer, but this is pretty close.”
- “This ‘except’ thing is great!”
- “You can start with the summary of changes and dive down to details using a tool like diff.”

Focus group comments (more)

- “This looks great for big architectural changes, but I wonder what it would give you if you had lots of random changes.”
- “This wouldn’t be used if you were just working with one file.”
- “This will look for relationships that do not exist.”

- Unsurprising comments as we focus on recovering systematic changes rather than heterogeneous changes
- When the delta is small, diff should works fine

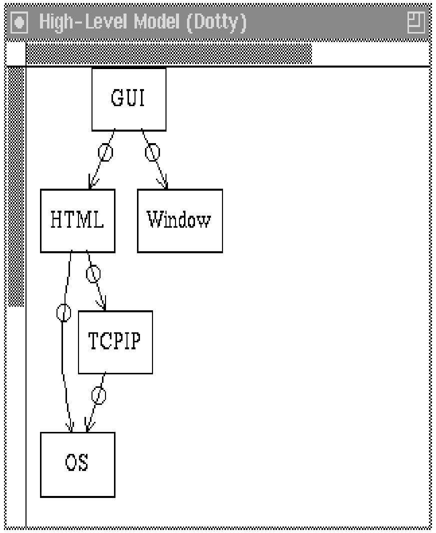
LSDiff plug-in for Eclipse

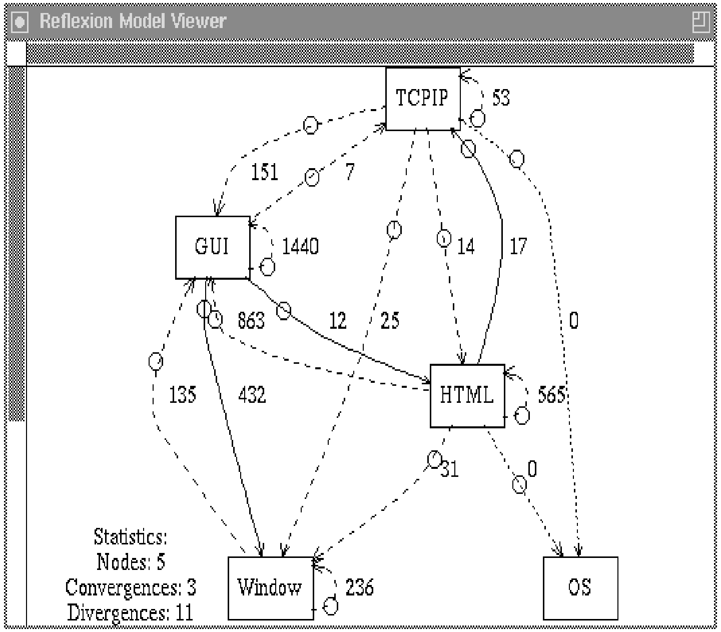
- And some other projects related to summarizing changes as rules

Languages and tools
Tools and languages

- The line between programming languages and tools (programs that help programmers write programs) is sometimes fuzzy
- Examples
  - **`lint`** vs. type systems

Summarization





Statistics:
Nodes: 5
Convergences: 3
Divergences: 11

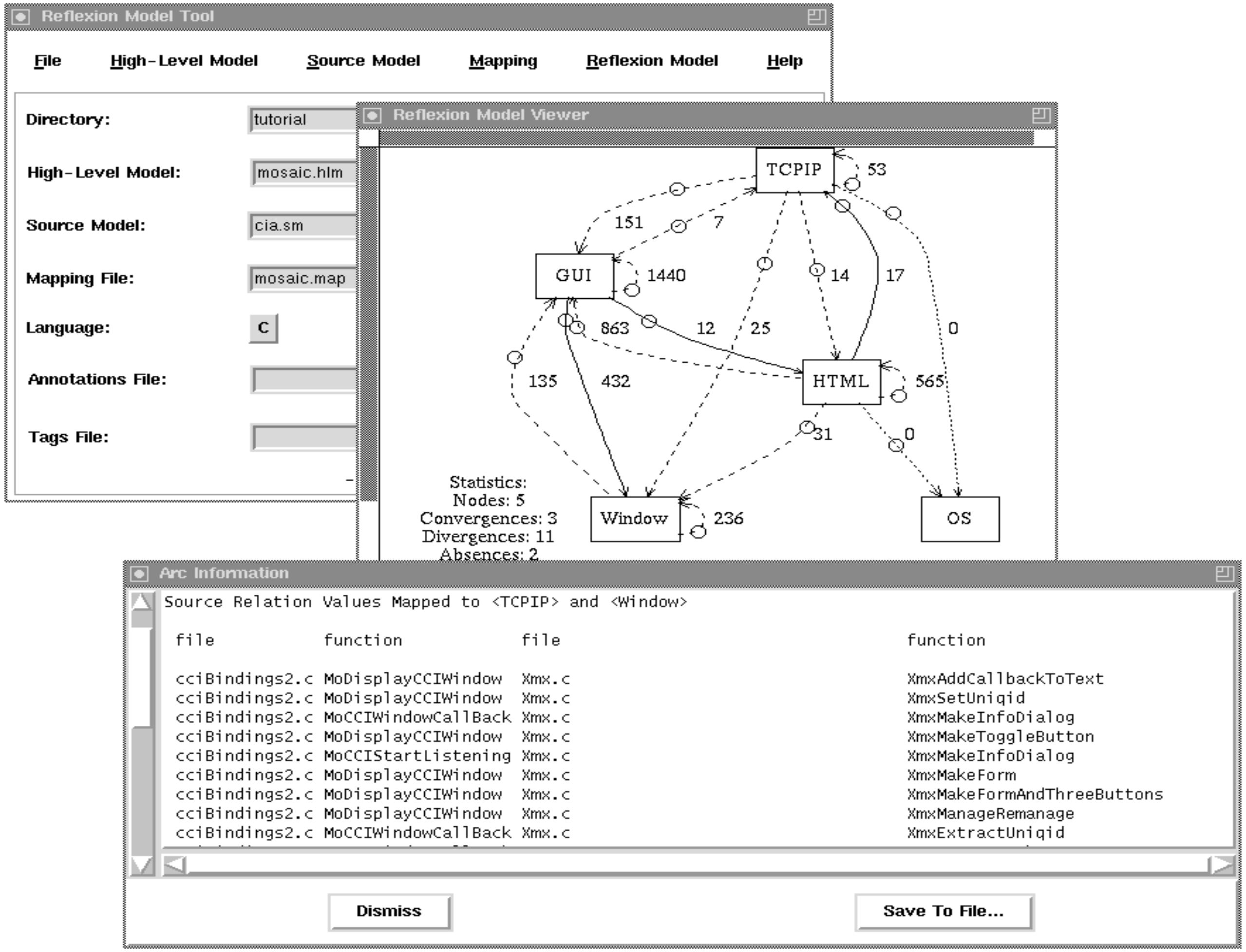
- e.g., software reflexion models

Summarization...

- A map file specifies the correspondence between parts of the source model and parts of the high-level model

```
[ file=HTTCP       mapTo=TCPIP ]
[ file=^SGML       mapTo=HTML ]
[ function=socket  mapTo=TCPIP ]
[ file=accept      mapTo=TCPIP ]
[ file=cci         mapTo=TCPIP ]
[ function=connect mapTo=TCPIP ]
[ file=Xm          mapTo=Window ]
[ file=^HT         mapTo=HTML ]
[ function=.*      mapTo=GUI ]
```

Summarization...



Directory: tutorial

High-Level Model: mosaic.hlm

Source Model: cia.sm

Mapping File: mosaic.map

Language: C

Annotations File:

Tags File:

Statistics:
Nodes: 5
Convergences: 3
Divergences: 11
Absences: 2

```
Source Relation Values Mapped to <TCPIP> and <Window>

 file           function           file                                  function

 cciBindings2.c MoDisplayCCIWindow  Xmx.c                                 XmxAddCallbackToText
 cciBindings2.c MoDisplayCCIWindow  Xmx.c                                 XmxSetUniqid
 cciBindings2.c MoCCIWindowCallBack Xmx.c                                 XmxMakeInfoDialog
 cciBindings2.c MoDisplayCCIWindow  Xmx.c                                 XmxMakeToggleButton
 cciBindings2.c MoCCIStartListening Xmx.c                                 XmxMakeInfoDialog
 cciBindings2.c MoDisplayCCIWindow  Xmx.c                                 XmxMakeForm
 cciBindings2.c MoDisplayCCIWindow  Xmx.c                                 XmxMakeFormAndThreeButtons
 cciBindings2.c MoDisplayCCIWindow  Xmx.c                                 XmxManageRemanage
 cciBindings2.c MoCCIWindowCallBack Xmx.c                                 XmxExtractUniqid
```

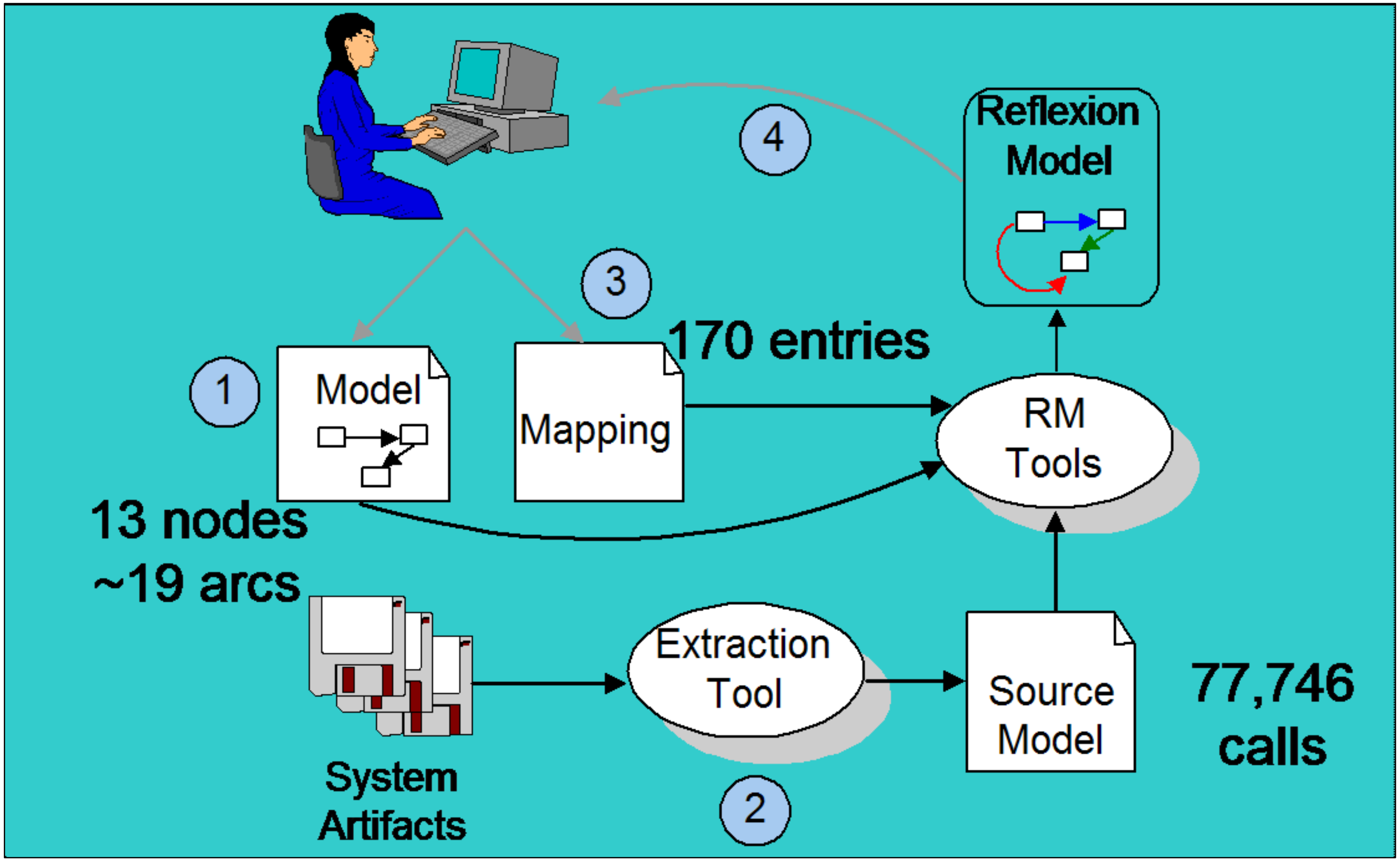
Summarization...

- Condense (some or all) information in terms of a high-level view quickly
  - In contrast to visualization and reverse engineering, produce an “approximate” view
  - Iteration can be used to move towards a “precise” view
- Some evidence that it scales effectively
- May be difficult to assess the degree of approximation

Case study: A task on Excel

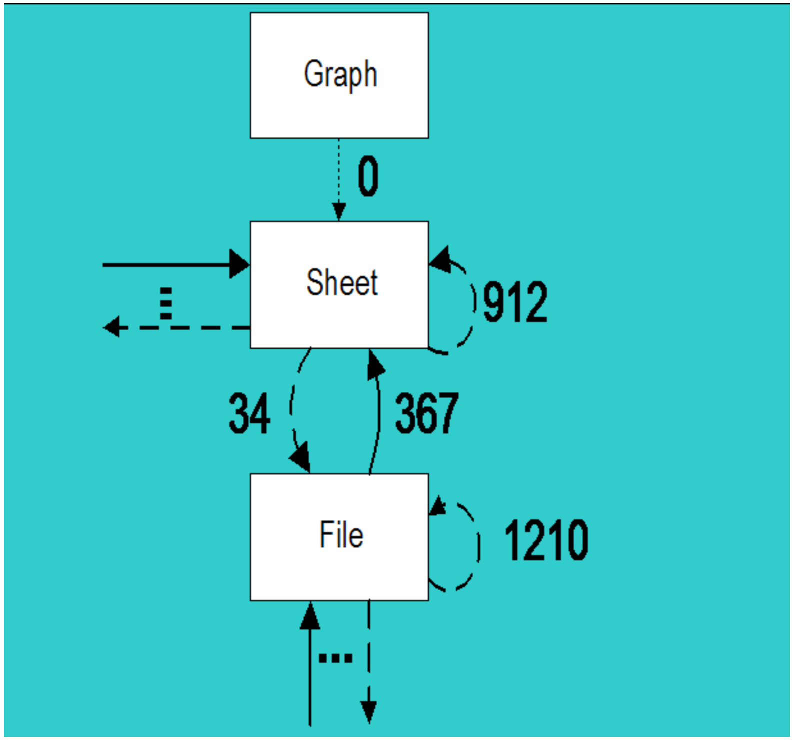
- A series of approximate tools were used by a Microsoft engineer to perform an experimental reengineering task on Excel
- The task involved the identification and extraction of components from Excel
- Excel (then) comprised about 1.2 million lines of C source
  - About 15,000 functions spread over ~400 files

The process used



An initial Reflexion Model

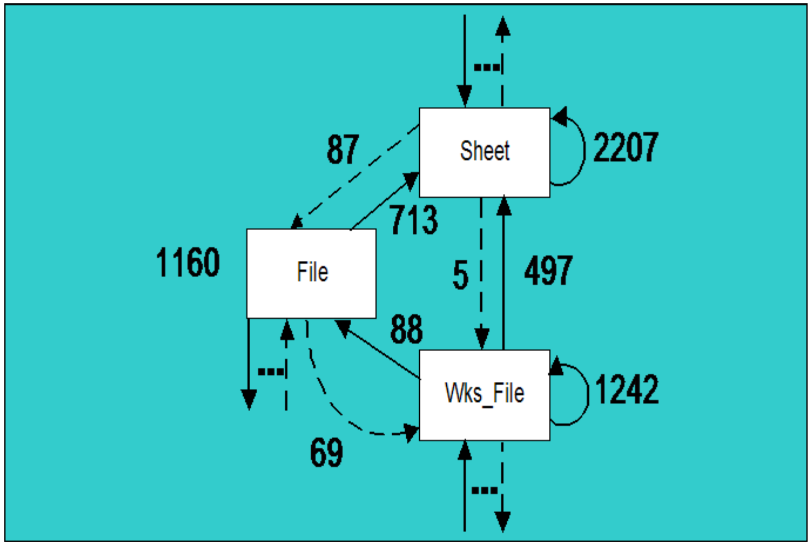
- The initial Reflexion Model computed had 15 convergences, 83, divergences, and 4 absences
- It summarized 61% of calls in source model

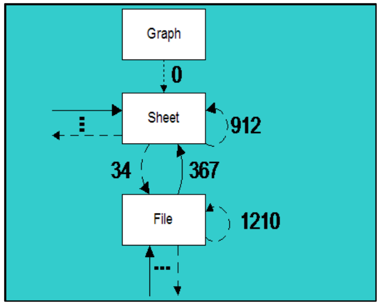


An iterative process

- Over a 4+ week period
- Investigate an arc
- Refine the map
  - Eventually over 1000 entries
- Document exceptions
- Augment the source model
  - Eventually, 119,637 interactions

A refined Reflexion Model





- A later Reflexion Model summarized 99% of 131,042 call and data interactions
- This approximate view of approximate information was used to reason about, plan and automate portions of the task

Results

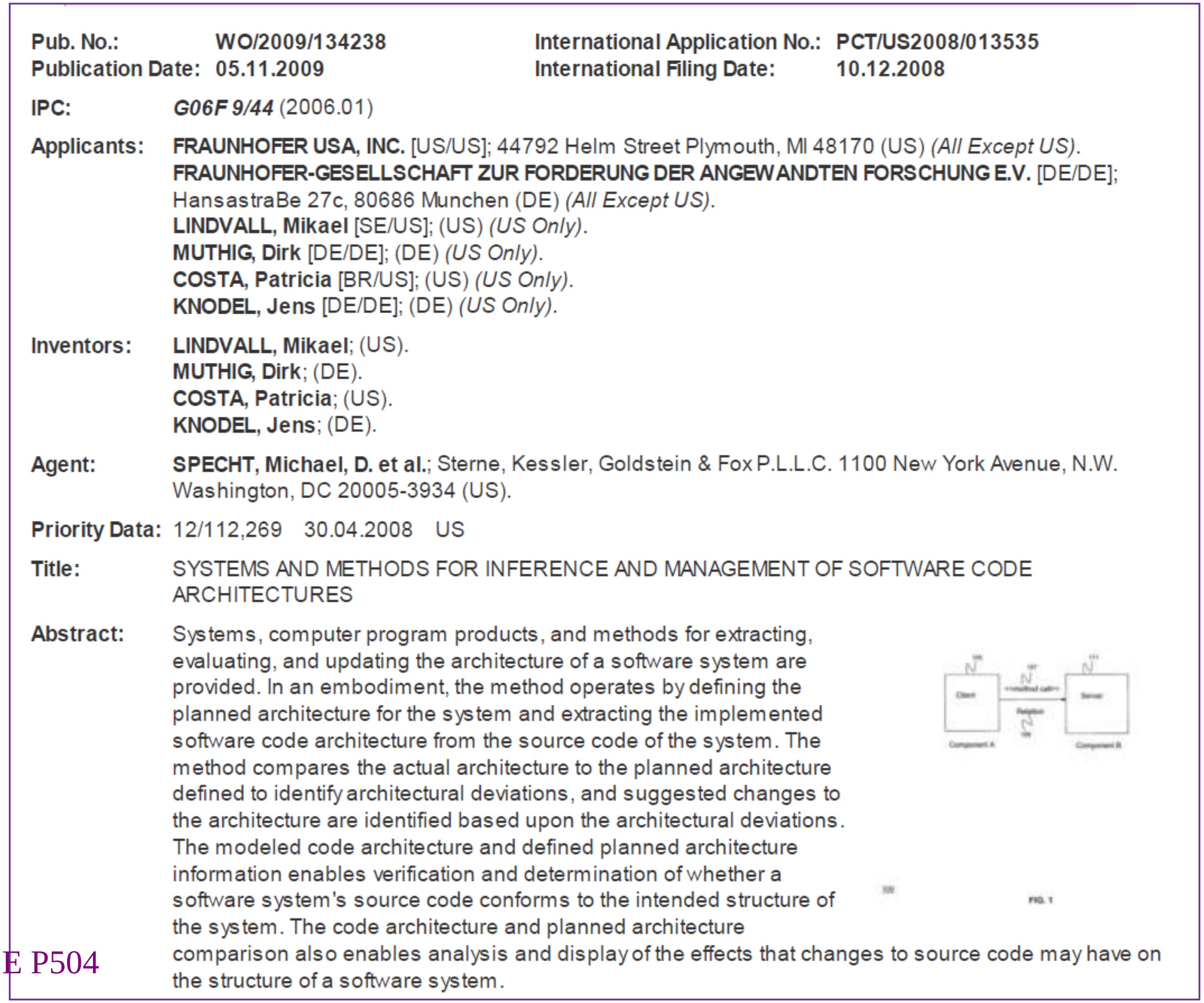
- Microsoft engineer judged the use of the Reflexion Model technique successful in helping to understand the system structure and source code

  “Definitely confirmed suspicions about the structure of Excel. Further, it allowed me to pinpoint the deviations. It is very easy to ignore stuff that is not interesting and thereby focus on the part of Excel that I want to know more about.” — Microsoft A.B.C. (anonymous by choice) engineer

Open questions

- How stable is the mapping as the source code changes?
- What if you don’t have a high-level model?
- How come it’s not used much at all?
- …

Imitation and flattery



Pub. No.: **WO/2009/134238** **International Application No.:** **PCT/US2008/013535**
Publication Date: **05.11.2009** **International Filing Date:** **10.12.2008**

IPC: ***G06F 9/44*** (2006.01)

Applicants: **FRAUNHOFER USA, INC.** [US/US]; 44792 Helm Street Plymouth, MI 48170 (US) *(All Except US)*.
FRAUNHOFER-GESELLSCHAFT ZUR FORDERUNG DER ANGEWANDTEN FORSCHUNG E.V. [DE/DE]; HansastraBe 27c, 80686 Munchen (DE) *(All Except US)*.
LINDVALL, Mikael [SE/US]; (US) *(US Only)*.
MUTHIG, Dirk [DE/DE]; (DE) *(US Only)*.
COSTA, Patricia [BR/US]; (US) *(US Only)*.
KNODEL, Jens [DE/DE]; (DE) *(US Only)*.

Inventors: **LINDVALL, Mikael**; (US).
MUTHIG, Dirk; (DE).
COSTA, Patricia; (US).
KNODEL, Jens; (DE).

Agent: **SPECHT, Michael, D. et al.**; Sterne, Kessler, Goldstein & Fox P.L.L.C. 1100 New York Avenue, N.W. Washington, DC 20005-3934 (US).

Priority Data: 12/112,269 30.04.2008 US

Title: SYSTEMS AND METHODS FOR INFERENCE AND MANAGEMENT OF SOFTWARE CODE ARCHITECTURES

Abstract: Systems, computer program products, and methods for extracting, evaluating, and updating the architecture of a software system are provided. In an embodiment, the method operates by defining the planned architecture for the system and extracting the implemented software code architecture from the source code of the system. The method compares the actual architecture to the planned architecture defined to identify architectural deviations, and suggested changes to the architecture are identified based upon the architectural deviations. The modeled code architecture and defined planned architecture information enables verification and determination of whether a software system's source code conforms to the intended structure of the system. The code architecture and planned architecture comparison also enables analysis and display of the effects that changes to source code may have on the structure of a software system.

Questions?