Software Design and Evolution

11. Dynamic Analysis

Jorge Ressia
Roadmap

- Motivation
- Sources of Runtime Information
- Dynamic Analysis Techniques
- Advanced Dynamic Analysis Techniques
- Dynamic analysis in a Reverse Engineering Context
- What can we achieve with all this?
- Conclusion
Roadmap

> Motivation
> Sources of Runtime Information
> Dynamic Analysis Techniques
> Advanced Dynamic Analysis Techniques
> Dynamic analysis in a Reverse Engineering Context
> What can we achieve with all this?
> Conclusion
What does this program do?

Source code can be hard to read and understand.
This is a legal C program, an extreme case of course.

This is intentionally obfuscated code. Obfuscation is a set of transformations that preserve the behavior of the program but make the internals hard to reverse-engineer.

- Programming contests
- Prevent reverse engineering
Finding Features

Software Feature:
A distinguishing characteristic of a software item.

IEEE 829
Bug reports often expressed in terms of Features.

I can’t add new contacts!!!

The software engineer needs to maintain a mental map between features and the parts of the code that implement them. Features are not implemented in one class. Their implementation spreads out over lots of classes. The behavior consist of objects that collaborate at runtime.

“Change requests and bug reports are usually expressed in a language that reflects the features of a system”

[Mehta and Heinemann 2002]
The software engineer needs to maintain a mental map between features and the parts of the code that implement them. Features are not implemented in one class. Their implementation spreads out over lots of classes. The behavior consist of objects that collaborate at runtime.

“Change requests and bug reports are usually expressed in a language that reflects the features of a system”

[Mehta and Heinemann 2002]
Feature-Centric Reverse Engineering

Software System

Users

Software developer
Feature-Centric Reverse Engineering

Software System

Users

Software developer
Finding Features

I have a system and I need to find the features. Which part of the system belongs to which features?
What is Dynamic Analysis?

The investigation of the properties of a *running* software system over one or more executions

(Static analysis examines the program code alone)
Why Dynamic Analysis?

Gap between run-time structure and code structure in OO programs

"Trying to understand one [structure] from the other is like trying to understand the dynamism of living ecosystems from the static taxonomy of plants and animals, and vice-versa."

-- Erich Gamma et al., Design Patterns
Roadmap

> Motivation
> **Sources of Runtime Information**
> Dynamic Analysis Techniques
> Advanced Dynamic Analysis Techniques
> Dynamic analysis in a Reverse Engineering Context
> What can we achieve with all this?
> Conclusion
Runtime Information Sources

Many possibilities: hardware monitoring, tracing method execution, values of variables, memory usage etc...

External view
execute program and watch it from outside

Internal view
instrument program and watch it from inside
Program output, UI (examine behavior, performance, …)

Analyze used resources
  CPU and memory usage (top)
  Network usage (netstat, tcpdump)
  Open files, pipes, sockets (lsof)

Examine logs (syslog, web logs, stdout/stderr, …)
Many different tools are based on tracing: execution profilers, test coverage analyzers, tools for **reverse engineering**…

- Logs: Single points in the execution.
- Stack trace: snapshot of the current stack
- Debugger: interactive, allows one to step into future method executions. Not persistent.
- Tracing: full history of all method executions
Execution Tracing

How can we capture “full” OO program execution?

Trace entry and exit of methods

Additional information:
- receiver and arguments
- class instantiations
- ...?

For now we consider this approach

Sequence and nesting: construct tree structure
Additional:
- receiver and arguments
- Return values
- Object creation
- Current feature
- Distinguish process (analyze concurrency properties)

Object referencing relationships not captured.
Object graph at particular point in time cannot be reconstructed, now, how it evolves
Tracing Techniques

Instrumentation approaches
  — Sourcecode modification
  — Bytecode modification
  — Wrapping methods (Smalltalk)

Simulate execution (using debugger infrastructure)

Sampling

At the VM level
  — Execution tracing by the interpreter
  — (Dynamic recompilation, JIT)

Simulate execution: slow, but very precise control possible
Sampling: mainly used for profiling
Dynamic recompilation:
  - control optimizations: compile hot blocks/paths/procedures to machine code
  - data optimizations: garbage collection: move objects for locality
Technical Challenges

- Instrumentation influences the behavior of the execution
- Overhead: increased execution time
- Large amount of data

- Code also used by the tracer, library and system classes cannot be instrumented
  -> Trace at the VM level
  -> Scope instrumentation (Changeboxes)
Roadmap

> Motivation
> Sources of Runtime Information
> **Dynamic Analysis Techniques**
> Advanced Dynamic Analysis Techniques
> Dynamic analysis in a Reverse Engineering Context
> What can we achieve with all this?
> Conclusion
I have a system and I need to find the features. Which part of the system belongs to which features?
“...debugging statements stay with the program; debugging sessions are transient. ”

Kernigham and Pike

public class Main {

    public static void main(String[] args) {
        Clingon aAlien = new Clingon();
        System.out.println("in main ");
        aAlien.spendLife();
    }
}

Inserting log statements into your code is a low-tech method for debugging it.
It may also be the only way because debuggers are not always available or applicable.
This is often the case for distributed applications.
“...debugging statements stay with the program; debugging sessions are transient.”

_Steven Kerningham and Peter Pike_

```java
public class Main {
    public static void main(String[] args) {
        Clingon aAlien = new Clingon();
        System.out.println("in main");
        aAlien.spendLife();
    }
}
```

Inserting log statements into your code is a low-tech method for debugging it.
It may also be the only way because debuggers are not always available or applicable.
This is often the case for distributed applications.
> become: function
> Method Wrappers
> Anonymous Classes
public class DebugProxy implements java.lang.reflect.InvocationHandler {

    private Object obj;

    public static Object newInstance(Object obj) {
        return java.lang.reflect.Proxy.newProxyInstance(
            obj.getClass().getClassLoader(),
            obj.getClass().getInterfaces(),
            new DebugProxy(obj));
    }

    public Object invoke(Object proxy, Method m, Object[] args)
        throws Throwable {
        .... Feature data gathering ...
        return m.invoke(obj, args);
        System.out.println("after method " + m.getName());
    }
}

http://docs.oracle.com/javase/1.3/docs/guide/reflection/proxy.html
In the pointcut-advice (PA) mechanism for aspect-oriented programming, as embodied in AspectJ and others, cross-cutting behavior is defined by means of pointcuts and advices. Points during execution at which advices may be executed are called (dynamic) join points. A pointcut identifies a set of join points, and an advice is the action to be taken at a join point matched by a pointcut. An aspect is a module that encompasses a number of pointcuts and advices. In AspectJ, the decision of whether or not to use an aspect within a program is done at build time; if so, the aspect has global scope, i.e. it sees all join points of the program execution. Restricting the scope of an aspect can be done by introducing conditions in the pointcut definitions.

AOP

Cross-cutting concerns

Logging

Security
AOP

Diagram:

- Advice
- Pointcut
- JoinPoints
- Program Execution
AOP

Diagram:

- Aspect Program
- Aspect Specification
- Aspect Weaver
- Modified Code
- Base Code

26
public class HelloWorld {

    public static void say(String message) {
        System.out.println(message);
    }

    public static void sayToPerson(String message, String name) {
        System.out.println(name + "", " + message);
    }
}

public aspect Example {
    pointcut callSayMessage() :
        call(public static void HelloWorld.say*(..));
    before() : callSayMessage() {
        System.out.println("Good day!");
    }
    after() : callSayMessage() {
        System.out.println("Thank you!");
    }
}
public aspect FeatureAnalysis {
    pointcut callMessage() :
        call(public * com.mycompany..*.*(..));

    before() : callMessage() {
        ... save feature information ...
    }
}
public aspect FeatureAnalysis {
    pointcut executeMessage() :
        execute(public * com.mycompany..*.*(..));

    before() : executeMessage() {
        ... save feature information ...
    }
}

So what's the difference between these join points? Well, there are a number of differences:

Firstly, the lexical pointcut declarations within and withincode match differently. At a call join point, the enclosing code is that of the call site. This means that \texttt{call(void m()) && withincode(void m())} will only capture directly recursive calls, for example. At an execution join point, however, the program is already executing the method, so the enclosing code is the method itself: \texttt{execution(void m()) && withincode(void m())} is the same as \texttt{execution(void m())}.

Secondly, the call join point does not capture super calls to non-static methods. This is because such super calls are different in Java, since they don't behave via dynamic dispatch like other calls to non-static methods.

The rule of thumb is that if you want to pick a join point that runs when an actual piece of code runs (as is often the case for tracing), use \texttt{execution}, but if you want to pick one that runs when a particular \texttt{signature} is called (as is often the case for production aspects), use \texttt{call}. 
AspectJ pointcuts

call(MethodPattern)  withincode(MethodPattern)
execution(MethodPattern) withincode(ConstructorPattern)
get(FieldPattern)    cflow(Pointcut)
set(FieldPattern)    cflowbelow(Pointcut)
call(ConstructorPattern)  this(Type or Id)
execution(ConstructorPattern) target(Type or Id)
initialization(ConstructorPattern) args(Type or Id, ...)
preinitialization(ConstructorPattern)  PointcutId(TypePattern or Id, ...)
staticinitialization(TypePattern)  if(BooleanExpression)
handler(TypePattern)       ! Pointcut
adviceexecution()        Pointcut0 && Pointcut1
within(TypePattern)       Pointcut0 || Pointcut1

This is just an example for AspectJ, there are many other aspect languages with many different pointcuts with different objectives.
Operational Decomposition

McAffer - CodA - Meta-level Programming with CodA - ECOOP 1995
Operational Decomposition

Iguana C++, IguanaJ
Bifröst
AOP
EAOP
AspectJ tracematches
We need to find which statements belong to which feature.
Sub-method Feature Analysis

Bytecode Instrumentation
Smalltalk
Example: Number>>asInteger

> Smalltalk code:

```
Number>>asInteger

"Answer an Integer nearest the receiver toward zero."

^self truncated
```

> Symbolic Bytecode

```
9 <70> self
10 <D0> send: truncated
11 <7C> returnTop
```
Example: Step by Step

>  9  <70>  self
   — The receiver (self) is pushed on the stack

> 10  <D0>  send: truncated
   — Bytecode 208: send literal selector 1
   — Get the selector from the first literal
   — Start message lookup in the class of the object that is on top of the stack
   — Result is pushed on the stack

> 11  <7C>  returnTop
   — Return the object on top of the stack to the calling method
ByteSurgeon

- Library for bytecode transformation in Smalltalk
- Full flexibility of Smalltalk Runtime
- Provides high-level API
- For Pharo, but portable

- Runtime transformation needed for
  - Adaptation of running systems
  - Tracing / debugging
  - New language features (MOP, AOP)
Example: Logging

> Goal: logging message send.
> First way: Just edit the text:

```python
example
    self test.
```

```python
example
    Transcript show: ‘sending #test’.
    self test.
```
Logging with ByteSurgeon

> Goal: Change the method without changing program text
> Example:

```
(Example>>#example)instrumentSend: [:send |
  send insertBefore:
  'Transcript show: "sending #test"'.
]
```
(Example>>#example)instrumentSend: [:send |
  send insertBefore:
  'Transcript show: ''sending #test'' ']

Example >> #example

Class

Name of Method

>>: - takes a name of a method
  - returns the CompiledMethod object
Logging: Step by Step

(Example>>#example)instrumentSend: [:send |
  send insertBefore:
    Transcript show: ‘‘sending #test’’ ‘‘.
]

> instrumentSend:
  — takes a block as an argument
  — evaluates it for all send bytecodes
The block has one parameter: send

> It is executed for each send bytecode in the method
Objects describing bytecode understand how to insert code

— insertBefore
— insertAfter
— replace
The code to be inserted.

> Double quoting for string inside string
  - `Transcript show: 'sending #test'`
Inside ByteSurgeon

> Uses IRBuilder internally

> Transformation (Code inlining) done on IR
ByteSurgeon Usage

> On Methods or Classes:

MyClass instrument: [.... ].
(MyClass>>#myMethod) instrument: [.... ].

> Different instrument methods:
  — instrument:
  — instrumentSend:
  — instrumentTempVarRead:
  — instrumentTempVarStore:
  — instrumentTempVarAccess:
  — same for InstVar
> Goal: extend a send with after logging

``` example
def self test:
    Logger logSendTo: self.
```
Advanced ByteSurgeon

> With ByteSurgeon, something like:

```
(Example>>#example)instrumentSend: [:send |
  send insertAfter:
    'Logger logSendTo: ?'.
]
```

> How can we access the receiver of the send?
> Solution: Metavariable
Advanced ByteSurgeon

> With Bytesurgeon, something like:

```
(Example>>#example)instrumentSend: [:send |
  send insertAfter:
  'Logger logSendTo: <meta: #receiver>'
]
```

> How can we access the receiver of the send?
> Solution: Metavariable
Java

www.javassist.org
http://commons.apache.org/bcel/
http://asm.objectweb.org/
Bytecode Manipulation

> Java
  — Javassist
    - reflection
    - RMI
  — BCEL
    - Decompiling, Obfuscation, and Refactoring
    - AspectJ
    - FindBugs
  — ASM
    - Groovy
    - AspectWerkz

www.javassist.org
http://commons.apache.org/bcel/
http://asm.objectweb.org/
class Point {
    int x, y;
    void move(int dx, int dy) { x += dx; y += dy; }
}
ClassPool pool = ClassPool.getDefault();
CtClass cc = pool.get("Point");
CtMethod m = cc.getDeclaredMethod("move");
m.insertBefore("{ System.out.println($1);
System.out.println($2); }");
cc.writeFile();
class Point {
    int x, y;
    void move(int dx, int dy) {
        System.out.println(dx); System.out.println(dy);
        x += dx; y += dy;
    }
}
Javassist - Edit Body

CtMethod cm = ... ;
cm.instrument(
    new ExprEditor() {
        public void edit(MethodCall m)
            throws CannotCompileException
        {
            if (m.getClassName().equals("Point")
                && m.getMethodName().equals("move")
                m.replace("{ $1 = 0; $_ = $proceed($$); }");
        }
    });

searches the method body represented by cm and replaces all calls to move() in class Point with a block:

* { $1 = 0; $_ = $proceed($$); }
Problems with Bytecode Instrumentation

> Bytecode is not a good meta model

> Lost of management infrastructure is needed
  — Hook composition
  — Synthesized elements (hooks) vs original code
  — Mapping to source elements

> Bytecode is optimized
  — e.g. no ifTrue:
Simulation
Parsing and Interpretation

> First step: *Parse bytecode*
  — enough for easy analysis, pretty printing, decompilation

> Second step: *Interpretation*
  — needed for simulation, complex analysis (e.g., profiling)

> Pharo provides frameworks for both:
  — InstructionStream/InstructionClient (parsing)
  — ContextPart (Interpretation)
The InstructionStream Hierarchy

InstructionStream
  ContextPart
    BlockContext
    MethodContext
  Decompiler
  InstructionPrinter
  InstVarRefLocator
  BytecodeDecompiler
**InstructionStream**

> Parses the byte-encoded instructions

> **State:**

— pc: program counter

— sender: the method (bad name!)

```ruby
Object subclass: #InstructionStream
instanceVariableNames: 'sender pc'
classVariableNames: 'SpecialConstants'
poolDictionaries: ''
category: 'Kernel-Methods'
```
Usage

> Generate an instance:

```
instrStream := InstructionStream on: aMethod
```

> Now we can step through the bytecode with:

```
instrStream interpretNextInstructionFor: client
```

> Calls methods on a client object for the type of bytecode, e.g.

- `pushReceiver`
- `pushConstant: value`
- `pushReceiverVariable: offset`
InstructionClient

> Abstract superclass
  — Defines empty methods for all methods that InstructionStream calls on a client

> For convenience:
  — Clients don’t need to inherit from this class

```
Object subclass: #InstructionClient
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Kernel-Methods'
```
Example: A test

```smalltalk
InstructionClientTest>>testInstructions
   "just interpret all of methods of Object"
   | methods client scanner |
   
   methods := Object methodDict values.
   client := InstructionClient new.
   
   methods do: [:method |
      scanner := (InstructionStream on: method).
      [scanner pc <= method endPC] whileTrue: [
         self shouldnt:
            [scanner interpretNextInstructionFor: client]
            raise: Error.
      ].
   ].
```
Example: Printing Bytecode

> InstructionPrinter:
  — Print the bytecodes as human readable text

> Example:
  — print the bytecode of $Number>>asInteger$:

```plaintext
String streamContents:
[:str | (InstructionPrinter on: Number>>#asInteger)
  printInstructionsOn: str ]

'9 <70> self
10 <D0> send: truncated
11 <7C> returnTop
'
```
InstructionPrinter

> Class Definition:

```smalltalk
InstructionClient subclass: #InstructionPrinter
  instanceVariableNames: 'method scanner stream indent'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Kernel-Methods'
```
> Main Loop:

```
InstructionPrinter>>printInstructionsOn: aStream
    "Append to the stream, aStream, a description
    of each bytecode in the instruction stream."
    | end |
    stream := aStream.
    scanner := InstructionStream on: method.
    end := method endPC.
    [scanner pc <= end]
        whileTrue: [scanner interpretNextInstructionFor: self]
```
InstructionPrinter

> Overwrites methods from InstructionClient to print the bytecodes as text
> e.g. the method for pushReceiver

```
InstructionPrinter>>pushReceiver
    "Print the Push Active Context's Receiver on Top Of Stack bytecode."

    self print: 'self'
```
Example: InstVarRefLocator

```smalltalk
InstructionClient subclass: #InstVarRefLocator
    instanceVariableNames: 'bingo'
    classVariableNames: '
    poolDictionaries: '
    category: 'Kernel-Methods'

InstVarRefLocator>>interpretNextInstructionUsing: aScanner
    bingo := false.
    aScanner interpretNextInstructionFor: self.
    ^bingo

InstVarRefLocator>>popIntoReceiverVariable: offset
    bingo := true

InstVarRefLocator>>pushReceiverVariable: offset
    bingo := true

InstVarRefLocator>>storeIntoReceiverVariable: offset
    bingo := true
```
InstVarRefLocator

> Analyse a method, answer true if it references an instance variable

```smalltalk
CompiledMethod>>hasInstVarRef
    "Answer whether the receiver references an instance variable."

| scanner end printer |

scanner := InstructionStream on: self.
printer := InstVarRefLocator new.
end := self endPC.

[scanner pc <= end] whileTrue:
    [ (printer interpretNextInstructionUsing: scanner)
      ifTrue: [^true]. ].
^false
```
InstVarRefLocator

> Example for a simple bytecode analyzer

> Usage:

```small
aMethod hasInstVarRef
```

> (has reference to variable testSelector)

```small
(TestCase>>#debug) hasInstVarRef → true
```

> (has no reference to a variable)

```small
(Integer>>#+) hasInstVarRef → false
```
> Sometimes we need more than parsing
   — “stepping” in the debugger
   — system simulation for profiling

InstructionStream subclass: #ContextPart
  instanceVariableNames: 'stackp'
  classVariableNames: 'PrimitiveFailToken QuickStep'
  poolDictionaries: ''
  category: 'Kernel-Methods'
Simulation

> Provides a complete Bytecode interpreter

> Run a block with the simulator:

```
(ContextPart runSimulated: [3 factorial])
```

6
What is the big picture?

Source code

Bytecode
What is the big picture?

Source code

? 

Bytecode
AST Instrumentation

code → Scanner / Parser → AST → Semantic Analysis → AST → Code Generation → Bytecode
> Marcus Denker
   — Pharo Smalltalk
   — Geppetto 2
   — Phersephone

> Using Partial Behavioral Reflection Model
   — Reflex, Tanter etal.
> AST: Abstract Syntax Tree
   — Encodes the Syntax as a Tree
   — No semantics yet!
   — Uses the RB Tree:
     - Visitors
     - Backward pointers in ParseNodes
     - Transformation (replace/add/delete)
     - Pattern-directed TreeRewriter
     - PrettyPrinter

```
RBProgramNode
RBDoItNode
RBMethoNode
RBReturnNode
RBSequenceNode
RBValueNode
   RBArrayNode
   RBAssignmentNode
   RBBlockNode
   RBCascadeNode
   RBLITERAL
   RBMessageNode
   RBOptimizedNode
   RBVariableNode
```
Reflectivity

Links for AST nodes
Reflectivity

meta-object

activation condition

source code (AST)

links

Denker 2008

Links for AST nodes
Reflectivity

meta-object

activation condition

source code (AST)

links

Links for AST nodes
Reflectivity

Links for AST nodes
Reflective Architecture

Bifröst
Organize the meta-level
Explicit meta-object
Structural and Behavioral reflection
Partial Reflection
Unanticipation
Selective Reifications
No VM requirements
Bifröst

Class

Meta-object

Object
Evolved Object

Class

Meta-object

Evolved Object
Feature Analysis

| aMetaObject |
aMetaObject := BFBehavioralMetaObject new.
aMetaObject
    when: (ASTExecutionEvent new)
    do: [ ... feature information gathering ...].
aMetaObject bindTo: self

http://scg.unibe.ch/research/bifrost
Implicit Problems

> Partial Reflection
  — We want to reflect on portions of the system

> Unanticipation
  — We want to reflect without having to anticipate where in the system

> Selective Reifications
  — We want to have runtime reifications available

> Composition
  — We want to be able to compose different analysis
Roadmap

> Motivation
> Sources of Runtime Information
> Dynamic Analysis Techniques
> **Advanced Dynamic Analysis Techniques**
> Dynamic analysis in a Reverse Engineering Context
> What can we achieve with all this?
> Conclusion
Simultaneous Feature Analysis
Simultaneous Feature Analysis

Dynamic Scope
Simultaneous Feature Analysis

Legend:
- Objects
- Dynamic Scope
Simultaneous Feature Analysis

Legend

- objects
- login feature
- printing feature
Dynamically scoped aspects

\texttt{deploy(a)\{block\}}


whereby the aspect instance a sees all join points produced in the dynamic extent of the execution of block
Dynamically scoped aspects

> AspectScheme

> CaesarJ

> AspectS
Deployment Strategies

\[ \text{depl}(a, \delta\langle c, d, f \rangle, e) \]

- \( a \) is an aspect
- \( \delta \) is the strategy
  - \( c \): stack propagation function
  - \( d \): object propagation function
  - \( f \): joint point filter
- \( e \) is an expression

Example from a car factory. Only some cars with a special package should get this adaptation.

```java
deploy[true,-,if(cars_sp.contains(jp.args(1)))](sp){
    next.process(batch);
}
```
Scoping Strategies

Propagation and Activation Problem

Dynamic Scoping

Prisma
Dynamic Scoping

Bifrost
Simultaneous Feature Analysis
Dynamic Scoping

> Prisma
  — Execution Reification
  — Reflective Architecture
  — Execution composed of meta-objects
  — Reuse of Execution
  — Execution is not tied to threads
  — Broadening of Scope
  — Dynamic change of conditions
Execution levels
Execution levels

Denker et al. Meta Context

Diagram showing levels of execution with metameta-object, meta-object, meta, operation, base, link, and level 0 and 1.
> Polymorphic Bytecode Instrumentation (PBI)

— Dynamic dispatch amongst several, possibly independent instrumentations
— Instrumentations are saved and indexed by a version identifier
— Implemented over BCEL
— JVM
— Scala, JRuby, etc.
— Execution levels
— Monitoring
— Mixin Layers
— Promising performance
Scoping Dimensions

> **Nature of Adaptation.** A structural adaptation depicts the addition or change of a structural element, like refinements in Classboxes. A behavioral adaptation execute some action when specific runtime events are triggered.

> **Scoped Definition.** The boundaries of the scope are defined by the entry and exit points. These boundaries can be implicit or explicit.

> **Scope Information Exposure.** Some approaches allow to bind a value to a variable which is bound to the scope. This trait is particularly important to provide reusable adaptations.

> **Scope Binding.** There are two binding dimensions. The adaptation can be defined at compile time or at runtime, this is call binding time. The binding mode describes wether an adaptation can be undone/redone during execution, if so the binding mode is said to be dynamic otherwise is static.

> **Thread Locality.** The scope can be defined locally to a single thread. For example, cflow in AspectJ is by default thread local, while tracematch in AspectJ extension is by default global.
Roadmap

> Motivation
> Sources of Runtime Information
> Dynamic Analysis Techniques
> Advanced Dynamic Analysis Techniques
> Dynamic analysis in a Reverse Engineering Context
> What can we achieve with all this?
> Conclusion
In this course you have been introduced to the concepts of reverse engineering. Reverse engineering abstracts high level abstractions that support system understanding [Chikofsky and Cross, 1990].

"Object-oriented language characteristics such as inheritance, dynamic binding and polymorphism mean that the behavior of a system can only be determined at runtime." [Jerding 1996, Demeyer2003a]

A static perspective of the system over looks semantic knowledge of the problem domain of a system. The semantic knowledge should not be ignored. We need a way to enrich the static views with information about their intent. Which features do they participate in at runtime? Are they specific to one part of the system, one feature, or is it general functionality that implements some infrastructural functionality?

So let's extend our analysis by incorporating dynamic data captured while executing the features.
In this course you have been introduced to the concepts of reverse engineering. Reverse engineering abstracts high level abstractions that support system understanding [Chikofsky and Cross, 1990].

"Object-oriented language characteristics such as inheritance, dynamic binding and polymorphism mean that the behavior of a system can only be determined at runtime." [Jerding 1996, Demeyer2003a]

A static perspective of the system overlooks semantic knowledge of the problem domain of a system. The semantic knowledge should not be ignored. We need a way to enrich the static views with information about their intent. Which features do they participate in at runtime? Are they specific to one part of the system, one feature, or is it general functionality that implements some infrastructural functionality?

So let's extend our analysis by incorporating dynamic data captured while executing the features.
Dynamic Analysis for Program Comprehension

Post Mortem Analysis of execution traces

Metrics Based Approaches

-Frequency Analysis [Ball, Zaidman]
-Runtime Coupling Metrics based on Web mining techniques to detect key classes in a trace.

[Zaidman 2005]

-High-Level Polymetric Views of Condensed Run-Time Information [Ducasse, Lanza and Bertoulli 2004]

 Query-based approaches
 Recovering high-level views from runtime data
 [Richner and Ducasse 1999]

They define an execution scenario to maximize coverage of the system and ‘preciseness’. To execute all the features.

Frequency analysis - small number of methods are responsible for a large amount of the trace. They focus on call relationships between methods to learn something about a system.

Coupling metrics:

Runtime metrics
- how many methods of a class were invoked during the execution of a system.
- which classes create objects
- Which classes communicate with each other
Traces of execution behavior lead to huge execution traces of tens of thousands of events. This makes them difficult to interpret or to extract high level views. We need techniques to reduce the volume of information without loss of details needed to answer a specific research question. For example: “Which classes and methods implement the save contact feature?”

Wim dePauw [JinSight, De Pauw 1993].

Other compression approaches

Use graph algorithms to detect patterns and reduce the volume of data. Use patterns to learn something about the system behavior.
Dividing a trace into features

Feature 1  Feature 2  Feature n
Feature Identification is a technique to map features to source code.

“A feature is an observable unit of behavior of a system triggered by the user” [Eisenbarth et al. 2003]

Software Reconnaissance [Wilde and Scully]
Run a (1) feature exhibiting scenario and a (2) non-exhibiting scenario and compare the traces. Then browse the source code.

Other researchers had devised variations of software reconnaissance - Antoniol, Eisenberg etc.
Feature Identification is a technique to map features to source code.

“A feature is an observable unit of behavior of a system triggered by the user” [Eisenbarth et al. 2003]

Software Reconnaissance [Wilde and Scully]
Run a (1) feature exhibiting scenario and a (2) non-exhibiting scenario and compare the traces. Then browse the source code.

Other researchers had devised variations of software reconnaissance - Antoniol, Eisenberg etc.
Feature Identification is a technique to map features to source code.

“A feature is an observable unit of behavior of a system triggered by the user” [Eisenbarth et al. 2003]

Software Reconnaissance [Wilde and Scully]
Run a (1) feature exhibiting scenario and a (2) non-exhibiting scenario and compare the traces. Then browse the source code.

Other researchers had devised variations of software reconnaissance - Antoniol, Eisenberg etc.
1) How are classes related to features?
2) How are features related to classes?
3) How are features related to each other?

We define a Feature-Affinity metric to distinguish between various levels of characterization of classes.
Feature-Centric Analysis: 3 Complementary Perspectives

1) How are classes related to features?
2) How are features related to classes?
3) How are features related to each other?

We define a Feature-Affinity metric to distinguish between various levels of characterization of classes.
1) How are classes related to features?
2) How are features related to classes?
3) How are features related to each other?

We define a Feature-Affinity metric to distinguish between various levels of characterization of classes.
Feature-Centric Analysis: 3 Complementary Perspectives

1) How are classes related to features?
2) How are features related to classes?
3) How are features related to each other?

We define a Feature-Affinity metric to distinguish between various levels of characterization of classes.
Dynamix - A Model for Dynamic Analysis
DynaMoose - An Environment for Feature Analysis
DynaMoose - An Environment for Feature Analysis
DynaMoose - An Environment for Feature Analysis
DynaMoose - An Environment for Feature Analysis
DynaMoose - An Environment for Feature Analysis
DynaMoose - An Environment for Feature Analysis
Here we see the feature views (of classes)

Our question was “Which classes participate in the addContacts feature?”
Which methods participate in the feature ‘addContacts()’?

22 single feature methods
Object Flow Analysis

Method execution traces do not reveal how
  ... objects refer to each other
  ... object references evolve

Trace and analyze object flow
  — Object-centric debugger: Trace back flow from errors to code that produced the objects
  — Detect object dependencies between features
Roadmap

> Motivation
> Sources of Runtime Information
> Dynamic Analysis Techniques
> Advanced Dynamic Analysis Techniques
> Dynamic analysis in a Reverse Engineering Context
> **What can we achieve with all this?**
> Conclusion
What is the future?

Live Feature Analysis
Live Feature Analysis

Feature tagger
meta-object

tags node with
feature annotation
on execution

source code
(AST)
What is the future?

Object Centric Debugging
Object Centric Debugging

Object Centric Debugging

http://scg.unibe.ch/research/bifrost/OCD

Object Centric Debugging

http://scg.unibe.ch/research/bifrost/OCD

Object Centric Debugging

**Object>>haltAtNextMessage**

```smalltalk
| aMetaObject |
aMetaObject := BFBehavioralMetaObject new.
aMetaObject
  when: (BFMessageReceiveEvent new)
  do: [ self metaObject unbindFrom: self.
      TransparentBreakpoint signal ].
aMetaObject bindTo: self
```

125
What is the future?

MetaSpy
Domain-specific Profiling
CPU time profiling

Mondrian is an open and agile visualization engine. Mondrian describes a visualization using a graph of (possibly nested) nodes and edges. In June 2010 a serious performance issue was raised. Tracking down the cause of the poor performance was not trivial. We first used a standard sample-based profiler. Execution sampling approximates the time spent in an application's methods by periodically stopping a program and recording the current set of methods under executions. Such a profiling technique is relatively accurate since it has little impact on the overall execution. This sampling technique is used by almost all mainstream profilers, such as JProfiler, YourKit, xprof, and hprof.

MessageTally, the standard sampling-based profiler in Pharo Smalltalk, actually describes the execution in terms of CPU consumption and invocation for each method of Mondrian:

54.8% {11501ms} MOCanvas>>drawOn:
54.8% {11501ms} MORoot(MONode)>>displayOn:
30.9% {6485ms} MONode>>displayOn:
 | 18.1% {3799ms} MDEdge>>displayOn:
 | 8.4% {1763ms} MDEdge>>displayOn:
 | 8.0% {1679ms} MOStraightLineShape>>displayOn:
 | 2.6% {546ms} FormCanvas>>line:to:width:color:

We can observe that the virtual machine spent about 54% of its time in the method displayOn: defined in the class MORoot. A root is the unique non-nested node that contains all the nodes of the edges of the visualization. This general profiling information says that rendering nodes and edges consumes a great share of the CPU time, but it does not help in pinpointing which nodes and edges are responsible for the time spent. Not all graphical elements equally consume resources.

Traditional execution sampling profilers center their result on the frames of the execution stack and completely ignore the identity of the object that received the method call and its arguments. As a consequence, it is hard to track down which objects cause the slowdown. For the example above, the traditional profiler says that we spent 30.9% in MONode>>displayOn: without saying which nodes were actually refreshed too often.

Coverage

PetitParser is a parsing framework combining ideas from scannerless parsing, parser combinators, parsing expression grammars and packrat parsers to model grammars and parsers as objects that can be reconfigured dynamically.

Domain-Specific Profiling

Profile

Domain

http://scg.unibe.ch/research/bifrost/metaspyp
Domain-Specific Profiling

Mondrian is an open and agile visualization engine. Mondrian describes a visualization using a graph of (possibly nested) nodes and edges. In June 2010 a serious performance issue was raised. Tracking down the cause of the poor performance was not trivial. We first used a standard sample-based profiler. Execution sampling approximates the time spent in an application's methods by periodically stopping a program and recording the current set of methods under executions. Such a profiling technique is relatively accurate since it has little impact on the overall execution. This sampling technique is used by almost all mainstream profilers, such as JProfiler, YourKit, xprof, and hprof.

MessageTally, the standard sampling-based profiler in Pharo Smalltalk, actually describes the execution in terms of CPU consumption and invocation for each method of Mondrian: 54.8% of 11501ms MOCanvas>>drawOn: 54.8% of 11501ms MORoot(MONode)>>displayOn: 30.9% of 6485ms MONode>>displayOn: 18.1% of 3799ms MEdge>>displayOn: 8.4% of 1763ms MEdge>>displayOn: 8.0% of 1679ms MOStraightLineShape>>displayOn: 2.6% of 546ms FormCanvas>>line:to:width:color:...

We can observe that the virtual machine spent about 54% of its time in the method displayOn: defined in the class MORoot. A root is the unique non-nested node that contains all the nodes of the edges of the visualization. This general profiling information says that rendering nodes and edges consumes a great share of the CPU time, but it does not help in pinpointing which nodes and edges are responsible for the time spent. Not all graphical elements equally consume resources.

Traditional execution sampling profilers center their result on the frames of the execution stack and completely ignore the identity of the object that received the method call and its arguments. As a consequence, it is hard to track down which objects cause the slowdown. For the example above, the traditional profiler says that we spent 30.9% in MONode>>displayOn: without saying which nodes were actually refreshed too often.

Coverage PetitParser is a parsing framework combining ideas from scannerless parsing, parser combinators, parsing expression grammars and packrat parsers to model grammars and parsers as objects that can be reconfigured dynamically.

1 http://forum.world.st/Mondrian-is-slow-next-step-tc2257050.html#a2261116
2 http://www.pharo-project.org/127

Domain-Specific Profiling

http://scg.unibe.ch/research/bifrost/metaspyp

http://forum.world.st/Mondrian-is-slow-next-step-tc2257050.html#a2261116

http://www.pharo-project.org/

Domain-specific Profiling

Profile

http://scg.unibe.ch/research/bifrost/metaspy

Domain-specific Profiling

Profile

http://scg.unibe.ch/research/bifrost/metaspy

Domain-specific Profiling
We claim to be doing dynamic analysis but we keep on going back to the static abstractions.

For dynamic languages the Dilemma is even worst. We are happy to have a dynamic environment like Smalltalk but, in certain way, we are trapped using the static abstractions when we should use the dynamic ones.
Roadmap

> Motivation
> Sources of Runtime Information
> Dynamic Analysis Techniques
> Advanced Dynamic Analysis Techniques
> Dynamic analysis in a Reverse Engineering Context
> What can we achieve with all this?
> Conclusion
Static analyses extract properties that hold for all possible program runs.

Dynamic analysis provides more precise information…but only for the execution under consideration.

Dynamic analysis cannot show that a program satisfies a particular property, but can detect violations of the property.
Conclusions: Pros and Cons

Dependent on input

— Advantage: Input or features can be directly related to execution
— Disadvantage: May fail to exercise certain important paths and poor choice of input may be unrepresentative

Broad scope: dynamic analyses follow long paths and may discover semantic dependencies between program entities widely separated in space and time

However, understanding dynamic behavior of OO systems is difficult

- Large number of executed methods
- Execution paths crosscut abstraction layers
- Side effects
Attribution-ShareAlike 3.0

You are free:
- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions:

**BY:** Attribution. You must attribute the work in the manner specified by the author or licensor.

**Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

[http://creativecommons.org/licenses/by-sa/3.0/](http://creativecommons.org/licenses/by-sa/3.0/)