Prof. Dr. Oscar Nierstrasz
Pascal Gadient, Pooja Rani

Solution
Assignment 09 — 11.11.2020 – v1.2
Static Program Analysis

Please submit this exercise by email to pascal.gadient@inf.unibe.ch before 18 November 2020, 10:15am.

You must submit your code as editable text, i.e., use plain text file(s).

For the second part of this exercise, we use the latest GT release from here. If you already imported some Moose (MSE) models from the previous assignments into your GT environment, please start with a fresh copy of GT to reduce the strains on your computer's random access memory.

First, we have to download and extract the *Weka* dataset, and second, we need to import it into GT. We can perform both tasks using GT's Playground. Be warned: this process will take several minutes depending on your device's CPU and internet connection. We strongly advise you to save the image when the process succeeded to avoid redoing these steps.

The datasets can be downloaded and extracted with the following script:

```
targetFolder := (FileLocator imageDirectory asFileReference / 'models')
ensureCreateDirectory.
archiveFileName := 'weka-3-8.zip'.
archiveUrl := 'https://dl.feenk.com/moose-tutorial/weka/'.
ZnClient new
  url:  archiveUrl, archiveFileName;
  signalProgress:  true;
  downloadTo:  targetFolder.
(ZipArchive new
  readFrom:  targetFolder / archiveFileName)
  extractAllTo:  targetFolder.
```

The sample dataset can be imported with the following script:

```
modelFile := (FileLocator imageDirectory asFileReference / 'models')
  / 'weka-3-8'
  / 'weka-3-8.mse'.
modelWeka := MooseModel new
  importMSEFromFile:  modelFile.
```

Exercise 1: Theory (3.5 pts)

a) What is the difference between static and dynamic code analysis? **Answer:**

Static code analysis is performed without executing any code but only observing source code, whereas dynamic code analysis is performed during the execution of code observing the current state of the application. Both analyses have individual advantages and limitations. For example, static analyses nicely scale vertically (by increasing the CPU speed or count) or horizontally (by adding more devices to the analysis infrastructure), and they can produce accurate results. However, they usually do not consider the executing environment (app configuration, build system, etc.) and the results might not be very comprehensive and suffer from many false positives or false negatives. On the other hand, dynamic analyses are harder to scale, because each analysis needs its own execution environment. On the bright side, they enable analyses that work on any executable code; even code that became available with the execution of the application, e.g., *software updates.*

b) Suppose you want to analyze the code that a method downloads arbitrarily from the internet. Can you perform such an inspection with *static analyses*? If yes, how, if not, why not? **Answer:**

No, because you cannot inspect code fetched at run time with static analysis tools. To circumvent this problem, analysts usually extract the corresponding URL statically and then download and manually analyze that code.

c) Suppose you want to analyze the code that a method downloads arbitrarily from the internet. Can you perform such an inspection with *dynamic analyses*? If yes, how, if not, why not? **Answer:**

Yes, because a dynamic analysis involves the execution of arbitrary code. When the code is under test, it will be executed in the analysis environment and thus download and also execute the requested code.

d) Choose a static analysis scenario where it is crucial to have no type 1 errors (false positives), but type 2 errors (false negatives) can be accepted. Explain for your scenario why you can accept type 2 errors, but no type 1 errors. **Answer:**

Dead code elimination. If such a tool suffers from false positives (dead code that isn't actually dead) it would cause serious problems as soon as it tries to remove affected code, because the code is still in use. On the contrary, false negatives are less harmful, because they only avoid that orphaned code is removed which does not break the system.

e) Choose a statically-typed language, and briefly describe what makes it statically-typed. **Answer:**

In statically-typed languages, types are assigned to variables at compile-time. Such languages provide the benefits of having a static type system at various places, e.g., *in the IDE with live feedback about potential mistakes and improvements, or in Unit testing frameworks.* Java: *The type of each variable must be declared explicitly in the source code. There exist generic types that can represent different types during run time, but Java still validates generic types at compile-time. Generically-typed variables cannot change their type once they have been initialized.*

f) Choose a dynamically-typed language, and briefly describe what makes it dynamically-typed. **Answer:**

In dynamically-typed languages, types are assigned to variables at run time and in many languages they can even change depending on the assigned values. Code in such languages does not need to hold any type information and is more compact (improves readability and efficiency of developers), however the dynamic nature of the execution usually impacts performance and leads to more subtile bugs during execution time, e.g., *code does work but not as expected.* Smalltalk: *The source code does not need any type information. Instead, types that do not match result in a* `doesNotUnderstand` *message.*

g) Is the collection of variable name declarations in the method body of Java's `String.println(String s)` intraprocedural or interprocedural? Explain why. **Answer:**

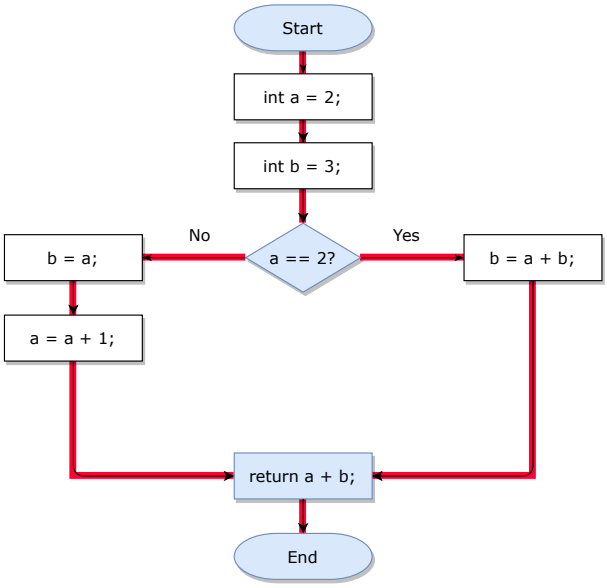
*The scope of intraprocedural analyses remains within a particular method (*e.g., *searching for* `if` *statements within a method), whereas the scope of interprocedural analyses could include the whole project (*e.g., *tracking of variable values through different classes and methods). The collection of variable name declarations in a method is an intraprocedural analysis, because it only requires look ups for entities within a particular method. The look up could be performed by traversing the AST of the method while collecting all* `VariableDeclaration` *nodes.*

Exercise 2: Control flow graphs (6.5 pts)

a) Draw a CFG by hand (or with a flow chart tool) for the following code block: (1.5 pts)

```
int a = 2;
int b = 3;
if (a == 2) {
  b = a + b;
} else {
  b = a++;
}
return a + b;
```

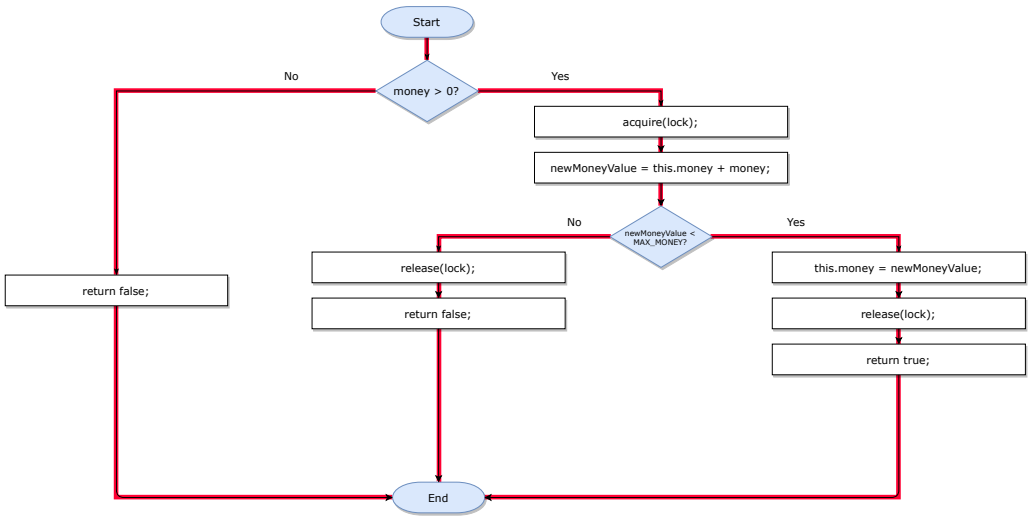
Answer:



b) Draw a CFG by hand (or with a flow chart tool) for the following code block: (2 pts)

```
public boolean addBalances(int money) {
  if (money > 0) {
    acquire(lock);
    int newMoneyValue = this.money + money;
    if (newMoneyValue < MAX_MONEY) {
      this.money = newMoneyValue;
      release(lock);
      return true;
    } else {
      release(lock);
      return false;
    }
  } else {
    return false;
  }
}
```

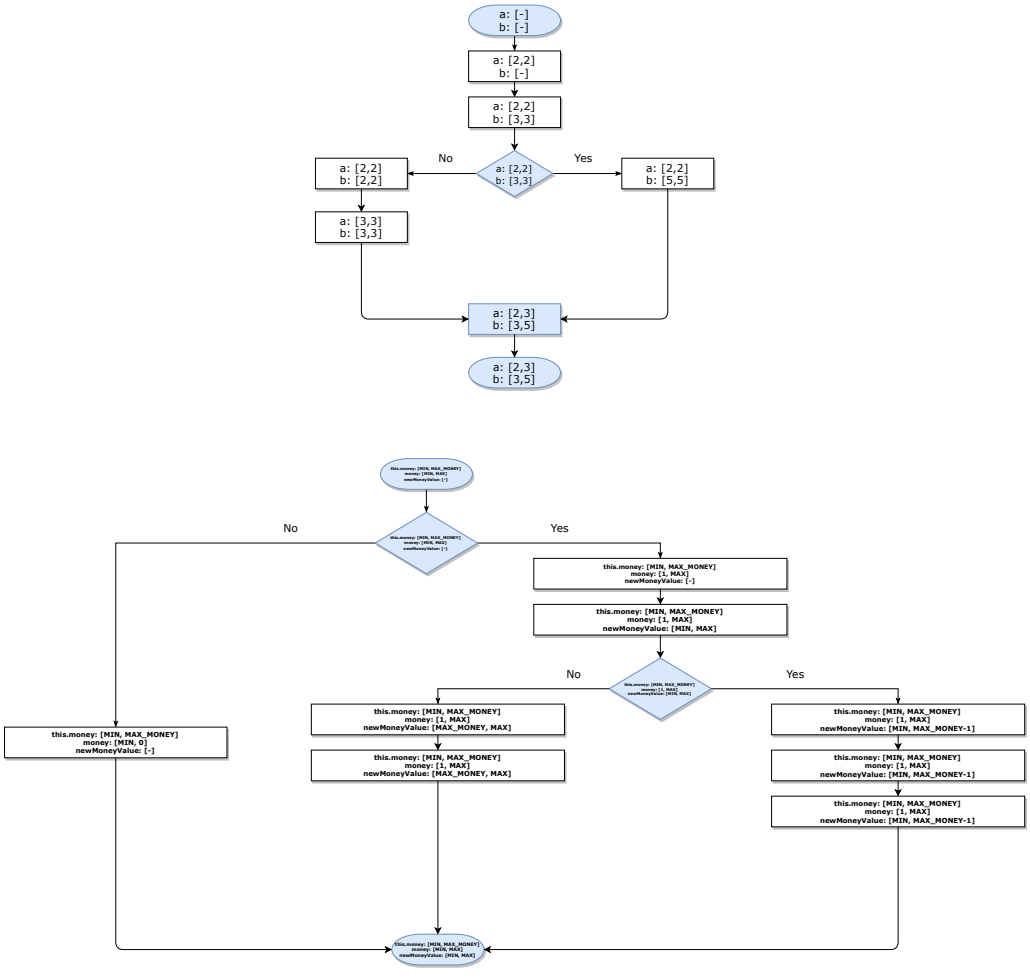
Answer:



c) What is the cyclomatic complexity of both code blocks, i.e., from task a) and task b)? You should use the formula from A08. (1 pt) **Answer:**

The cyclomatic complexity can be calculated with the following formula: M = E - N + 2. In the previous CFGs we colored the edges in red, and the relevant nodes in light blue. The (white) statement nodes do not contribute to the nodes required by the formula, i.e., *they are "invisible" for the cyclo computation. For a) the formula yields M = 4 - 4 + 2 = 2 and for b) it yields M = 5 - 4 + 2 = 3.*

d) Create the interval CFG for both code blocks, i.e., from task a) and task b). You can find an example CFG at the bottom of slide 20 (page 32) in the slide deck of lecture 09 which is available here. (2 pts) **Answer:**



Exercise 3: Template methods (6 pts BONUS)

Find all template methods in `modelWeka` with the help of `#gtASTNodes`, and plot them with `GtMondrian`. Please follow the steps below, one after another. The resulting plot should look similar to Figure 1.

NB: Template methods are abstract methods (in classes or interfaces).

Step 1: Select all methods that are in the namespace scope `weka::core`.

Step 2: Of those methods, find those that are abstract. Use `gtASTNode` for the AST traversal. You can search for `JavaAbstractMethodDeclarationNode` nodes to find the relevant methods.

In this step, GT will build the AST of the relevant entities. This requires several minutes and around 6 GB of RAM on your PC. If you have less RAM the computation will slowdown massively. Note: There is currently a bug in the FAMIX implementation that avoids proper detection of abstract methods in classes.

Step 3: Visualize the found methods in *Step 2* with `GtMondrian`. Use the following parameters:

- shape type: `BlElement` with a size that represents `#children` of each method
- shape geometry: `BlCircle`
- shape background: all methods that have more than three elements in `children` should be in red, the others in gray

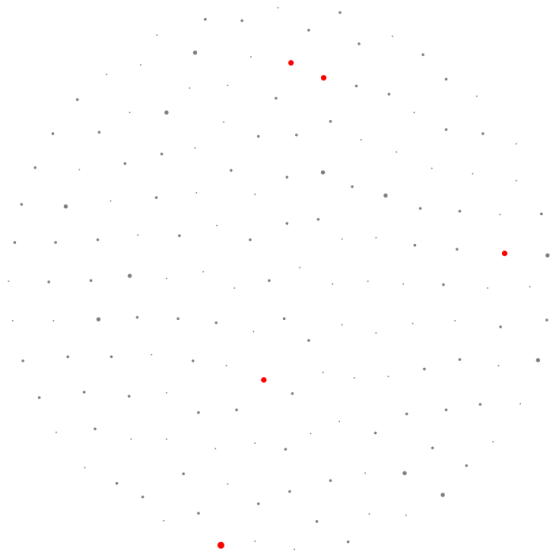


Figure 1: The resulting GtMondrian visualization built with GT

Answer:

```
sampleClasses := modelWeka allModelMethods select:  [ :each |
  each namespaceScope asString beginsWith:
    'weka (Namespace)::core (Namespace)'].

templateMethods := sampleClasses select:  [ :m |
| templateMethod astNode |
astNode := m gtASTNode.
astNode ifNotNil:  [
  (astNode isMemberOf:  JavaAbstractMethodDeclarationNode)
    ifTrue:  [ templateMethod := true.  ]  ].
templateMethod = true.  ].

view := GtMondrian new.
view nodes
  shape:  [ :m |
    BlElement new size:  m children size @ m children size;
    geometry:  BlCircle new;
    background:  ((m children size > 3)
      ifTrue:  [ Color red ]
      ifFalse:  [ Color gray ])];
  with:  templateMethods.
view layout force.
view.
```