

CALL GRAPH CONSTRUCTION FOR SPRING FRAMEWORK

Mugdha Khedkar

Chennai Mathematical Institute, India

Supervisors:

Prof. Dr. Eric Bodden ¹ Dr. Johannes Späth ²

¹Universität Paderborn,
Germany

²Codeshield,
Germany

May 8, 2020

Overview

- 1 Motivation
- 2 Introduction
 - Call Graph Construction
 - Spring Framework
- 3 Observations
- 4 Possible Solution
- 5 Our Solution
- 6 Conclusion
- 7 References

Motivation

- Call graphs give information necessary for compilers to determine whether specific optimizations can be applied
- Software engineering tools like IDEs use call graph information to help software engineers increase their understanding of a program
- Call graphs are crucial for any interprocedural static analysis
- Soundness and precision of a call graph directly affects the soundness and precision of a client analysis that uses it

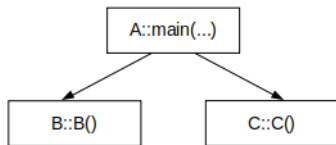
Call Graph

- A static abstraction of all the methods that can be called by a program
- An interprocedural analysis requires an approximation of the call graph

Example Program

```
1 public class A
2 {
3     public static void main(String args
4         [])
5     {
6         if(...)
7             B b = new B();
8         else
9             C c = new C();
10    }
11    ...
12 }
```

Call Graph



Soundness and Precision

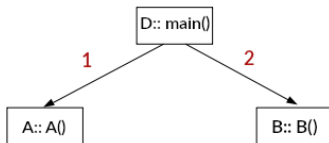
Example Program

```

1 public class A
2 {
3   A foo(A x) {return x;}
4 }
5 public class B extends A
6 {
7   A foo(A x) {return new C();}
8 }
9 public class C extends A
10 {
11   A foo(A x) {return new A();}
12 }
13 public class D
14 {
15   public static void main(String[]
      args)
16   {
17     A x = new B();
18     A y = new A();
19     y.foo(x);
20   }
21 }

```

Unsound Call Graph^a



^aEdges have been numbered for convenience

Soundness and Precision

Example Program

```

1 public class A
2 {
3   A foo(A x) {return x;}
4 }
5 public class B extends A
6 {
7   A foo(A x) {return new C();}
8 }
9 public class C extends A
10 {
11   A foo(A x) {return new A();}
12 }
13 public class D
14 {
15   public static void main(String[]
16     args)
17   {
18     A x = new B();
19     A y = new A();
20     y.foo(x);
21 }

```

Imprecise Call Graph^a



^aEdges have been numbered for convenience

Soundness and Precision

Example Program

```

1 public class A
2 {
3   A foo(A x) {return x;}
4 }
5 public class B extends A
6 {
7   A foo(A x) {return new C();}
8 }
9 public class C extends A
10 {
11   A foo(A x) {return new A();}
12 }
13 public class D
14 {
15   public static void main(String[]
      args)
16   {
17     A x = new B();
18     A y = new A();
19     y.foo(x);
20   }
21 }

```

Sound and Precise Call Graph^a



^aEdges have been numbered for convenience

Call Graph Construction

- Call graphs can be constructed in advance or on-the-fly
- In C, the problem is difficult because of function pointers
- In C++, we additionally have virtual functions
- In Java and web frameworks like Spring, we have reflection and all calls are virtual by default (unlike C++)

Existing Algorithms: CHA

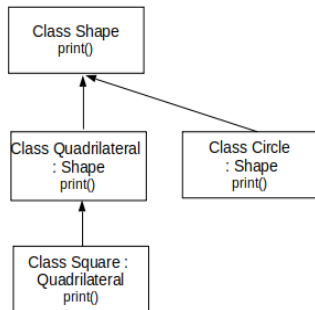
- Simplest flow-insensitive algorithm
- Looks at class hierarchy:
 - In Java, if a reference variable r has a type A , the possible classes of run-time objects are included in the subtree of A
 - Denoted by $\text{cone}(A)$
- Finds out what methods may be called at a virtual call site
- Assumes that entire inheritance hierarchy is available
- Example follows

Example: CHA Call Graph (1)

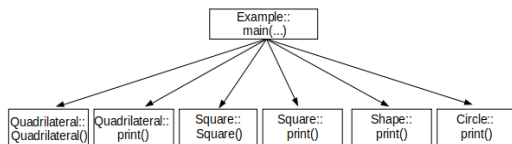
```
1 public class Example
2 {
3     public static void main(String[] args)
4     {
5         Shape b1 = new Quadrilateral();
6         Shape c1 = new Square();
7         Shape b2 = b1;
8         Shape c2 = c1;
9         b2.print(c2);
10    }
11    public static class Shape extends objects
12    {
13        public void print(Shape object) { ....}
14    }
15    public static class Quadrilateral extends Shape
16    {
17        public void print(Shape object) { ....}
18    }
19    public static class Square extends Quadrilateral
20    {
21        public void print(Shape object) { ....}
22    }
23    public static class Circle extends Shape
24    {
25        public void print(Shape object) { ....}
26    }
27 }
```

Example: CHA Call Graph (2)

Inheritance Hierarchy



CHA Call Graph



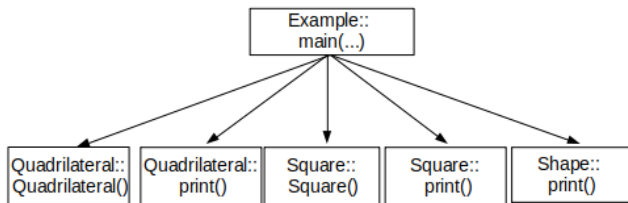
Existing Algorithms: RTA

- Starts with a call graph generated by performing CHA
- Eliminates from the hierarchy classes that are never instantiated
- Iteratively builds a set of instantiated types, method names invoked and concrete methods called (starts from the main function)
- Example follows

Example: RTA Call Graph (1)

```
1 public class Example
2 {
3     public static void main(String[] args)
4     {
5         Shape b1 = new Quadrilateral();
6         Shape c1 = new Square();
7         Shape b2 = b1;
8         Shape c2 = c1;
9         b2.print(c2);
10    }
11    public static class Shape extends objects
12    {
13        public void print(Shape object) { ....}
14    }
15    public static class Quadrilateral extends Shape
16    {
17        public void print(Shape object) { ....}
18    }
19    public static class Square extends Quadrilateral
20    {
21        public void print(Shape object) { ....}
22    }
23    public static class Circle extends Shape
24    {
25        public void print(Shape object) { ....}
26    }
27 }
```

Example: RTA Call Graph (2)



RTA Call Graph

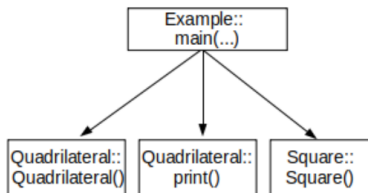
Existing Algorithms: SPARK

- A flexible framework for experimenting with points-to analyses for Java
- Supports both subset-based and equality-based flow-insensitive analyses
- Analysis on jimple input consists of three stages:
 - building the pointer assignment graph (PAG)
 - simplifying the PAG
 - propagating the points-to sets along it to obtain the final solution
- Example follows

Example: SPARK Call Graph (1)

```
1 public class Example
2 {
3     public static void main(String[] args)
4     {
5         Shape b1 = new Quadrilateral();
6         Shape c1 = new Square();
7         Shape b2 = b1;
8         Shape c2 = c1;
9         b2.print(c2);
10    }
11    public static class Shape extends objects
12    {
13        public void print(Shape object) { ....}
14    }
15    public static class Quadrilateral extends Shape
16    {
17        public void print(Shape object) { ....}
18    }
19    public static class Square extends Quadrilateral
20    {
21        public void print(Shape object) { ....}
22    }
23    public static class Circle extends Shape
24    {
25        public void print(Shape object) { ....}
26    }
27 }
```


Example: SPARK Call Graph (2)

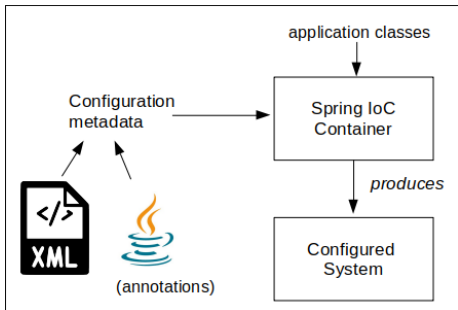


SPARK Call Graph

Spring Framework

- Provides a comprehensive programming and configuration model for modern Java-based enterprise applications
- An application framework for the Java platform
- Commonly used for web applications

- Spring Inversion of Control container
 - The core of the Spring Framework
 - Inversion of Control: custom-written portions of a computer program receive the flow of control from a generic framework
 - Creates the objects, wires them together, configures them and manages their complete life cycle
 - Uses dependency injection to manage the components that make up an application



● Dependency Injection

- Application classes should be as independent as possible of other Java classes to increase the possibility to reuse them
- Dependency injection helps in gluing these classes together and at the same time keeping them independent
- Dependency Injection in Spring can be done through constructors, setters or fields

- Spring beans
 - The objects that form the backbone of an application
 - Managed by the Spring IoC container
 - Objects that are instantiated, assembled and otherwise managed by a Spring IoC container

- Spring annotations
 - Describe a bean wiring directly in a Java file without using an XML file
 - Annotations move bean configuration into the component class
 - Can be used on the relevant class, method or field declaration
 - Examples follow

Some Important Annotations (1)

- *@Configuration*
 - Indicates that the class can be used by the Spring IoC container as a source of bean definitions
- *@Bean*
 - Marks a factory method which instantiates a Spring bean

```
1 <?xml version = "1.0" encoding = "UTF-8"?>
2 <beans xmlns = "http://www.springframework.org/schema/
  beans"
3   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
4   xmlns:context = "http://www.springframework.org/schema/
  context"
5   xsi:schemaLocation = "http://www.springframework.org/
  schema/beans
6   http://www.springframework.org/schema/beans/spring-
  beans-3.0.xsd
7   http://www.springframework.org/schema/context
8   http://www.springframework.org/schema/context/spring-
  context-3.0.xsd">
9   <context:component-scan base-package="package"/>
10  <bean id = "b" class = "package.B"/>
11 </beans>
```

Listing 1: Bean Configuration File

```
1 @Configuration
2 public class A
3 {
4     @Bean
5     public B b()
6     {
7         return new B();
8     }
9     ...
10 }
```

Listing 2: Java File with Annotations

Some Important Annotations (2)

- *@Autowired*
 - Marks a dependency which Spring is going to resolve and inject
 - Can be used with a constructor, setter or field injection
- *@Qualifier*
 - Used along with *@Autowired* to provide the bean id or bean name we want to use in ambiguous situations
 - Can be used with a constructor, setter or field injection

```
1  ....
2  <!-- Definition for student1 bean -->
3  <bean id = "student1" class = "Education.
4     Student">
5     <property name = "name" value = "Zara" />
6     <property name = "age" value = "11"/>
7 </bean>
8 <!-- Definition for student2 bean -->
9 <bean id = "student2" class = "Education.
10    Student">
11    <property name = "name" value = "Neha" />
12    <property name = "age" value = "2"/>
13 </bean>
```

Listing 3: Bean Configuration File

```
1  @Configuration
2  public class Profile
3  {
4     @Autowired
5     @Qualifier("student1")
6     private Student student;
7     ...
8  }
```

Listing 4: Java File with Annotations

Components of a Spring Application



Java classes
of the
application

+



Bean
configuration
file

A Simple Spring Application (1)

```
1 public class Student
2 {
3     private Integer age;
4     private String name;
5     // Getter setter methods
6 }
```

```
1 public class Profile
2 {
3     private Student student;
4     // Getter setter methods and print method for Name, Age of student
5 }
```

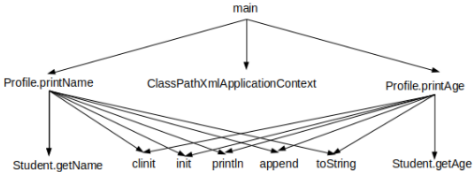

```
1 public class MainApp
2 {
3     public static void main(String[] args)
4     {
5         ApplicationContext context = new ClassPathXmlApplicationContext("Bean.xml");
6         Profile profile = (Profile) context.getBean("profile");
7         profile.printName();
8         profile.printAge();
9     }
10 }
```

A Simple Spring Application (2)

```
1 <!-- Definition for profile bean -->
2 <bean id = "profile" class = "Education.Profile"/>
3
4 <!-- Definition for student1 bean -->
5 <bean id = "student1" class = "Education.Student">
6   <property name = "name" value = "Zara" />
7   <property name = "age" value = "11"/>
8 </bean>
9
10 <!-- Definition for student2 bean -->
11 <bean id = "student2" class = "Education.Student">
12   <property name = "name" value = "Neha" />
13   <property name = "age" value = "2"/>
14 </bean>
```

Call Graph Observations (1)

Remark: These are the call graphs for the example in Listing 5. Only the edges relevant to the input are shown.

<p>CHA/RTA/VTA</p> <p>Total edges in the call graph : 84,522 (for all three algorithms)</p>	 <pre> graph TD main --> Profile.printName main --> ClassPathXmlApplicationContext main --> Profile.printAge Profile.printName --> Student.getName Profile.printName --> clinit Profile.printName --> init Profile.printName --> println Profile.printName --> append Profile.printName --> toString Profile.printAge --> Student.getAge Profile.printAge --> clinit Profile.printAge --> init Profile.printAge --> println Profile.printAge --> append Profile.printAge --> toString </pre>
<p>SPARK</p> <p>Total edges in the call graph : 40,646</p>	 <pre> graph TD main --> ClassPathXmlApplicationContext </pre>

Call Graph Observations (2)

SPARK misses some edges related to the class Profile, Student.
Time to dive into the implementation of Spring!

Debug, Dude!



Debugging the code

```

public static <T> T instantiateClass(Constructor<T> ctor, Object... args) throws BeanInstantiationException {
    Assert.notNull(ctor, message: "Constructor must not be null");
    try {
        ReflectionUtils.makeAccessible(ctor);
        return ctor.newInstance(args);
    } catch (InstantiationException var3) {
        throw new BeanInstantiationException(ctor.getDeclaringClass(), "Is it an abstract class?", var3);
    } catch (IllegalAccessException var4) {
        throw new BeanInstantiationException(ctor.getDeclaringClass(), "Is the constructor accessible?", var4);
    } catch (IllegalArgumentException var5) {
        throw new BeanInstantiationException(ctor.getDeclaringClass(), "Illegal arguments for constructor", var5);
    } catch (InvocationTargetException var6) {
        throw new BeanInstantiationException(ctor.getDeclaringClass(), "Constructor threw exception", var6.getTargetException());
    }
}

public static Method findMethod(Class<?> clazz, String methodName, Class<?>... paramTypes) {

```

Instantiating the class *Profile*

Observations

- Spring converts every call to the function *getBean(...)* into a call to the reflective function *newInstance(...)*
- SPARK is unable to establish the link between the class (defined in the Java file) and the bean (defined in the XML file) due to its inability to handle this reflective function call
- Use of reflection makes SPARK unsound

A Spring Application with Core Annotations

```
1 @Component("engine")
2 public class Engine
3 {
4     @Autowired
5     Engine Engine()
6     {
7         return new Engine();
8     }
9     void print()
10    {
11        System.out.println("In print()");
12    }
13 }
```

```
1 public class App
2 {
3     public static void main(String[] args)
4     {
5         ApplicationContext context = new ClassPathXmlApplicationContext("Bean.xml");
6         Engine e = (Engine) context.getBean(Engine.class);
7         e.print();
8     }
9 }
```

Listing 6: Input Program

Call Graph Observations

Remark: These are the call graphs for the example in Listing 6. Only the edges relevant to the input are shown.

<p>CHA/RTA/VTA</p> <p>Total edges in the call graph : 52,553 (for all three algorithms)</p>	<pre> graph TD main --> getBean(...) main --> ClassPathXmlApplicationContext main --> Engine.print() </pre> <p>The call graph misses the annotated function Engine()</p>
<p>SPARK</p> <p>Total edges in the call graph : 20,537</p>	<pre> graph TD main --> getBean(...) main --> ClassPathXmlApplicationContext </pre> <p>The call graph misses the annotated function Engine(), also missing the explicit call to print()</p>

Debugging the code

The screenshot shows an IDE window with the following tabs: `ostProcessor.class`, `ConfigurationClassPostProcessor.class`, and `AutowiredAnnotationBeanPostProcessor.class`. The status bar indicates "Decompiled .class file, bytecode version: 50.0 (Java 6)" and a "Download..." button. The "Alter..." dropdown is set to `spring-beans-4.1.6.RELEASE.jar`.

The code editor displays the following Java code with a breakpoint (red bug icon) on line 433:

```

421         }
422     } else {
423         this.cachedMethodArguments = null; cachedMethodArguments: Object[0]@
424     }
425
426     this.cached = true; cached: true
427 }
428 }
429 }
430
431 if (arguments != null) {
432     ReflectionUtils.makeAccessible(method);
433     method.invoke(bean, arguments); method: "Vehicle.Engine.Vehicle.Engine.Engine()"
434 }
435
436 } catch (InvocationTargetException var16) {
437     throw var16.getTargetException();
438 } catch (Throwable var17) {
439     throw new BeanCreationException("Could not autowire method: " + method, var17);

```

The breadcrumb at the bottom of the editor reads: `AutowiredAnnotationBeanPostProcessor > AutowiredMethodElement > inject()`.

Reflective Calls in Annotation Handler

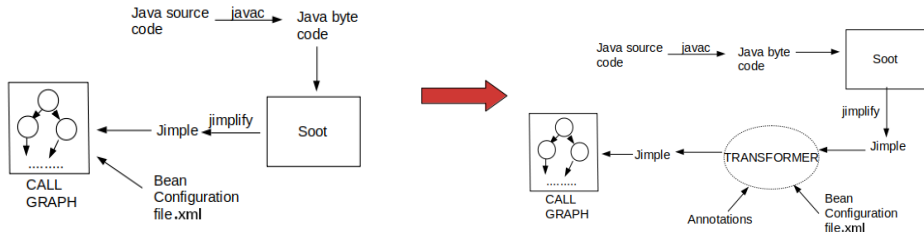
Observations

- Spring annotations call reflective calls at the backend
- These reflective calls are *method.invoke(...)* calls
- All annotated functions absent from the final call graph (for all algorithms)
- Unsoundness observed for all core annotations
- SPARK also misses the call to *print()* due to its inability to detect the *Engine* bean

Compiling Reflective Calls: The First Attempt

Idea - Create a transformer:

- Input: Jimple code + XML file
- Output: Jimple code (without reflective functions)

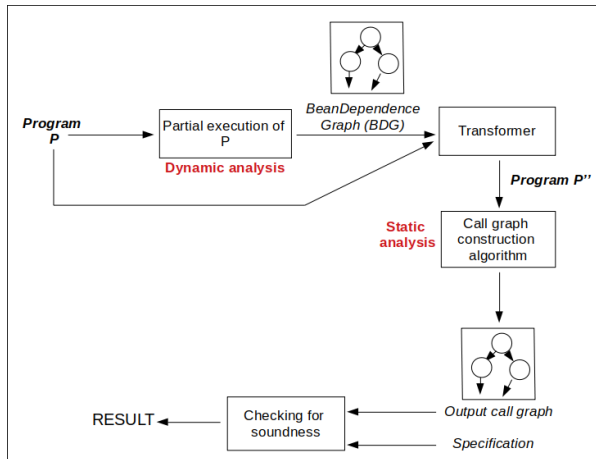


Research Questions

- 1 Which other basic functions in Spring use reflection?
- 2 Can we replace these basic functions by transforming them into statically equivalent code?
- 3 For the reflective functions that cannot be transformed into equivalent code,
 - How can we handle the rest of the functions?
 - If those functions are not frequently used, can we avoid handling them?
- 4 Will the transformer output unambiguous results? If not, what can be done to ensure unambiguity?
- 5 Will such a transformer be too restricted to Spring framework? How can it be extended to other web frameworks?

Our Solution (1)

- 1 Extract Bean Data from Spring Container
- 2 Store it in Bean Dependence Graph (BDG)
- 3 Transform reflective calls into non-reflective calls



Our Solution (2)

```
1 public class App
2 {
3     public static void main(String[] args)
4     {
5         ApplicationContext context = new ClassPathXmlApplicationContext("file:Bean.xml");
6         Car c = (Car) context.getBean(Car.class);
7         c.print();
8     }
9 }
```

Listing 7: Original Program P

```
1 public class App
2 {
3     public static void main(String[] args)
4     {
5         Engine e = new Engine();
6         Car c = new Car();
7         e = c.engine();
8         c.setEngine(e);
9         c.print();
10    }
11 }
```

Listing 8: Updated Program P"

Bean Dependence Graph

- A directed acyclic graph $G = (V, E)$ such that
 - $\forall v \in V, v = (b, \text{listOfAbstractObjects})$
i.e. every vertex v is a pair of a bean b and a list of abstract objects (objects of the bean class and its subclasses)
 - $\forall u, v \in V$ where $u = (u_b, u_listOfObjects)$ and $v = (v_b, v_listOfObjects)$,
 $(u, v) \in E$ iff bean v_b is autowired in bean u_b
i.e. every bean b in BDG points to a list of beans which are autowired as fields of class b
- In case of some containers, Spring constructs a dependency graph which can be updated to construct a BDG
- Otherwise, we need to borrow the bean information stored in the container and construct a BDG
- Example follows

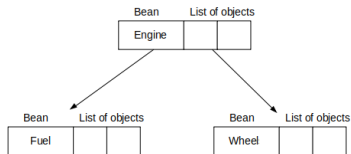
Bean Dependence Graph: An Example

```

1 @Component
2 public class Engine
3 {
4     @Autowired
5     Wheel w;
6     @Autowired
7     Fuel f;
8     @Autowired
9     public void setWheel(Wheel w1)
10    {
11        this.w = w1;
12    }
13    @Autowired
14    public void setFuel(Fuel f1)
15    {
16        this.f = f1;
17    }
18    public void func() {...}
19 }
20 @Component
21 public class Wheel {...}
22 @Component
23 public class Fuel {...}

```

Listing 9: Input Program



Example

Step 1: Original Program

```

1 @Component
2 public class Engine
3 {
4     @Autowired
5     Wheel w;
6     @Autowired
7     Fuel f;
8     @Autowired
9     public void setWheel(Wheel w1)
10    {
11        this.w = w1;
12    }
13    @Autowired
14    public void setFuel(Fuel f1)
15    {
16        this.f = f1;
17    }
18    public void func() {...}
19 }
20 @Component
21 public class Wheel {...}
22 @Component
23 public class Fuel {...}

```

```

1 public class App
2 {
3     public static void main(String[]
4         args)
5     {
6         ApplicationContext context = new
7             ClassPathXmlApplicationContext("
8                 file:Bean.xml");
9         Engine e = (Engine) context.
10             getBean("engine");
11         e.func();
12     }
13 }

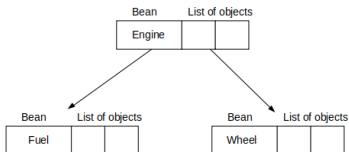
```

Step 2: For every `getBean(...)` call C, construct a BDG rooted at the bean returned by C (here, *Engine*)

```

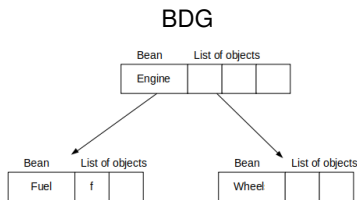
1 @Component
2 public class Engine
3 {
4     @Autowired
5     Wheel w;
6     @Autowired
7     Fuel f;
8     @Autowired
9     public void setWheel(Wheel w1)
10    {
11        this.w = w1;
12    }
13    @Autowired
14    public void setFuel(Fuel f1)
15    {
16        this.f = f1;
17    }
18    public void func() {...}
19 }
20 @Component
21 public class Wheel {...}
22 @Component
23 public class Fuel {...}

```



Step 3:

- 3.1. Traverse the BDG in postorder
- 3.2. Construct objects for every node (except root)
- 3.3. Update the BDG with objects
- 3.4. Update the output program with the constructors and annotated methods

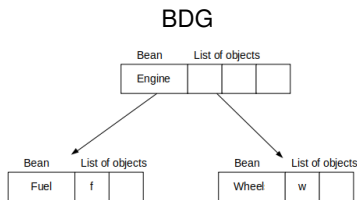


```

1 public class App
2 {
3     public static void main(String[] args)
4     {
5         Fuel f = new Fuel();
6     }
7 }
  
```

Step 3:

- 3.1. Traverse the BDG in postorder
- 3.2. Construct objects for every node (except root)
- 3.3. Update the BDG with objects
- 3.4. Update the output program with the constructors and annotated methods

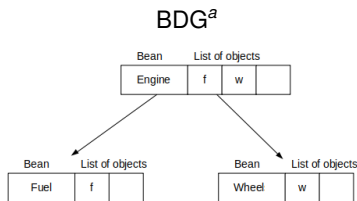


```

1 public class App
2 {
3     public static void main(String[] args)
4     {
5         Fuel f = new Fuel();
6         Wheel w = new Wheel();
7     }
8 }
  
```

Step 3:

- 3.1. Traverse the BDG in postorder
- 3.2. Construct objects for every node (except root)
- 3.3. Update the BDG with objects
- 3.4. Update the output program with the constructors and annotated methods



```

1 public class App
2 {
3     public static void main(String[] args)
4     {
5         Fuel f = new Fuel();
6         Wheel w = new Wheel();
7     }
8 }
  
```

^aThe parent node *Engine* gets the objects of the children

Step 4:

4.1. Use the BDG to update the output program

4.2. Explicitly call annotated methods and other methods on the root

```
1 public class App
2 {
3     public static void main(String[] args)
4     {
5         ApplicationContext context = new ClassPathXmlApplicationContext("file:Bean.xml");
6         Engine e = (Engine) context.getBean("engine");
7         e.func();
8     }
9 }
```

Listing 10: Input Program

```
1 public class App
2 {
3     public static void main(String[] args)
4     {
5         Fuel f = new Fuel();
6         Wheel w = new Wheel();
7         Engine e = new Engine();
8         e.setWheel(w);
9         e.setFuel(f);
10        e.func();
11    }
12 }
```

Listing 11: Output Program

Limitations

Our prototype proposal is specific to the following:

- Spring API method *getBean(...)*
- Spring Containers *ClassPathXmlApplicationContext* and *FileSystemXmlApplicationContext*
- Some core annotations (*@Bean*, *@Autowired*, *@Qualifier*, *@Component*, *@Configuration*, *@Scope*)

At the moment, it does not handle the following:

- Other annotations and reflective calls
- MVC framework in Spring
- Applications with a cyclic bean dependency

Conclusion

- While a lot of research effort has gone into development of sound call graph construction algorithms, the area of call graph construction for web frameworks remains almost unexplored
- Unsoundness of call graph construction algorithms for Spring framework is only due to the presence of reflective calls
- We present a hybrid analysis algorithm which generates a simple, non-reflective version of the input Spring program written in Java
- This algorithm would work on programs written in web frameworks designed on top of the Spring framework and the call graph constructed for these programs would be sound and precise, as required

Future Work

- Apart from Spring beans and core annotations, reflective calls are used in the Spring API for the following:
 - Creating an Async Web Request, which is the first step in running a Spring MVC application
 - Calling annotated methods in MVC Annotation Handlers (*@Controller*, *@RequestMapping* etc.)
 - Specifying the JDBC Driver implementation class
 - Creating a new annotation type filter for the given annotation type
- Our hybrid analysis algorithm can be extended to handle these cases and construct sound and precise call graphs for Spring MVC applications

References (1)

- (1) Call Graph Construction Algorithms.
<https://ben-holland.com/call-graph-construction-algorithms-explained/>.
- (2) Dependency Injection.
<https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring>.
- (3) Spring Annotations.
<https://www.baeldung.com/spring-core-annotations>.
- (4) Spring Framework.
<https://spring.io/projects/spring-framework>.
- (5) Spring Tutorial.
<https://www.tutorialspoint.com/spring>.
- (6) Spring Web MVC Framework.
<https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/mvc.html>.

References (2)

- (7) Wikipedia contributors.
https://en.wikipedia.org/wiki/Call_graph.
- (8) Wikipedia contributors.
https://en.wikipedia.org/wiki/Inversion_of_control.
- (9) Ondrej Lhotak and Laurie Hendren. Scaling Java Points-to Analysis Using SPARK. In Proceedings of the 12th International Conference on Compiler Construction. CC 03. Warsaw, Poland.
<http://dl.acm.org/citation.cfm?id=1765931.1765948>.

Questions?



Extra slides

Algorithm (1)

Algorithm 1: Traverse the BDG in postorder and call annotated functions

Input: Bean Dependence Graph BDG^a

Output: Root of BDG

```
1 begin
2   for (every node n) do
3     Instantiate object obj for all children(n)
4     Call all methods annotated by @Autowired on obj
5     if (n is not the root) then
6       Instantiate object obj for n
7       Call all methods annotated by @Autowired on obj
8     end
9   end
10  return root(BDG)
11 end
```

^aAssumption: BDG is acyclic

Algorithm (2)

Algorithm 2: Transform Code^a Using Bean Information

```
1 begin
2   for every getBean(...) call C in input program P do
3     b = bean returned by C
4     root = populateBeans(BDG)
5     Update the output program:
6       Instantiate object obj for root
7       Call all methods annotated by @Autowired on obj
8       Call all direct methods of class(root) on obj
9   end
10 end
```

^aWe only consider the containers *ClassPathXmlApplicationContext* and *FileSystemXmlApplicationContext* in this prototype

Handling Different Bean Scopes (1)

When defining a bean, there is an option to declare a scope for that bean. The Spring Framework supports two main scopes:

- 1 **singleton**: Scopes the bean definition to a single instance per Spring IoC container (default)

```

1 public class App
2 {
3     public static void main(String[] args)
4     {
5         ApplicationContext context = new
6             ClassPathXmlApplicationContext(...);
7         Car c = (Car) context.getBean("car");
8         c.print();
9         Car cc = (Car) context.getBean("car");
10        cc.print();
11    }

```

Listing 12: Original Program

```

1 public class App
2 {
3     public static void main(String[]
4         args)
5     {
6         Engine e = new Engine();
7         Car c = new Car();
8         e = c.engine();
9         c.setEngine(e);
10        c.print();
11        Car cc = c;
12        cc.print();
13    }

```

Listing 13: Updated Program

Handling Different Bean Scopes (2)

- ② *prototype*: Scopes a single bean definition to have any number of object instances. The Spring IoC container creates a new bean instance of the object every time a request for that specific bean is made

```

1 public class App
2 {
3     public static void main(String[] args)
4     {
5         ApplicationContext context = new
6             ClassPathXmlApplicationContext(...);
7         Car c = (Car) context.getBean("car");
8         c.print();
9         Car cc = (Car) context.getBean("car");
10        cc.print();
11    }

```

Listing 14: Original Program

```

1 public class App
2 {
3     public static void main(String[]
4         args)
5     {
6         Engine e = new Engine();
7         Car c = new Car();
8         e = c.engine();
9         c.setEngine(e);
10        c.print();
11        Engine e2 = new Engine();
12        Car cc = new Car();
13        e2 = cc.engine();
14        cc.setEngine(e2);
15        cc.print();
16    }

```

Listing 15: Updated Program

Handling Different Spring Containers

What happens when we initialize a Spring IoC Container?

