Lecture 6a: Model and Code Analysis

Gregory Gay
TDA594 - November 19, 2020
Where We Stand

- Feature Models can be expressed using propositional logic formulae ($\varphi$).
  - Based on model and cross-tree constraints.
- Valid feature selections result in ($\varphi = \text{true}$).
- SAT Solvers can identify valid configurations.
  - If none can be found, the model is inconsistent.
  - Enables many different model analyses.
Today’s Goals

• Feature-to-Code Mappings
• Domain Implementation (Analysis of Code)
Feature-to-Code Mappings
Feature-To-Code Mappings

- Feature models describe the problem space.
- Models are implemented in source code.
- Similar analyses can examine mapping of feature models to code.
  - Which code assets are never used?
  - Which code assets are always used?
  - Which features have no influence on product portfolio?
Dead Code

- Features that can never be incorporated.
- Feature B, in the code, required Feature A to also be selected.
- Model states that A and B are mutually exclusive.

```
1 line 1
2 #ifdef A
3 line 3
4 #ifdef B
5 line 5
6 #endif
7 #else
8 line 8
9 #endif
```
Presence Conditions

• Describes the set of products containing a code fragment.

• $pc(c) = \text{(conditions for } c\text{ to be included in a product)}$
  
  • $pc(\text{line 3}) = A$
  • $pc(\text{line 5}) = A \land B$
  • $pc(\text{line 8}) = \neg A$

  • $pc(\text{lines 3-5}) = A \land B$
  • $pc(\text{lines 3-8}) = A \land B \land \neg A$
    ○ (cannot be included in any product)
Dead Code

- Fragment is dead if never included in any product.
  - $\varphi$ represents all valid products.
  - Fragment C is dead iff $(\varphi \land \text{pc}(C))$ is not satisfiable.

$\varphi = \text{Program} \land (A \lor B) \land \neg(A \land B)$

$(\varphi \land \text{pc(line 5)})$ is not satisfiable:
Program $\land (A \lor B) \land \neg(A \land B) \land (A \land B)$
Mandatory Code

• Fragment is mandatory if always included in a product.
  • $\phi$ represents all valid products.
  • Fragment C is mandatory iff $(\phi \land \neg pc(C))$ is not satisfiable.

If code implemented correctly, the fragment for EdgeType will be mandatory.
Domain Implementation
Analysis of Product Line Code

• Focus on analyzing variability in program structures

• Variability-aware Analyses
  • Traditional analyses (i.e., type checking) extended from one product to entire line.
  • Goal of analyzing whole line in one pass instead of all individual products.
Example: Type Checking

• Verifying and enforcing constraints of data types.
  • Is String being used as Integer?
  • If we call a method, does it return the right type of data?

• Can be checked during compilation or at runtime.

• Same analyses can be applied to other properties.

Part1 = 10
Part2 = “Wobuffet”
Sum = Part1 + Part2

String getName() {
    return “Wobuffet”;
}
Part1 = 10
Sum = Part1 + getName()
Terminology

• Check **properties** about program or feature model.
  • Type Checking: Does the program have type errors?
  • We assume a property must hold over all **products**.

• **Complete** variability-aware analyses give same results as brute-force analysis.

• **Sound** analyses ensure all violations in domain artifacts hold in concrete products.
Sampling Strategies

• Instead of brute-force, try a subset of products.

• Selection criteria:
  • **Feature Coverage**: All features covered at least once.
  • **Feature-Code Coverage**: All code fragments included at least once.
  • **Pairwise Feature Coverage**: All pairs of features covered at least once.
    • **N-wise Coverage**: All N-way (3-way, 4-way,...) combinations.
Sampling Strategies

• Strategies:
  • **Popular Features**: Focus on what customers use
  • **Domain-Specific**: Base coverage on factors important to product domain.

• Balance between # of analyses and error detection.
  • Sampling is *sound*, but *not complete*.
    • Detected errors hold in products, but not all products tested.
Family-Based Type Checking

- Compiler uses `#ifdef` annotation to decide what code to include in binary.
- Graph product line, Node class.
  - Features: NAME, NONAME, COLOR.
  - Selecting neither or both NAME/NONAME leads to error.

```java
class Node {
    int id = 0;

    // ifdef NAME
    private String name;
    String getName() { return name; }
    // endif

    // ifdef NONAME
    String getName() { return String.valueOf(id); }
    // endif

    // ifdef COLOR
    Color color = new Color();
    // endif

    void print() {
        // ifdef defined(COLOR) && defined(NAME)
        Color.setColor(Color(color);
        // endif
        System.out.print(getName());
    }
    // endif

    // ifdef COLOR
    class Color {
        static void setColor(Color c){/*...*/}
    }
    // endif
}
```
Presence Conditions on Structures

- Can identify presence conditions for classes, methods, fields, variables.
  - \(pc(\text{name}() \ [\text{line } 6]) = \text{NAME}\)
  - \(pc(\text{name}() \ [\text{line } 9]) = \text{NONAME}\)
  - \(pc(\text{Name}.\text{setDisplayColor}(\text{color}) \ [\text{line } 18]) = \text{COLOR} \land \text{NAME}\)
  - \(pc(\text{System.out.print}(\text{name}()) \ [\text{line } 20]) = \text{TRUE} \Rightarrow (\text{NAME} \lor \text{NONAME})\)
  - Calls name(), requires at least one to exist.
Presence Conditions on Structures
Reachability

- Examine lines reachable from each line to identify presence conditions.
- If NAME \land NONAME, error on line 9.
- If \neg NAME \land \neg NONAME, error on line 20.
Reachability Conditions

• When a call is made from source to target, a valid target must exist.
  • $\varphi \Rightarrow (pc(s) \Rightarrow \lor_{t \in T} pc(t))$

• If negation of this constraint can be satisfied, there are feature selections that will not compile.
  • SAT solver can identify selections where there are no valid targets for a call from a source.
Reachability

<table>
<thead>
<tr>
<th>Construct (type reference)</th>
<th>Source</th>
<th>Target</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>String (type reference)</td>
<td>5</td>
<td>JSL</td>
<td>$\phi \Rightarrow (\text{Name} \Rightarrow T)$</td>
</tr>
<tr>
<td>String (type reference)</td>
<td>6</td>
<td>JSL</td>
<td>$\phi \Rightarrow (\text{Name} \Rightarrow T)$</td>
</tr>
<tr>
<td>name (field access)</td>
<td>6</td>
<td>5</td>
<td>$\phi \Rightarrow (\text{Name} \Rightarrow \text{Name})$</td>
</tr>
<tr>
<td>String (type reference)</td>
<td>9</td>
<td>JSL</td>
<td>$\phi \Rightarrow (\text{NoName} \Rightarrow T)$</td>
</tr>
<tr>
<td>String.valueOf (method invocation)</td>
<td>9</td>
<td>JSL</td>
<td>$\phi \Rightarrow (\text{NoName} \Rightarrow T)$</td>
</tr>
<tr>
<td>id (field access)</td>
<td>9</td>
<td>2</td>
<td>$\phi \Rightarrow (\text{NoName} \Rightarrow T)$</td>
</tr>
<tr>
<td>Color (type reference)</td>
<td>13</td>
<td>24</td>
<td>$\phi \Rightarrow (\text{Color} \Rightarrow \text{Color})$</td>
</tr>
<tr>
<td>Color (instantiation)</td>
<td>13</td>
<td>24</td>
<td>$\phi \Rightarrow (\text{Color} \Rightarrow \text{Color})$</td>
</tr>
<tr>
<td>Color.setDisplayColor (method inv.)</td>
<td>18</td>
<td>25</td>
<td>$\phi \Rightarrow ((\text{Color} \land \text{Name}) \Rightarrow \text{Color})$</td>
</tr>
<tr>
<td>color (field access)</td>
<td>18</td>
<td>13</td>
<td>$\phi \Rightarrow ((\text{Color} \land \text{Name}) \Rightarrow \text{Color})$</td>
</tr>
<tr>
<td>System.out (field access)</td>
<td>20</td>
<td>JSL</td>
<td>$\phi \Rightarrow (T \Rightarrow T)$</td>
</tr>
<tr>
<td>PrintStream.print (method invocation)</td>
<td>20</td>
<td>JSL</td>
<td>$\phi \Rightarrow (T \Rightarrow T)$</td>
</tr>
<tr>
<td>getName (method invocation)</td>
<td>20</td>
<td>6, 9</td>
<td>$\phi \Rightarrow (T \Rightarrow (\text{Name} \lor \text{NoName}))$</td>
</tr>
<tr>
<td>Color (type reference)</td>
<td>25</td>
<td>24</td>
<td>$\phi \Rightarrow (\text{Color} \Rightarrow \text{Color})$</td>
</tr>
<tr>
<td>getName (method redeclaration)</td>
<td>9</td>
<td>6</td>
<td>$\phi \Rightarrow \neg (\text{Name} \land \text{NoName})$</td>
</tr>
</tbody>
</table>

JSL = Java Standard Library

```java
class Node {
    int id = 0;
    private String name;
    String getName() { return name; }
}
```

```java
void print() {
    //if defined(COLOR) && defined(NAME)
    Color.setColor(Color.valueOf(id));
}
```

```java
static void setColor(Color c){/*...*/}
```
Beyond Type Checking

• Same approach can be used for checking many properties.
• Lift from individual product to whole line.
  • Analyze shared code once.
  • Reason about configurations using logic and SAT solvers.
We Have Learned

• Feature Models can be expressed using propositional logic formulae ($\phi$).
  • Based on model and cross-tree constraints.

• Valid feature selections result in ($\phi = \text{true}$).

• SAT Solvers can identify valid configurations.
  • If none can be found, the model is inconsistent.
  • Enables many different model analyses.
We Have Learned

• Feature-Model Analysis
  • Check properties of model are true.
  • Dead and mandatory features
  • Effects of partial selections
  • Comparisons between two models

• Mapping of models and code
  • Dead and mandatory code

• Implementation analysis
  • Do called assets exist and return the correct data type?
Let’s take a break!
Variability

• The ability to derive different products from a common set of assets.

• Implementation: How do we build a custom product from a feature selection?
  • Binding Time
  • Technology (Language vs Tool-Based Implementation)
  • Representation (Annotation vs Composition)
Today’s Goals

• Basic implementation concepts
• Tool-based Implementation
  • Focus on preprocessor-based implementation
• Introduce language-based implementation
  • Parameters
  • Next class: Implementing variability via design patterns.
**Binding Time**

- **Compile-time Binding**
  - Decisions made when we compile.
  - `#IFDEF` preprocessor in C/C++.

- **Load-time Binding**
  - Decisions made when program starts.
  - Configuration file or command-line flags.

- **Run-time Binding**
  - Decisions made while program runs.
  - Method or API call.
Binding Time

• Compile-time binding improves performance.
  • … but executable cannot be configured further.
• Load-time binding configured at execution.
• Run-time binding can be configured any time.
  • … but results in reduced performance, security hazards, and program complexity.
Technology

• Language-based Implementation
  • Use programming language mechanisms to implement features and derive product.
  • Pass parameters at run-time.

• Tool-based Implementation
  • Use external tools to derive a product.
  • Use preprocessor to compile only the requested features.
Technology

• Language-Based Implementation
  • Feature implementation **and** management in code.
  • Easy to understand.
  • Feature management/boundaries easily vanishes.

• Tool-Based Implementation
  • Separation between implementation and management.
  • Can simplify code.
  • Must reason about multiple artifacts.
Annotation-Based Representation

• All code in common code base.
• Code related to a feature marked in some form.
  • Preprocessor annotations, if-statement that checks input.
• Code belonging to deselected features ignored (run-time) or removed (compile-time).
• Adds complexity, reduces modularity/readability.
Composition-based Representation

• Code belonging to feature in dedicated location.
  • Class, file, package, service
• Selected units combined to form final product.
• Requires clear mapping between features and units
• Can combine annotation and composition.
  • Annotation-based approaches remove code.
  • Composition-based approaches add code.
Some Examples

• Preprocessors
  • Compile-time, tool-based, annotation-based

• Parameters
  • Load or run-time, language-based, annotation-based

• Design Patterns
  • Load or run-time, language-based, composition-based
Preprocessor-Based Implementation
Preprocessors

- Optimize code before compilation.
  - Often used by compilers to produce faster executable.
  - Can selectively include or exclude code.
- Most famous - cpp
  - “The C Preprocessor”
- Exist for many languages.

```java
class Node {
    int id = 0;

    //ifdef NAME
    private String name;
    String getName() { return name; }
    //endif

    //ifdef NONAME
    String getName() { return String.valueOf(id); }
    //endif

    //ifdef COLOR
    Color color = new Color();
    //endif

    void print() {
        //if defined(COLOR) && defined(NAME)
        Color.setDisplayColor(color);
        //endif
        System.out.print(getName());
    }
}
```
Implementation with cpp

• `#include` enables import from another file.
  • `#include <string.h>`

• `#define` used to substitute value for reference.
  • Reserve one per feature.
  • `#define FEATURE_NAME_TRUE`  
    • (if the feature is selected, don’t use `#define` if not selected)

• `#ifdef/#endif` used to conditionally include code.
  • `#ifdef FEATURE_NAME`
Implementation with cpp

```
class Node {
    int id = 0;
    private String name;
    String getName() { return name; }
    //endif
    //ifndef NONAME
    String getName() { return String.valueOf(id); }
    //endif
    //ifndef COLOR
    Color color = new Color();
    //endif
    void print() {
        //if defined(COLOR) && defined(NAME)
        Color.setColorDisplayColor(color);
        //endif
        System.out.print(getName());
    }
    //ifndef COLOR
    class Color {
        static void setColorDisplayColor(Color c){/*...*/}
    }
    //endif
```

- **#ifdef**
- **#if defined(MACRO)**
  - Check if a macro is defined. If true, code is included.
  - Define macro for included features.
- **#if (...)** can check a user-defined condition.
Implementation with cpp

```cpp
static int __rep_queue_filedone(dbenv, rep, rfp)
{
    DB_ENV *dbenv;
    REP *rep;
    __rep_fileinfo_args *rfp; {
        #ifndef HAVE_QUEUE
        COMPUIGHT(rep, NULL);
        COMPUIGHT(rfp, NULL);
        return __db_no_queue_am(dbenv);
        #else
        db_pgno_t first, last;
        u_int32_t flags;
        int empty, ret, t_ret;
        #ifdef DIAGNOSTIC
        DB_MSGBUF mb;
        #endif
        // over 100 lines of additional code
        #endif
    }
}
```

- `#ifndef`  
  - “if not defined”
- `#else`
- Note nesting of directives.
  - Line 17 ends line 5 directive.
Let’s take a break!
Implementation with Antenna (Java)

• Similar to cpp
  • Annotations written as comments.
  • Comments out code that is not selected and uncomment code that is selected.

• Available from http://antenna.sourceforge.net/
  • Part of FeatureIDE or can used from command line.
Implementation with Antenna (Java)

• Annotate code using comments:
  • //=if FEATURE_NAME
    • If FEATURE_NAME is chosen, include this code.
  • //=elif OTHER_FEATURE
    • else if OTHER_FEATURE chosen, include this code.
  • //=else
  • //=endif

• Instead of removing lines, Antenna comments out lines, inserting //=@
Examples

(Hello, Beautiful, World)

```java
public class Main {
    public static void main(String[] args) {
        //if Hello
        System.out.print("Hello");
        //endif
        //if Beautiful
        System.out.print(" beautiful");
        //endif
        //if Wonderful
        //@ System.out.print(" wonderful");
        //endif
        //if World
        System.out.print(" world!");
        //endif
    }
}
```

(Hello, Wonderful, World)

```java
public class Main {
    public static void main(String[] args) {
        //if Hello
        System.out.print("Hello");
        //endif
        //if Beautiful
        //if Wonderful
        //@ System.out.print(" beautiful");
        //endif
        //if Wonderful
        //if World
        System.out.print(" wonderful");
        //endif
        //if World
        System.out.print(" world!");
        //endif
    }
}
```
Proper Use of Preprocessors

• Should wrap around an entire function, declaration, or expression.

```
1 #if defined(__MORPHOS__) &&
2 \defined(__libnx__)
3 extern unsigned long *__stdfiledes;
4
5 static unsigned long
6 fdtofh(int filedescriptor) {
7     return __stdfiledes[filedescriptor];
8 }
9 #endif
```

• Bad annotations wrap partial expressions.

```
1 #ifdef DYNAMIC_TCL
2 void tcl_end() {
3     #ifdef DYNAMIC_TCL
4         FreeLibrary(hTclLib);
5     } hTclLib = NULL;
6 } #endif
7 #endif
8 }
```

```
1 int n = NUM2INT(num);
2 #ifdef FEAT_WINDOWS
3     w = curwin;
4 #else
5     for (w = firstwin; w != NULL;
6         w = w->w_next, --n)
7 #endif
8     if (n == 0)
9         return window_new(w);
10 }
11 #endif
12 #ifdef DYNAMIC_RUBY
13     if (!ruby_initialized) {
14 #ifdef DYNAMIC_RUBY
15         ruby_init();
16     #endif
17     nowrapifier
18     }
19 #endif
20 }
21 int put_eol(fd)
22     FILE *fd;
23 }
24 if (putc(\n', fd) < 0)) ||
25 #endif
26 (putc(\r', fd) < 0))
27 return FAIL;
28 return OK;
29 }
```
Benefits of Preprocessors

• Easy to learn (annotate and remove code).
• Can be applied to code and other artifacts.
• Allow changes at any level of granularity.
• Easy to map features and code.
• Can be added to a non-product line to transform it into one over time.
Drawbacks of Preprocessors

• Feature code scattered across codebase and mixed with other features.
• Encourage developers to patch and add to code instead of refactoring.
• Can make it hard to understand control flow in code
• Can introduce errors, especially when used on partial statements.
Parameter-Based Implementation
Language-Based Variability

- Programming languages offer means to implement variability in different ways.
  - if-statement offers a choice between two options.
- Common approaches:
  - Parameters
  - Design Patterns
  - Frameworks
  - Components and Services
Parameter-based Implementation

- Use conditional statements to alter control flow based on features selected.
- Boolean variable based on feature, set globally or passed directly to methods:
  - From command line or config file (load-time binding)
  - From GUI or API (run-time binding)
  - Hard-coded in program (compile-time binding)
• Choices read from command line and stored in Conf.
• Other classes check variables and invoke code appropriately.
Discussion

• Variation is evaluated at run-time.
• All functionality is included, even if never used.
  • More memory required.
  • If-statements add computational overhead.
• Security risks introduced, i.e., buffer overflow attacks.

```java
Edge add(Node n, Node m, Weight w) {
    if (!Conf.WEIGHTED)
        throw new RuntimeException();
    Edge e = new Edge(n, m);
    e.weight = w;
    nodes.add(n);
    nodes.add(m);
    edges.add(e);
    return e;
}
```
Discussion

• Can alter feature selection at run-time.
  • However, code may depend on initialization steps.
  • May be easier to restart.
• Can pass to methods instead of setting globally.
  • Allows different configurations throughout program.

```java
Edge add(Node n, Node m, Weight w) {
    if (!Conf.WEIGHTED)
        throw new RuntimeException();
    Edge e = new Edge(n, m);
    e.weight = w;
    nodes.add(n);
    nodes.add(m);
    edges.add(e);
    return e;
}
```
Discussion

- Conditional statements are a form of annotation.
  - Mark boundaries between features.
- Global variables reduce independence of modules.
  - However, passing many arguments reduces understandability/requires repetition.
  - Pass a “configuration object” containing settings.
- Feature code mixed and scattered across project.
  - Hard to understand and change.
Benefits and Drawbacks

• Benefits
  • Easy to understand and use.
  • Flexible
  • Allows different configurations in same program.

• Drawbacks
  • All code in executable.
  • Feature code and configuration knowledge scattered across program.
  • Difficult to link feature model and implementation.
We Have Learned

• *How do we build a custom product from a feature selection?*
  • Binding Time
    • Compile, load, run-time
  • Technology
    • Language vs Tool-Based Implementation
  • Representation
    • Annotation vs Composition
We Have Learned

• Preprocessors
  • Mark code to include in compiled executable.
  • Omit code that we do not select entirely.
  • Compile-Time, Tool-Based, Annotation-Based

• Parameters
  • Set Boolean variables via command-line, config file, GUI, API, etc. globally or pass to methods.
  • Use if-statements to execute correct code.
  • Load or Run-Time, Language-Based, Annotation-Based
Next Time

• Variability implementation using design patterns.
  • Load or run-time binding, language-based, composition-based.

• Assignment 2 - any questions?
  • Due November 29
  • Feature modelling and analysis for mobile robots