Controlled Evolution of Adaptive Programs

By:
Ahmed Abdelmeged
Therapon Skotiniotis
Karl Lieberherr
What makes a program both easy and safe to evolve?
Minimize the coupling between the program's behavior* and structure. In other words, Schema Obliviousness.

* One or more collaborating methods.
Example (1)

- Structure sensitive behavior:

```plaintext
double volume(Cylinder c)
{ return PI * c.dimensions.radius^2
  * c.dimensions.height; }
```

- Makes five structural assumptions: Cylinder, dimensions, radius, height, PI.
- Three are relevant to the method's "function": PI, radius, height.
Example (2)

- Structure-shy behavior:

```java
class VolumeCalculator {
    double volume = PI;
    double for_radius(double host) {
        volume *= host ^ 2;
    }
    double for_height(double host) {
        volume *= host;
    }
}
```

- Minimal structural assumptions.
- Requires **specialization/binding** to specific execution contexts.
- Implicit structural assumptions.
  - There is at most one radius, height.
Evolving Structure-Shy Programs Is Easy

- Structure-shy behavior is generic:
  - The structure-shy volume behavior can execute, without changes, against any structure containing at most one radius and at most one height.
- Structure-shy behavior seamlessly adapts to structural evolution.
- Structure-shy behavior is reusable in more contexts.
  - Less reuse via copy-and-paste.
Evolving Structure-Shy Programs Can Be Dangerous

- **Problem 1:** Evolving the structure might violate implicit assumptions made by the behavior.
  - “Express” and check these implicit assumptions.
  - Infer some implicit assumptions using static analysis.
Evolving Structure-Shy Programs Can Be Dangerous

- **Problem 1**: Evolving the structure might violate implicit assumptions made by the behavior.
  - “Express” and check these implicit assumptions.
  - Infer some implicit assumptions using static analysis.

- **Problem 2**: A small change to the program might have a “drastic” effect on its meaning.
  - Increase the “syntactic distance” between legal programs by adopting a stricter notion of legality.
Contribution

- “Solve” these two problems in the context of one concrete paradigm for writing structure-shy programs: *Adaptive Programming*. 
Adaptive Programs: Overview

- Adaptive programs are organized as advised depth first traversals over semi-structured data objects.
Adaptive Programs: Example

- An adaptive program comprises:
  - An input object: to be traversed - **The context**.
    ```java
    Cylinder input = new Cylinder(new Dimensions(3.0, 1.0));
    ```
  - A set of advices: fired along the traversal – **The Behavior**.
    ```java
    class VolumeCalculator extends Visitor{
    double volume = PI;
    void before_radius (double host) { volume *= host ^ 2; }
    void before_height (double host) { volume *= host; }
    }
    ```
  - A traversal strategy: picks a set of paths to be traversed - **Specializes methods to a context**.
    ```java
    Visitor v = new VolumeCalculator();
    ExecuteAdaptiveProgram(input, v, "from * to double");
    ```
Adaptive Programs: Execution

- Construct an automaton from the input schema and the strategy.
  - Traversal graph.
- Traverse the input object guided by the automaton.
  - Before traversing a child, make sure that it won't drive the automaton to a state with no tokens.
- Fire advices as the traversal proceeds.
Constructing Traversal Graphs

Strategy

Input Schema

Traversal Graph
Constructing Traversal Graphs

Strategy

Input Schema

Traversal Graph
Constructing Traversal Graphs

Strategy

Input Schema

Traversals Graph
Constructing Traversal Graphs

Strategy

Input Schema

Traversal Graph
Constructing Traversal Graphs

Strategy

Input Schema

Traversal Graph
Constructing Traversal Graphs

Strategy

Input Schema

Traversal Graph
Smoothing out Traversal Graphs

- Represents the set of all possible advice execution traces.

Traversals:
- Smoothened out Traversal Graph:
  - Only A, D are advised
- Traversal Graph:
  - All possible advice execution traces.
Evolving Adaptive Programs: Implicit Assumptions

- Evolving the input schema or the traversal strategy or the set of advices can result in:
  - No impact
    - Same smoothed out traversal graph.
    - Adding Color to Cylinder.
  - Minor impact
    - Change to the number of times an advice executes.
    - Adding an inner radius.
  - Drastic impact
    - Change to an advice execution context.
  - Other impacts:
    - The time between advice execution.
Controlling Minor Impacts

- Annotate advices with a cardinality constraint.
  - The method before_radius is executed only once in the context of a Cylinder.
  - $\leq 1$ in Cylinder.
  - There is exactly one path leading from a Cylinder to a radius in the smoothed out traversal graph.
  - > | < | == | <= | >= | != .
Controlling Drastic Impacts

- Method A executes in the context of B and either C or D:
  - in B [C D].
- Method A never executes in the context of B:
  - not in B.
- Method A executes directly after method B:
  - directly in B.
Evolving Adaptive Programs: Stricter Notion of legality (1)

- Behavior must be compatible with the traversal graph:
  - All Advised nodes must be mentioned in the strategy.
  - Every strategy graph node is either advised or can reach an advised node.
- Strategy must be compatible with the input schema:
  - Strategy must look-like the input schema.
  - Strategy must be identical to the input schema after smoothing out non-strategy nodes.
Evolving Adaptive Programs: Stricter Notion of legality (2)

Strategy

Input Schema

Smoothed Input Schema
Evolving Adaptive Programs: Stricter Notion of legality (3)

- How would the traversal graph look like?
Conclusion

- Structure-Shy programs are easy, but dangerous to evolve.
- Structure-Shy programs can be made safer by checking implicit assumptions and enforcing strict notion of compatibility.
- Adaptive programming is a traversal based paradigm for writing structure-shy programs.
Questions, Comments?