Transformation-based Library Adaptation for Embedded Systems

Victor Winter, Azamat Mametjanov, Steven E. Morrison, James A. McCoy, and Gregory L. Wickstrom

Tenth IEEE High Assurance Systems Engineering Symposium
Dallas, Texas
November 14 – 16, 2007
Problem and Solution

The Problem: Enable embedded systems to be programmed at a higher level of abstraction.

The Solution: Make libraries defining high-level abstractions accessible within an embedded programming environment.
An Instance of the Problem

- Computational Environment:
  - The SCORE Processor – A JVM-based platform designed for use in embedded systems.

- Specific Considerations:
  - Restricted computing capabilities – For example, floating point operations are not supported.

- Target Libraries:
  - To the extent possible, we want to make general purpose libraries such as `java.lang` and `java.util` available to the embedded systems programmer in this environment.
The SCORE Processor
A Quick Summary

- The software that we want to adapt is a **Java library**.
- The computational restriction we are considering is a **floating point operation**.
- The target platform is the **Score processor**.
Our Approach to Library Adaptation

- Is based on removing (i.e., filtering out) components of a library that depend on unsupported computations.

- Assumes that library components are loosely coupled.

- **DISCLAIMER:** We do not attempt adaptation that is based on re-implementing components.
Adaptation is Similar to Program Slicing

The approach presented is conceptually similar to program slicing where:

- One is given program $p$ and an initial set of constructs $R$.

- One produces a output program $p'$ that is obtained by removing, from $p$, all code depending either directly or indirectly on $r$ when $r \in R$. 
A General Purpose Approach

Our approach is **general-purpose** in the sense that:

- The techniques can be **adapted** to any language.
- For a given language, the set $R$ can be arbitrary.
Theoretical Considerations

An understanding of removal. That is, the UNIT of removal.

- Well-formed
  - syntactically
  - semantically

- Conservative approximations – remove too much rather than too little (to assure semantic consistency)

- Safeguards
  - the Java compiler
  - code inspection
Syntactic Considerations

```plaintext
int x = (int) 1.0;  \rightarrow  \text{---}

double x;          \rightarrow  \text{---}

float foo() {...}  \rightarrow  \text{---}

if (y < 1.0) x = 0; \rightarrow  if (y < \text{---}) x = 0;
```
Semantic Considerations

```java
int x = 0;
int f() { double x = 1.0; return (int) x; }
→
int x = 0;
int f() { return (int) x; }
```
Theoretical Considerations

An understanding of dependency.

► A direct dependency is recognized at the syntactic level.

► A transitive dependency is recognized at the semantic level and involves analysis that is based on scope and type.
Semantic Considerations

int ask() {return (int)answer();}
double answer(){return 3.33;}

→

int ask() {return (int)answer();}
Practical Considerations

- Infrastructure supporting **automatic activities**: parser

- Infrastructure supporting **manual activities** (e.g., code inspection):
  - preservation of comments
  - pretty-printer (i.e., un-parser)
A Transformation-based Perspective

- A Java library is modelled as a set of classes:

\[ \mathcal{Lib} = \{ \text{Class}_1, \ldots, \text{Class}_n \} \]

- The adaptation of \( \mathcal{Lib} \) is viewed primarily in terms of the adaptation of individual classes, with analysis results exchanged between classes.
Let $C$, with or without subscripts, denote an arbitrary Java class. The adaptation of a class is viewed from a transformational perspective as follows:

$$C_{initial} \xrightarrow{T_1} C_1 \xrightarrow{T_2} \ldots \xrightarrow{T_{final}} C_{final}$$

where $T_i$ denotes a transformation step that removes a portion of the class.
Constraints on Transformation

1. Transformational steps should yield results that are syntactically well-formed.

2. The result of transforming $C_{\text{initial}}$ should yield a class $C_{\text{final}}$, with respect to which $C_{\text{initial}}$ is consistent.

   $$C_{\text{final}} \sqsubseteq C_{\text{initial}}$$

3. $C_{\text{final}}$ must be compatible with the SCORE platform.

4. Removal must be minimal; otherwise the empty class would be an acceptable adaptation.
Removal-oriented Abstractions for Java

- A Java library is modelled as a set of classes (C).

- A Java class is modelled as a set of members.

```java
class name {
    inner classes
    fields
    methods
    constructors
    instance initializers
    static initializers
}
```

- The unit of removal is defined to be a class member.
The Syntactic Categories for Class Members

M ::= x | mb

x ::= inner-classes

mb ::= m | b

m ::= field-declaration
    | method-declaration
    | constructor-declaration

b ::= instance-initializer
    | static-initializer

This notation enables us to describe terms in an abstract fashion.
An Abstract Transformation

- Henceforth, we will view classes as terms.

- We will write $C[\ldots t \ldots]$ to denote an occurrence of the term $t$ within the class $C$.

- The removal of the subterm $t$ from the term $C$ as follows:

$$C[\ldots t \ldots] \overset{T}{\rightarrow} C[\ldots \epsilon \ldots]$$
The Axioms and Rule Defining $R$

$\text{FpLiteral} \in R_C$

$\text{Modifier}[\text{strictfp}] \in R_C$

$\text{BasicType}[\text{float}] \in R_C$

$\text{BasicType}[\text{double}] \in R_C$

$C[\ldots m \ldots] \quad t \in R_C \quad m[\ldots \hat{t} \ldots]$

$\text{ref-to}(m) \in R_C$
Example: ref-to

class A {
    double x = 1.0;
    int y;
    int z;
    { int x = (int)this.<span class="hl">x</span>; y = x; }
    { z = (int)<span class="hl">x</span>; }
}
Removal Rules

\[
\frac{C[\ldots mb \ldots]}{t \in R_C \quad mb[\ldots \hat{t} \ldots]} \quad C[\ldots mb \ldots] \rightarrow C[\ldots \epsilon \ldots]
\]

(T-adapt1)

\[
\frac{C[\ldots x \ldots]}{t \in R_C \quad x[\ldots \hat{t} \ldots]} \quad x[\ldots \hat{t} \ldots] \rightarrow x[\ldots \epsilon \ldots]
\]

(T-adapt2)
Example – Input

```java
class Cafe {
    float fabulous = 0xCAFEBAE; // catch me if you can
    int innocent = (int) fabulous; // masquerade as int

    Cafe(int innocent) {
        this. fabulous = 0xDD; // explicit ref to FP field
        this.innocent = innocent;
    }

    int environment(int innocent) { // re-decl in params
        innocent = 0xFACADE;
        return innocent;
    }

    int environment() {
        int harmless = innocent; // use comes before decl
        int innocent = harmless;
        return innocent;
    }

    int order(int choice) {
        int waitress;
        if (choice == 0xC0FFEE) {
            int innocent = 0xBABE; // re-decl in local if-block
            waitress = innocent;
        } else {
            waitress = innocent; // indirect ref to FP field
        }
        return waitress;
    }
}
```
Example – Output

class Cafe {
    int environment(int innocent) {
        // re-decl in params
        innocence = 0xFACADE;
        return innocent;
    }
}

SCORE: Unsupported Feature Profile

**synchronized** serialized access to data or execution of code; compiles to unsupported opcodes

**volatile** thread coherence; multi-threading is not supported

**transient** I/O persistence; persistence is not supported

**assert** assertion-checking with dependencies on reflection features of a JVM; reflection is not supported

**native** code implemented in target platform’s native code; not supported unless implemented on the embedded platform

**floating-point** keywords `float`, `double`, `strictfp`

**other** unsupported classes: e.g. Classloader, Compiler, etc.
The util library consists of 96 source files with a number of files containing more than 1000 (SLOC).

<table>
<thead>
<tr>
<th></th>
<th>sloc</th>
<th>orig</th>
<th>auto</th>
<th>man</th>
<th>end</th>
<th>sloc’</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrays</td>
<td>4173</td>
<td>161</td>
<td>43</td>
<td>1</td>
<td>117</td>
<td>2788</td>
</tr>
<tr>
<td>Date</td>
<td>1315</td>
<td>53</td>
<td>45</td>
<td>0</td>
<td>8</td>
<td>216</td>
</tr>
<tr>
<td>Total</td>
<td>42343</td>
<td>2177</td>
<td>442</td>
<td>40</td>
<td>1695</td>
<td>29199</td>
</tr>
</tbody>
</table>
An Example Requiring Manual Intervention

The class Throwable required manual adjustment.

- The removal of 4 constructors in Throwable triggered compile errors in all Error and Exception classes.

- The constructors were put back; statements within the constructors referencing native code were manually removed (fillInStackTrace())

- Root cause: cohesion

- Solution: minor re-implementation
Initial investment of 3 man/months:

- 132 SLOC of Java lexer
- 277 SLOC of Java BNF
- 2655 SLOC of Java pretty-printer
- 1133 SLOC of transformation code
Payoff

- 1 day automated vs. 45 days manual transformation of the `java.util` library.

- We have adapted:
  - `java.util` with a \( \approx 91\% \) migration rate
  - `java.lang` with a \( \approx 98\% \) migration rate

- **Reuse**: Keep in mind that new versions of `java.lang` and `java.util` are released at regular intervals (approximately every 12 -18 months). 

---

Library Adaptation

Victor Winter, Azamat Mametjanov, Steven E. Morrison, James A. McCoy, and Gregory L. Wickstrom
Remarks

Industrial rule-of-thumb [Akers et al.]: investment into automation is well-justified if 75% of legacy code is migrated using automation.
Conclusion

- Embedded Java programming needs library support.

- Manual library adaptation is resource-intensive.

- Program transformations are well suited to this kind of activity.

- Re-targeting of the libraries to heterogeneous Java-based embedded platforms with different instruction sets is easily handled.
The End