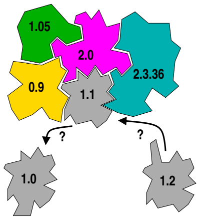


Predicting Problems Caused by Component Upgrades

Stephen McCamant and Michael D. Ernst

Program Analysis Group

{smcc,mernst}@CSAIL.MIT.EDU

Upcoming Zeminars

- Future Zeminars will be here in room 518, except as noted
  - Monday August 25th 3pm: Jonathan Edwards on a type system for Alloy
  - Monday September 1st 3pm: No Zeminar, Labor Day
  - Monday September 8th: Future schedule TBA

Outline

- The upgrade problem
- Solution: Compare observed behavior
- Comparing observed behavior (details)
- Example: Sorting and `swap`
- Case study: Perl modules
- Scaling to larger systems
- Conclusion

Upgrade safety

- A system uses version 1.1 of a component
- Might version 1.2 cause the system to misbehave?

(The general question is undecidable)

Terminology

- The component might be any separately developed piece of software
- The application uses the component
- The vendor develops the component
- The user integrates the component with the rest of the application

Previous solutions

- Integrate new component, then test
  - Resource intensive
- Vendor tests new component
  - Impossible to anticipate all uses
  - User, not vendor, must make upgrade decision
- Static analysis to guarantee identical or subtype behavior
  - Difficult to provide adequate guarantees

Behavioral subtyping

- Behavioral subtyping [Liskov 94] guarantees behavioral compatibility
  - Provable properties about supertype are provable about subtype
  - Operates on human-supplied specifications
- Behavioral subtyping is too strong
  - OK to change aspects that the application does not use
- Behavioral subtyping is too weak
  - An application may depend on implementation details

Outline

- The upgrade problem
- Solution: Compare observed behavior
- Comparing observed behavior (details)
- Example: Sorting and `swap`
- Case study: Perl modules
- Scaling to larger systems
- Conclusion

Run-time behavior comparison

- Compare run-time behaviors of component
  - Old component, in context of the application's use
  - New component, in context of vendor test suite
- Compatible if the vendor tests all the functionality that the application uses (and gets the right output)

Operational abstraction

- Abstraction of run-time behavior of component
- Set of program propertiesÐ mathematical statements about component behavior
- Syntactically identical to formal specification
- Consists of pre- and post-conditions
- Can compare via logical implication

Dynamic invariant detection

- Recover likely invariants by examining runtime values, using Daikon `http://pag.lcs.mit.edu/daikon`
- Output is logical statements describing program behavior (potential invariants)
- Algorithm:
  - Conjecture all properties from a large grammar
  - At each dynamic program point, discard falsified properties
  - Eliminate implied and statistically unjustified statements
  - To find conditional properties (x is even $\Rightarrow a[x] = 0$), use subsets of data

Outline

- The upgrade problem
- Solution: Compare observed behavior
- Comparing observed behavior (details)
- Example: Sorting and `swap`
- Case study: Perl modules
- Scaling to larger systems
- Conclusion

Testing upgrade compatibility

1. User computes operational abstraction of old component, in context of application's use
2. Vendor computes operational abstraction of new component, over test suite
3. Vendor supplies operational abstraction along with new component
4. User compares operational abstractions using an automated tool

Verifying unchanged behavior

- The operational abstraction of the new version, with the vendor's tests, consists of pre- and post-conditions $\mathrm{Pre}_{\mathrm{test}}$ and $\mathrm{Post}_{\mathrm{test}}$
- The abstraction of the old version, in the context of the application, is $\mathrm{Pre}_{\mathrm{app}}$ and $\mathrm{Post}_{\mathrm{app}}$
- For the upgrade to be safe, verify that $\mathrm{Pre}_{\mathrm{app}}$ and the new component imply $\mathrm{Post}_{\mathrm{app}}$

New abstraction must be stronger

Rest of application
$\Downarrow$
$\mathrm{Pre}_{\mathrm{app}}$
$\Downarrow$
$\boxed{\text{Old component}}$
$\Downarrow$
$\mathrm{Post}_{\mathrm{app}}$
$\Downarrow$
Rest of application

New abstraction must be stronger

Rest of application
$\Downarrow$
$\mathrm{Pre}_{\mathrm{app}}$

$\mathrm{Post}_{\mathrm{app}}$
$\Downarrow$
Rest of application

New abstraction must be stronger

Rest of application
$\Downarrow$
$\mathrm{Pre}_{\mathrm{app}}$

$\boxed{\text{New component}}$

$\mathrm{Post}_{\mathrm{app}}$
$\Downarrow$
Rest of application

New abstraction must be stronger

- We want to check that $\mathrm{Pre}_{\mathrm{app}} \Rightarrow \mathrm{Post}_{\mathrm{app}}$

Rest of application
$\Downarrow$
$\mathrm{Pre}_{\mathrm{app}}$
$\Downarrow$
$\boxed{\text{New component}}$
$\Downarrow$
$\mathrm{Post}_{\mathrm{app}}$
$\Downarrow$
Rest of application

New abstraction must be stronger

- We want to check that $\text{Pre}_{\text{app}} \Rightarrow \text{Post}_{\text{app}}$
- We know that $\text{Pre}_{\text{test}} \Rightarrow \text{Post}_{\text{test}}$

$$
\begin{array}{ccc}
\text{Rest of application} & & \\
\Downarrow & & \\
\text{Pre}_{\text{app}} & & \text{Pre}_{\text{test}} \\
 & & \Downarrow \\
\boxed{\text{New component}} & & \boxed{\text{New component}} \\
 & & \Downarrow \\
\text{Post}_{\text{app}} & & \text{Post}_{\text{test}} \\
\Downarrow & & \\
\text{Rest of application} & &
\end{array}
$$

New abstraction must be stronger

- We want to check that $\text{Pre}_{\text{app}} \Rightarrow \text{Post}_{\text{app}}$
- We know that $\text{Pre}_{\text{test}} \Rightarrow \text{Post}_{\text{test}}$

$$
\begin{array}{ccc}
\text{Rest of application} & & \\
\Downarrow & & \\
\text{Pre}_{\text{app}} & \Rightarrow & \text{Pre}_{\text{test}} \\
 & & \Downarrow \\
\boxed{\text{New component}} & & \boxed{\text{New component}} \\
 & & \Downarrow \\
\text{Post}_{\text{app}} & \Leftarrow & \text{Post}_{\text{test}} \\
\Downarrow & & \\
\text{Rest of application} & &
\end{array}
$$

- Sufficient condition:

  $(\text{Pre}_{\text{app}} \Rightarrow \text{Pre}_{\text{test}}) \land (\text{Post}_{\text{test}} \Rightarrow \text{Post}_{\text{app}})$

Comparing operational abstractions

- Sufficient, but usually false:

  $(\text{Pre}_{\text{app}} \Rightarrow \text{Pre}_{\text{test}}) \land (\text{Post}_{\text{test}} \Rightarrow \text{Post}_{\text{app}})$

$$
\begin{array}{cccccc}
 & \begin{array}{|c|}\hline \text{Pre}_{\text{app}} \\ \hline x \text{ is even} \\ \hline\end{array} & \Rightarrow & \begin{array}{|c|}\hline \text{Pre}_{\text{test}} \\ \hline x \text{ is an integer} \\ \hline\end{array} & \\
\text{[Application]} & \Downarrow & & \Downarrow & [\texttt{inc} \text{ test suite}] \\
 & \begin{array}{|c|}\hline \text{Post}_{\text{app}} \\ \hline x' = x + 1 \\ x' \text{ is odd} \\ \hline\end{array} & \not\Leftarrow & \begin{array}{|c|}\hline \text{Post}_{\text{test}} \\ \hline x' = x + 1 \\ \\ \hline\end{array} &
\end{array}
$$

- Just right:

  $(\text{Pre}_{\text{app}} \Rightarrow \text{Pre}_{\text{test}}) \land (\text{Pre}_{\text{app}} \land \text{Post}_{\text{test}} \Rightarrow \text{Post}_{\text{app}})$

Highlighting new failures

- This check could reject an ‘upgrade’ of a component to the same version
  - Use of untested behavior (vendor testing insufficient)
  - Abstraction or prover failure
- Repeat comparison, using vendor’s abstraction for old component version
- Especially interested in failures that occur only with the new component abstraction

Reasons for behavioral differences

- Differences between application and test suite uses of component require human judgment
  - True incompatibility
  - Change in behavior might not affect application
  - Change in behavior might be a bug fix
  - Vendor test suite might be deficient
  - It may be possible to work around the incompatibility

Outline

- The upgrade problem
- Solution: Compare observed behavior
- Comparing observed behavior (details)
- Example: Sorting and `swap`
- Case study: Perl modules
- Scaling to larger systems
- Conclusion

Sorting application

```
// Sort the argument into ascending order
static void bubble_sort(int[] a) {
  for (int x = a.length - 1; x > 0; x--) {
    // Compare adjacent elements in a[0..x]
    for (int y = 0; y < x; y++) {
      if (a[y] > a[y+1])
        swap(a, y, y+1);
    }
  }
}
```


Swap component

```
// Exchange the two array elements at i and j
static void swap(int[] a, int i, int j) {
  int temp = a[i];
  a[i] = a[j];
  a[j] = temp;
}
```


Upgrade to swap component

```
// Exchange the two array elements at i and j
static void swap(int[] a, int i, int j) {
  a[i] ^= a[j]; // XOR
  a[j] ^= a[i];
  a[i] ^= a[j];
}
```


Compare abstractions

$\underline{\text{Pre}_{\text{app}}}$		
$0 \le i < \text{size}(a) - 1$		$\underline{\text{Pre}_{\text{test}}}$
$1 \le j \le \text{size}(a) - 1$	$\Rightarrow$	$0 \le i \le \text{size}(a) - 1$
$j = i + 1, i < j$		$0 \le j \le \text{size}(a) - 1$
$a[i] > a[j]$		$i \ne j$
bubble_sort application $\Downarrow$		$\Downarrow$ swap test suite
$\underline{\text{Post}_{\text{app}}}$		
$a'[i] = a[j]$		$\underline{\text{Post}_{\text{test}}}$
$a'[j] = a[i]$		$a'[i] = a[j]$
$a'[i] = a'[j-1]$	$\Leftarrow$	$a'[j] = a[i]$
$a'[i] < a'[j]$		

Compare abstractions

$\underline{\text{Pre}_{\text{app}}}$		
$0 \le i < \text{size}(a) - 1$		$\underline{\text{Pre}_{\text{test}}}$
$1 \le j \le \text{size}(a) - 1$	$\Rightarrow$	$0 \le i \le \text{size}(a) - 1$
$j = i + 1, i < j$		$0 \le j \le \text{size}(a) - 1$
$a[i] > a[j]$		$i \ne j$
bubble_sort application $\Downarrow$		$\Downarrow$ swap test suite
$\underline{\text{Post}_{\text{app}}}$		
$a'[i] = a[j]$		$\underline{\text{Post}_{\text{test}}}$
$a'[j] = a[i]$		$a'[i] = a[j]$
$a'[i] = a'[j-1]$	$\Leftarrow$	$a'[j] = a[i]$
$a'[i] < a'[j]$		

$$\text{Pre}_{\text{app}} \Rightarrow \text{Pre}_{\text{test}}$$

Compare abstractions

$\underline{\text{Pre}_{\text{app}}}$		
$0 \le i < \text{size}(a) - 1$		$\underline{\text{Pre}_{\text{test}}}$
$1 \le j \le \text{size}(a) - 1$	$\Rightarrow$	$0 \le i \le \text{size}(a) - 1$
$j = i + 1, i < j$		$0 \le j \le \text{size}(a) - 1$
$a[i] > a[j]$		$i \ne j$
bubble_sort application $\Downarrow$		$\Downarrow$ swap test suite
$\underline{\text{Post}_{\text{app}}}$		
$a'[i] = a[j]$		$\underline{\text{Post}_{\text{test}}}$
$a'[j] = a[i]$		$a'[i] = a[j]$
$a'[i] = a'[j-1]$	$\Leftarrow$	$a'[j] = a[i]$
$a'[i] < a'[j]$		

$$\text{Pre}_{\text{app}} \wedge \text{Post}_{\text{test}} \Rightarrow \text{Post}_{\text{app}}$$

Compare abstractions

$\underline{\text{Pre}_{\text{app}}}$		$\underline{\text{Pre}_{\text{test}}}$
$0 \le i < \text{size}(a) - 1$		
$1 \le j \le \text{size}(a) - 1$	$\Rightarrow$	$0 \le i \le \text{size}(a) - 1$
$j = i + 1, i < j$		$0 \le j \le \text{size}(a) - 1$
$a[i] > a[j]$		$i \ne j$
$\boxed{\texttt{bubble\_sort}\text{ application}}$ $\Downarrow$		$\Downarrow$ $\boxed{\texttt{swap}\text{ test suite}}$
$\underline{\text{Post}_{\text{app}}}$		
$a'[i] = a[j]$		$\underline{\text{Post}_{\text{test}}}$
$a'[j] = a[i]$		$a'[i] = a[j]$
$a'[i] = a'[j-1]$	$\Leftarrow$	$a'[j] = a[i]$
$a'[i] < a'[j]$		

Upgrade succeeds

Another sorting application

```
// Sort the argument into ascending order
static void selection_sort(int[] a) {
  for (int x = 0; x <= a.length - 2; x++) {
    // Find the smallest element in a[x..]
    int min = x;
    for (int y = x; y < a.length; y++) {
      if (a[y] < a[min])
        min = y;
    }
    swap(a, x, min);
  }
}
```


Compare abstractions

$\underline{\text{Pre}_{\text{app}}}$		$\underline{\text{Pre}_{\text{test}}}$
$0 \le i < \text{size}(a) - 1$		$0 \le i \le \text{size}(a) - 1$
$i \le j \le \text{size}(a) - 1$	$\Rightarrow$	$0 \le j \le \text{size}(a) - 1$
$a[i] \ge a[j]$		$i \ne j$
$\boxed{\texttt{selection\_sort}\text{ application}}$ $\Downarrow$		$\Downarrow$ $\boxed{\texttt{swap}\text{ test suite}}$
$\underline{\text{Post}_{\text{app}}}$		
$a'[i] = a[j]$		$\underline{\text{Post}_{\text{test}}}$
$a'[j] = a[i]$		$a'[i] = a[j]$
$a'[i] = a'[j-1]$	$\Leftarrow$	$a'[j] = a[i]$
$a'[i] \le a'[j]$		

Compare abstractions

$\underline{\text{Pre}_{\text{app}}}$		$\underline{\text{Pre}_{\text{test}}}$
$0 \le i < \text{size}(a) - 1$		$0 \le i \le \text{size}(a) - 1$
$i \le j \le \text{size}(a) - 1$	$\not\Rightarrow$	$0 \le j \le \text{size}(a) - 1$
$a[i] \ge a[j]$		$i \ne j$
$\boxed{\texttt{selection\_sort}\text{ application}}$ $\Downarrow$		$\Downarrow$ $\boxed{\texttt{swap}\text{ test suite}}$
$\underline{\text{Post}_{\text{app}}}$		
$a'[i] = a[j]$		$\underline{\text{Post}_{\text{test}}}$
$a'[j] = a[i]$		$a'[i] = a[j]$
$a'[i] = a'[j-1]$	$\Leftarrow$	$a'[j] = a[i]$
$a'[i] \le a'[j]$		

Upgrade fails:

$\text{Pre}_{\text{app}} \not\Rightarrow \text{Pre}_{\text{test}}$, $i \ne j$ not valid

Outline

- The upgrade problem
- Solution: Compare observed behavior
- Comparing observed behavior (details)
- Example: Sorting and `swap`
- Case study: Perl modules
- Scaling to larger systems
- Conclusion

CPAN case studies

Module	From Version	To Version	Upgrade is	Relevant Method
Math-BigInt	1.40	1.42	Unsafe	`bcmp()`
Math-BigInt	1.47	1.48	Safe	`bmul()`
Date-Simple	1.03	2.00	Unsafe	Constructor
Date-Simple	1.03	2.03	Unsafe	Constructor
Date-Simple	2.00	2.03	Safe	Constructor

- The “applications” were other CPAN modules
- We supplied simple randomized test suites

BigFloat::bcmp() results

- An upgrade from 1.40 to 1.42 is not behavior preserving. Our tool finds an inconsistency caused in part by a bug that also causes the following difference:
  - In 1.40, `bcmp(1.67, 1.75)` $\Rightarrow 0$
  - In 1.42, `bcmp(1.67, 1.75)` $\Rightarrow -1$
- Our tool also declares a downgrade from 1.42 to 1.40 to be unsafe, since
  - In 1.42, `bcmp` returns -1, 0, or 1
  - In 1.40, `bcmp` returns any integer

BigFloat::bmul() results

- In from version 1.47 to 1.48, the `bmul` floating-point multiplication routine was partially rewritten
- The system verifies that this change was behavior-preserving for Math-Currency
- Caveat:
  - Daikon required four hand-written splitting conditions to capture special-case behavior

Date::Simple results

- Date-Simple 2.00 and 2.03 are compatible with each other, but not with 1.03
- This incompatibility is caused by a bug in 1.03
  - The constructor relies on undefined behavior of POSIX's `mktime`, and fails to check for an error return value

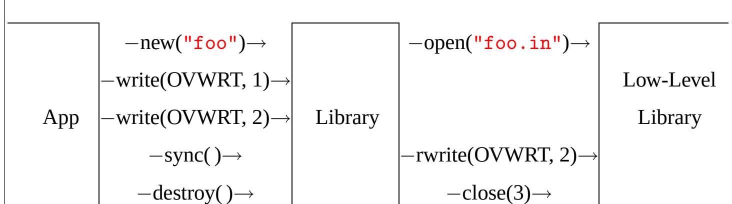
Outline

- The upgrade problem
- Solution: Compare observed behavior
- Comparing observed behavior (details)
- Example: Sorting and `swap`
- Case study: Perl modules
- Scaling to larger systems
- Conclusion

Challenges of larger systems

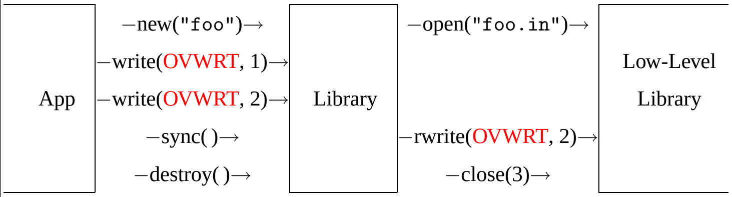
- There may be no formal test suite available
  - Treat other applications' use as tests
- Behavior may depend on other system state
  - Use program's own methods to access
- Error conditions may be unpredictable
  - Treat exceptional returns as a special case
- Components may only work when upgraded together (e.g., producer and consumer)
  - Characterize inter-component communication…

Discovering cross-component links



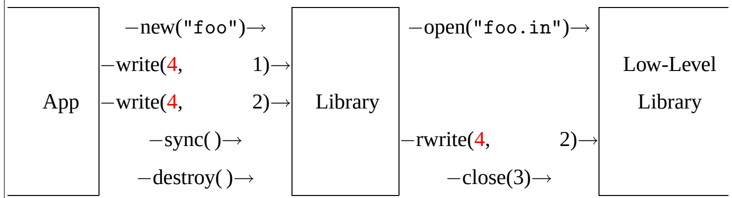
- Match argument values with other recent calls to guess data flow
  - `open_file = new_name + ".in"`

Discovering cross-component links



- Recognize common interfaces
  - `write_mode` one of $\{\mathrm{OVWRT}, \mathrm{APPND}\}$
  - `write_mode` $=$ `rwrite_mode`
  - `rwrite_mode` one of $\{\mathrm{OVWRT}, \mathrm{APPND}\}$

Discovering cross-component links



- Allow consistent changes
  - `write_mode` one of $\{4, 8\}$
  - `write_mode` $=$ `rwrite_mode`
  - `rwrite_mode` one of $\{4, 8\}$

Linux C library case study

- Unmodified binary applications and library versions
- Capture behavior by dynamic-library interposition
- Can efficiently compare abstractions with hundreds of functions
- Main challenge: avoiding false alarms

Getting to Yes

- Rejecting an upgrade is easier than approving it
- Application postconditions may be hard to prove
  - Can explain the reason for the rejection
  - Highlight only cross-version failures
- Grammar of operational abstractions may be inappropriate
  - Theorem prover may not be powerful enough
- Application's use may be a novel special case
  - Improve automatic selection of splitting conditions

Outline

- The upgrade problem
- Solution: Compare observed behavior
- Comparing observed behavior (details)
- Example: Sorting and `swap`
- Case study: Perl modules
- Scaling to larger systems
- Conclusion

Contributions

- New technique for early detection of (some) upgrade problems
- Compares run-time behavior of old and new components
- Technique is
  - Application-specific
  - Lightweight, Pre-integration
  - Source-free, Specification-free
  - Blame-neutral
  - Output-independent