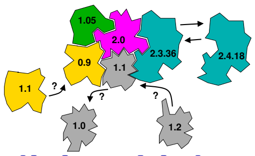


Formalizing Lightweight Verification of Software Component Composition

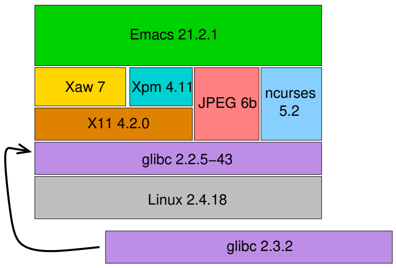
Stephen McCamant and Michael D. Ernst

{smcc,mernst}@CSAIL.MIT.EDU

http://pag.csail.mit.edu/

MIT Computer Science and Artificial Intelligence Laboratory

Upgrade safety



- Will it still work with this new component?
- We have a system that vetted this upgrade

Overview

- Technique assesses upgrade safety
  - Unsound tool builds abstractions
  - Check property of combined abstractions
- Goal: prove checking step sound
- Results to date:
  - Formalization of upgrade safety problem
  - Approach for relative soundness proof
  - Improvements to previous algorithm
  - Proof outline for soundness result

Our approach

Abstractions:

- should be stated in an expressive language
- should describe concrete implementations
- should be created automatically
- need not be sound over arbitrary executions

Comparison of run-time behavior

- 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
- Set of program properties — mathematical statements about module behavior
- For x++:
  - Precondition: x is an integer
  - Postcondition: $x' = x + 1$
- Depends on how the module is used

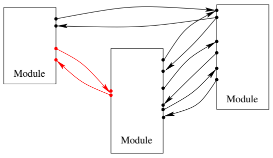
Operational abstraction

- Abstraction of run-time behavior
- Set of program properties — mathematical statements about module behavior
- For x++, **used on even values**:
  - Precondition: x is **even**
  - Postcondition: $x' = x + 1$, x' **is odd**
- Depends on how the module is used

Operational abstraction

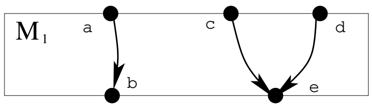
- Abstraction of run-time behavior
- Set of program properties — mathematical statements about module behavior
- For x++, **used on even values**:
  - Precondition: x is **even**
  - Postcondition: $x' = x + 1$, x' **is odd**
- Depends on how the module is used
- Obtained using the Daikon tool

Modules: inputs and outputs



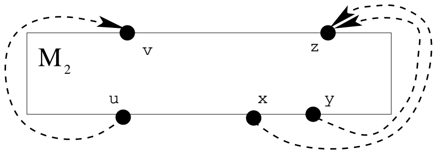
- Consider just the behavior of modules at their boundaries
- The outputs of one module are connected to the inputs of another via procedure calls and returns
- Connections just represent identity

Flow and summary relations



Flow relations $M_1(b \mid a)$, $M_1(e \mid c, d)$

$b.x > a.y, \; e.y = c.y + e.z$



Summary relations $\overline{M}_2(v \mid u)$, $\overline{M}_2(z \mid x, y)$

$v.\mathrm{ret} = u.\mathrm{arg} + 3, \; x.i \neq z.j \cdot y.j$

Formalizing the upgrade condition

- Combined flow relations must imply summaries
- Do we have the right combination?
- Snag: what formal property to aim for?
- Describe idealized version that should be sound
  - Postulate existence of sound abstractions
- Final result is relative soundness, up to abstractions

Abstraction and formalization

Concrete program

$\Downarrow$

Daikon

$\Downarrow$

Operational abstraction

Abstraction and formalization

Concrete program $\Rightarrow$ Formal program (in a simple language)

$\Downarrow$ $\Downarrow$

Daikon $\exists$

$\Downarrow$ $\Downarrow$

Operational abstraction $\Rightarrow$ Idealized abstraction (sound)

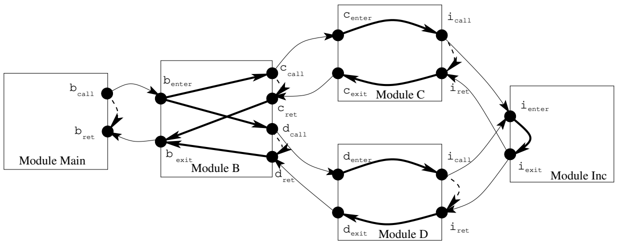
A formal imperative language

- Consider a simple language:

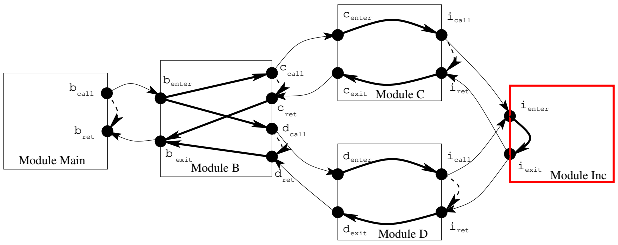
$$C ::= C \,;\, C \mid \mathsf{skip} \mid \mathsf{assert}(P) \mid v := E$$
$$\mid \mathsf{if}\ P\ \mathsf{then}\ C\ \mathsf{else}\ C \mid v := M.f(v_1, \ldots, v_k)$$

- Procedures f are grouped in modules M that share some variables
- ‘assert’ doesn’t affect control flow
- Goal: Correct execution without assertion failure

Example of modules

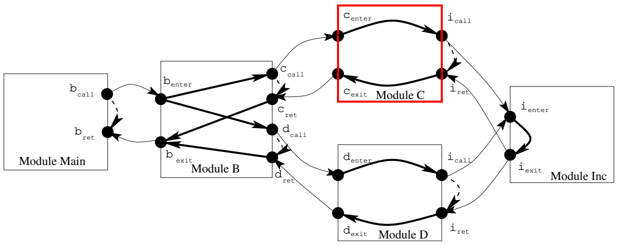


Example of modules



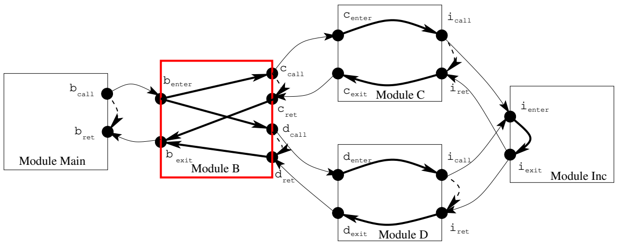
Inc.$i(x)$: `r := x + 1`

Example of modules



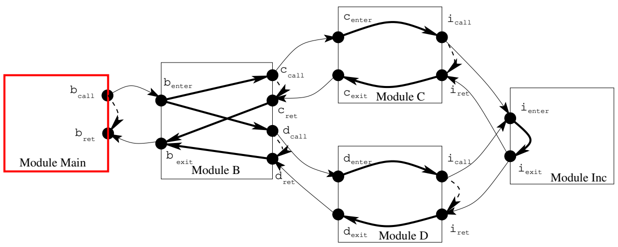
C.$c(v)$: `r := Inc.i(v)`

Example of modules



B.$b(y)$: `r := C.c(2*y) + D.d(2*y + 1)`

Example of modules



Main.$m(x)$: `r := B.b(x);`
`assert(r > 4*x)`

Ideal flow relations

- Idealized flow relations are sound over a module's code
- Valid properties for any possible module inputs
- Some represent pure **data flow**
- Others also model **control flow**, with a 'guarding condition'

Reality vs. formalism

- Real operational abstractions are correct only with respect to observed inputs
  - 'if $x = 271828$ then $y := 2$ else $y := 1$' might produce '$y = 1$'
- Idealized abstractions come are sound with respect to any input
  - Could be '$y = 1 \vee y = 2$'

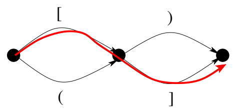
Ideal summary relations

- Idealized summary relations guarantee no assertion failures
- If they hold over module inputs, assertions in the module will succeed
- Capture the well-tested subset of behavior
- Includes program input-output relation as a special case

Consistency condition

- If holds, combined system satisfies expectations
- $(\bigwedge_i \phi_i) \Rightarrow \sigma$
  - Flow relations ϕ_i
  - Summary relation σ
- To construct:
  - Find relevant flow relations
  - Transform relations for sound combination
  - Conjoin

Context-free language reachability



- Graph with edges labelled by symbols
- Context-free language over the symbols
- Is there a path from u to v whose labels are a word of the language?
- Determine by dynamic programming

Selecting relevant flow relations

- Label calls and returns with parenthesis kinds
- Exclude paths with mismatched returns
- Data-flow edges can reset the ‘stack’
  - Gadget allows arbitrary returns then calls
- Take anything on a CFL path

Soundness transformations

- Goal: consistent variable references, so conjunction $(\bigwedge_i \phi_i)$ is sensible
- **Guard** conditional flows
- **Duplicate** procedures by calling context
- **Mix** data flow between replicas

Guarding conditional control flow

- Suppose u is only sometimes followed by v
- From v, looks like $\psi(u, v)$
- Rewrite as $\gamma(u) \Rightarrow \psi(u, v)$ where γ holds only on those instances of u followed by v.

Duplication by calling context

- If Inc.i_{exit} is procedure exit and $C.i_{\text{ret}}$ is return in caller, Inc.$i_{\text{exit}}.r = C.i_{\text{ret}}.x$
- Similarly Inc.$i_{\text{exit}}.r = D.i_{\text{ret}}.x$ for second call site
- Uh-oh, but $C.i_{\text{ret}}.x \neq D.i_{\text{ret}}.x$ in general
- Avoid problem if every call is distinct

Mixing data flow

- After duplicating, what about pure data flow (e.g. from shared state)?
- Conservatively allow flow between any replicas
- Every input gets at least one output, but not vice-versa

Soundness proof outline

- Suppose $(\bigwedge_i \phi_i) \Rightarrow \sigma$
- Each ϕ_i is sound by assumption
- Conjunction is legitimate, by transformations
- LHS is true, so RHS (σ) must be true
- Summary relation truth implies safety

Contributions

- Model and algorithm correct bugs in previous versions
- Formalization for soundness checking
- Complete proof for single component case (see paper)
- Proof outline for general case

Future work

- Avoid need for duplication
  - Sound treatment of repeated calls
- Complete detailed soundness proof
- Add more language features
  - Loops, recursion, higher-order procedures

Questions?