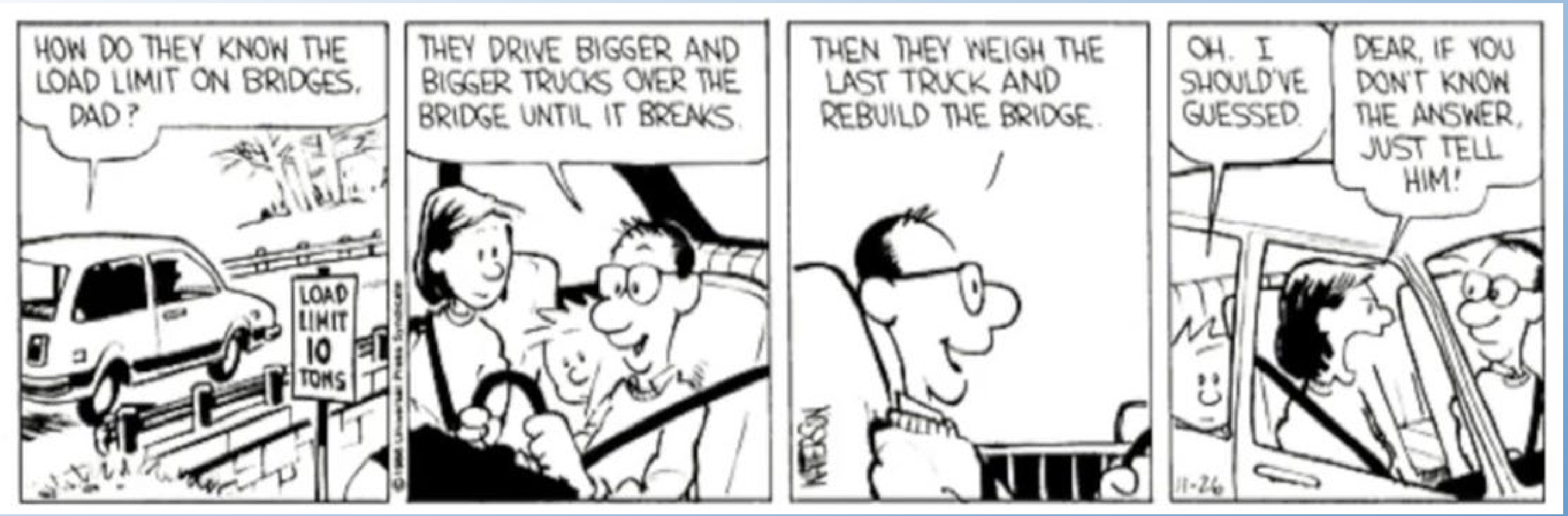


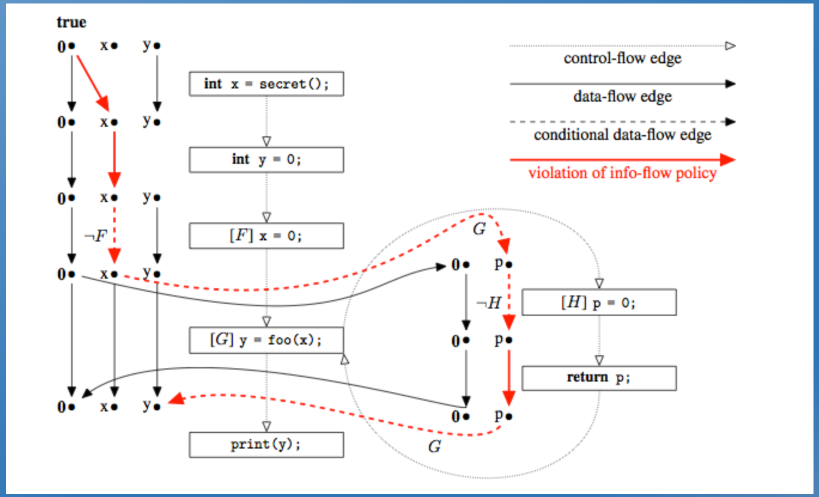
Static Analysis II

CSCI2340: Software Engineering of Large Systems

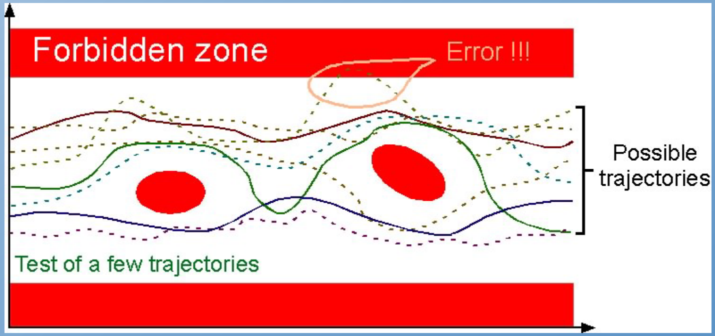
Steven P. Reiss

Type Checking Requires Work

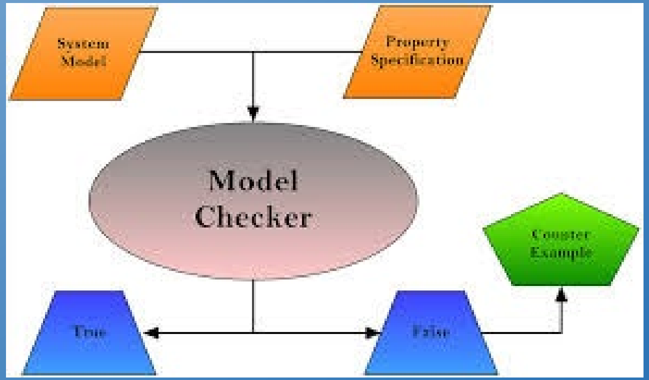
- Type checking doesn't take flow of control into account
  - Considers the declared type, not the actual types
    - Routine without a declaration can return NULL, even if it wouldn't
    - Doesn't know what actual types might occur a a given point
  - Methods use their declared type, not what is effectively passed in
  - Need to annotate all intermediate methods and local declarations to make it work
    - And there can be a lot of them
- Want to understand the actual types occurring in execution
  - And the actual subtypes
  - More precise than the declared types
  - This provides more accurate analysis
  - With many fewer annotations
- Also want to track program states
  - Properties that aren't data-specific
  - More general safety conditions



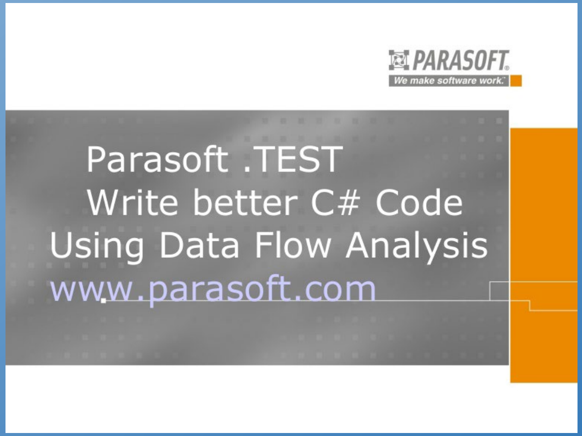
Beyond Type Checking



- Check data-oriented properties without all the annotations
  - Annotate only source and sink
- Check program-oriented properties
  - That aren't related to a type
- Approaches exist
  - Using **Data Flow Analysis** to propagate the properties
    - Running a model of the program with symbolic values
  - Using **Model Checking** to validate a safety property
    - Map program to an automata to a temporal logic predicate showing next state
    - Map safety condition to an automata to a temporal logic formula
    - Create a formula that says the program can enter an error state
    - Prove that formula is correct
  - Specialized approaches for specific properties (e.g., thread safety) also exist

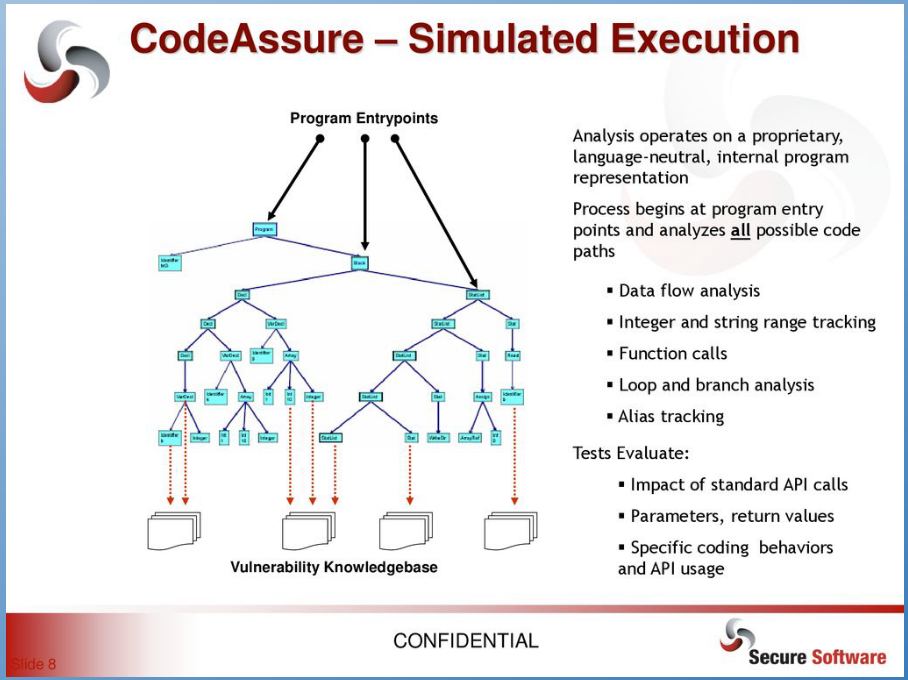


Data Flow Analysis: A More General Solution



- Determines what code can be reached
  - And what code orderings are possible
  - Considers different calling situations
- Determines what values that can reach each point
  - Value is an abstract representation of program data
    - Specific type & subtype and finite value representation
  - Tracked for all instances: stack, local variables, fields, globals
  - Values can then represent types (since they are typed)
    - Actual types, not declared types
- Does implicit type checking
  - Without having to specify the intermediate constraints

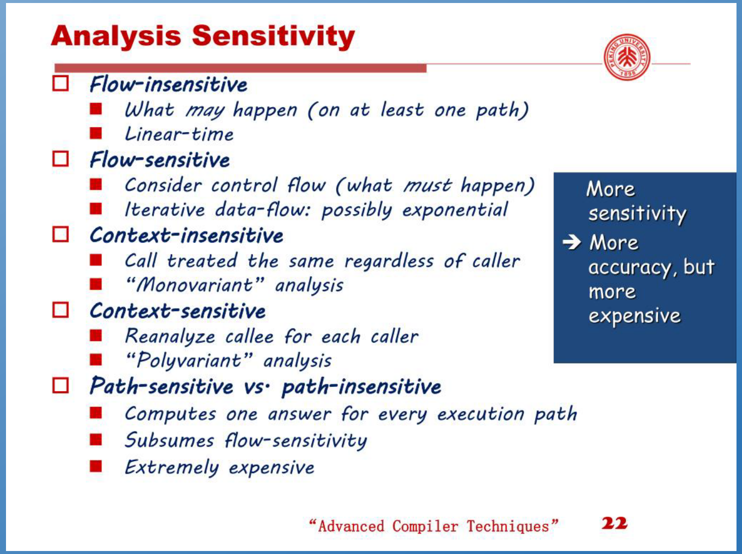
Data Flow Analysis



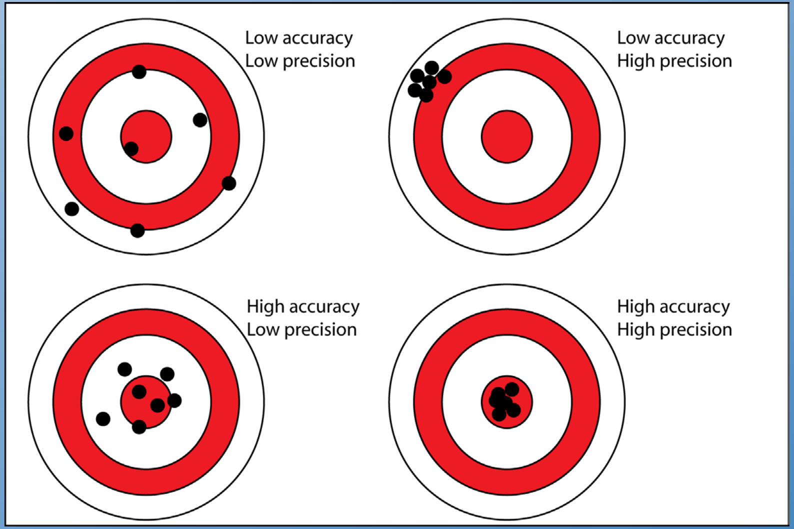
- Can be used to check state-based safety conditions
  - Associate set of safety condition states with each program point
  - Propagate these through the execution
    - Changing states based on potential program events
  - Any point with an associated error state represents a potential error
- Can be used to check data-based safety conditions
  - Associate data states (subtypes) with each data element
    - Extend the notion of type to include base type plus subtypes
  - Propagate these through the execution
    - Computing and adjusting types and subtypes appropriately
  - Determine when data is used in an incorrect manner – subtypes conflict
- Can be used for more sophisticated checks
  - Tracking integer values (constraints), string lengths and contents
  - Check for buffer overflows, possible exceptions, invalid URLs and regular expressions, …
- Still limited
  - It is an approximation – program and values are modeled, not exact
  - Checking for null violations is still difficult
  - Doesn't handle thread interleaving or locking (but handles threading)

Sensitivity Levels of Data Flow Analysis

- Path-sensitive
  - Consider each different program path separately
    - First time through a loop versus 2nd time, …
  - More accurate, but can get very expensive (recall path coverage)
- Flow-sensitive
  - Consider control flow within a method
  - But merge states when control flow comes together
- Flow-insensitive
  - Just look at all possible computations done in the method
    - In any order and without considering conditions, etc.
    - Much less expensive
  - Useful for some analyses
  - Too imprecise for safety condition checking
- Context-sensitive
  - Consider different call sites for methods as different methods
  - Otherwise merge inputs to a method; pass merged outputs back to each call site
  - Can also have partial context sensitivity

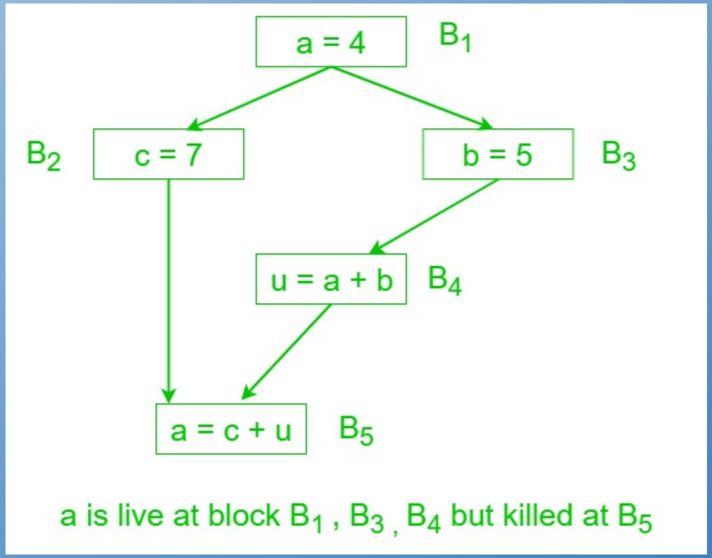


Precision & Accuracy



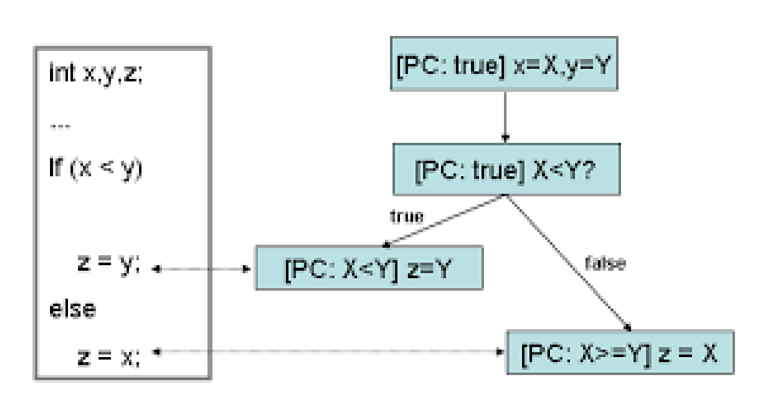
- A flow analysis is precise if
  - Every possible execution is included in the flow analysis
  - The analysis covers all executions and then some
- A precise analysis guarantees
  - If it shows there is no problem, the program is safe
  - If it shows a problem, the program MAY not be safe
- A flow analysis is accurate if
  - An error found in the analysis indicates an error in the program
  - But no errors in the analysis does not indicate the program is safe
  - Much more difficult to achieve, much less useful
- You want both accuracy and precision
  - Precision is probably more important
  - But there are tradeoffs between them
  - And both tradeoff with efficiency

Compiler Data Flow Analysis



- Data flow analysis is used for compiler optimization
  - For specific properties
    - What variables are alive (will be used) at a program point
    - What definitions reach a given program point
    - What expressions are available
    - What values are constants
  - Fix set of bits indicating yes or no
- Bit sets are propagated and computed over the flow graph
  - Over a graph of basic blocks and their connections
  - Using relatively simple Boolean formulas (union, intersection)
- Very efficient computation of information needed
  - In terms of both time and space
- But this doesn’t help with types & subtypes & program properties

Abstract Interpretation

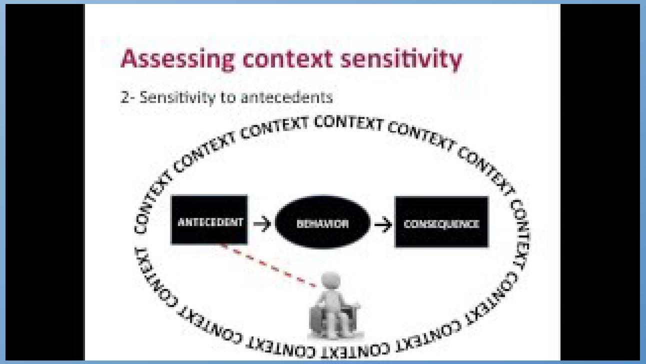


- **Abstract interpretation** based flow analysis
  - Simulate program execution
  - Using a finite state version of the program
  - Starting at all possible starting points
    - Main programs; test cases; …
- Using abstract values rather than actual ones
  - Abstract value: data type with subtypes, source, constraints, subvalues
  - Simplified (finite) versions of data as discussed in prior lecture
    - With type and subtype information
- Keeping track of program states for general safety properties

Abstract Execution Uses a Finite Program

- Calls are not call-return
  - Each method/function has its own execution model
  - Call merges call arguments into parameter values for called model
  - Called models are handled independently
  - Return value of called routine used at call site
- Conditionals can take multiple branches
  - Based on what is known about values
  - Nondeterministic finite state automata
- Associate values with each instruction
  - Values are finite but are always increasing
  - Control flow meeting points merge values
  - Need to reevaluate instructions when values change
- The result is effectively a finite state automata
  - With a finite set of values, evaluation will terminate

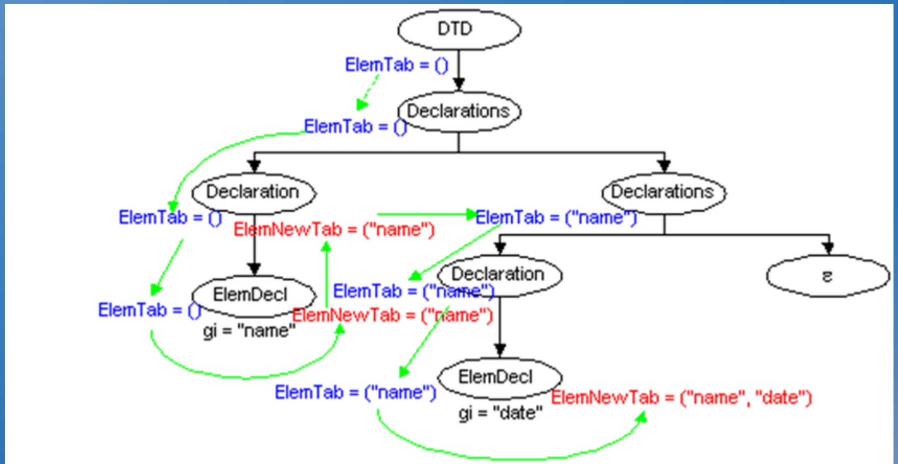
Execution and Sensitivity



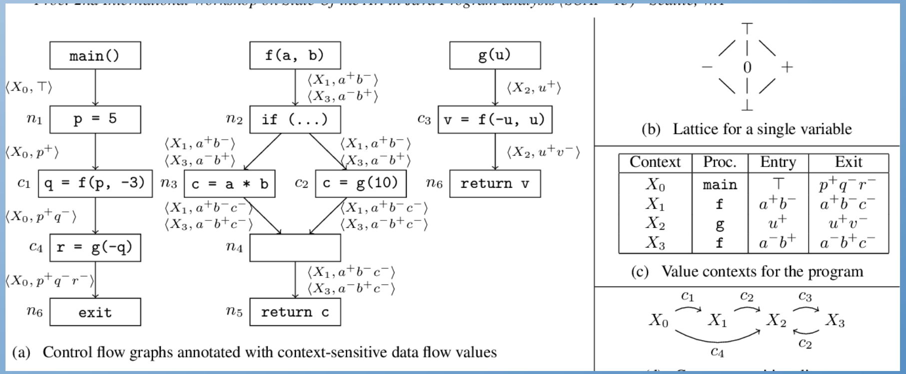
- How to interpret code depends on flow sensitivity
  - Multiple instances of a function -> context sensitive
    - Generally, a separate instance for each call site
  - Single instance of a function -> context insensitive
  - Consider flow inside a function -> flow sensitive
    - Otherwise, assume all statements are executed in any order
    - Flow insensitive is useful for some analyses, not in general
  - Multiple instances of a statement -> path sensitive
- Note that all of these keep the process finite
  - And hence all ensure the analysis will terminate

Program Points

- Abstract interpretation effectively executes the program
  - Looking at the effect of each instruction if assembler/byte code
  - Interpreting source code at the instruction level
    - This can be done using the abstract syntax tree
    - Think of an ordered AST **visitation** as program instructions
      - Enter a node
      - Return to a node after each individual child
    - Dotted AST node (Node plus location in the visitation)
- These are **program points**
  - Exact set determined by sensitivity
  - But the set is finite
- Associate values with each program point
  - Finite set of finite values



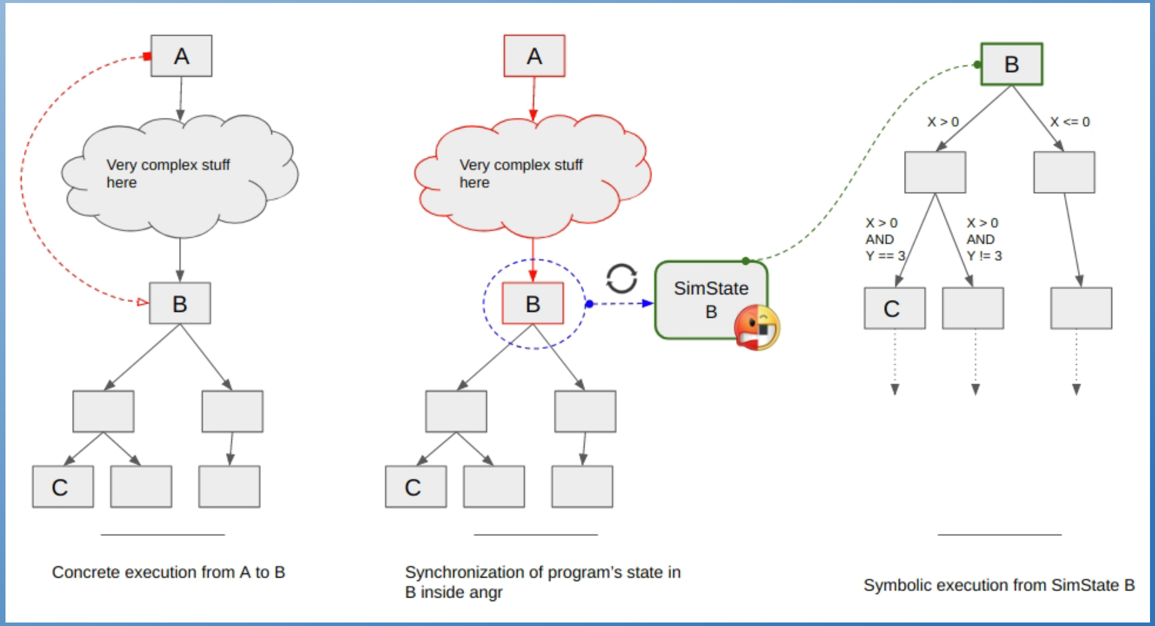
Values in Flow Analysis



- Typed Data: finite representations of data
  - With type and subtype information
  - For numbers, can indicate a particular value, value set, range, range set, ANY
    - Kept to a small number of items for each value
  - For strings, a particular value, value set, length range, ANY
  - For objects, keep a set of entities representing possible objects
    - Each creation site (new for example) has its own entity (finite number of these)
    - Generic entity for unknown sources
    - Each entity can have field values (which are typed data)
    - And array entities can have indexed values (which are typed data)
  - ANY values let flow analysis consider all possible inputs
- Each program point has a collection of possible values
  - Values accessible at that point (local variables, stack)
  - Global values are kept globally but can be accessed at any program point
  - Can keep track of fields that are set locally and then accessed (not necessarily accurate)
  - Overall precision of the flow analysis depends on value representations
- Need to define how values are changed by program operations
  - This is the nitty-gritty of abstract interpretation

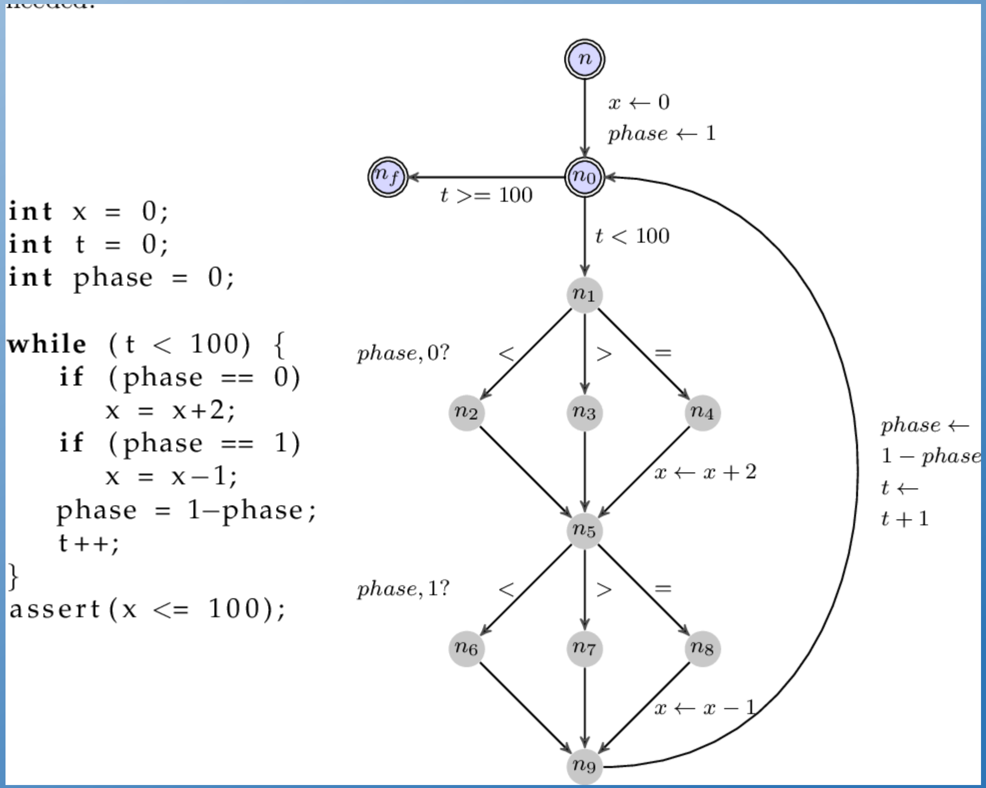
Values Stored for Program Points

- Start of a Method:
  - Value of parameters
  - Return value (merged)
  - Avoid reinterpretiation if no changes
- Program point:
  - Instruction, Dotted AST node
  - Contents of local variables, stack
  - (Contents of fields stored locally)
- Objects:
  - Contents of various fields
  - Contents of array elements (by index and globally)
- Global:
  - Contents of global (static) variables: single value

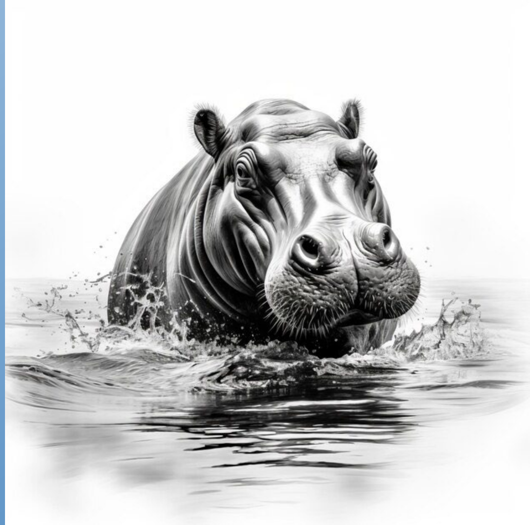


Operations on Values

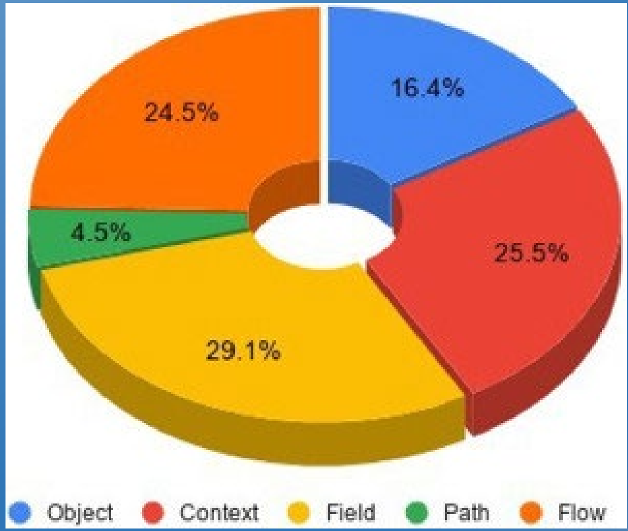
- Merging two values
  - This is the most common operation
  - Needed at all flow merge points
  - Needed for global & field assignments
  - Needed for merging call arguments
  - Needed for merging return values
- Based on operators in the language
  - Assignments
  - Integer operations
  - String operations
  - Stack operations
  - Accessors (field & index)
  - Casts (implicit and explicit)
- Individual methods might be special
  - Sanitize, MD5 hashing can change subtype states



Practical Abstract Interpretation

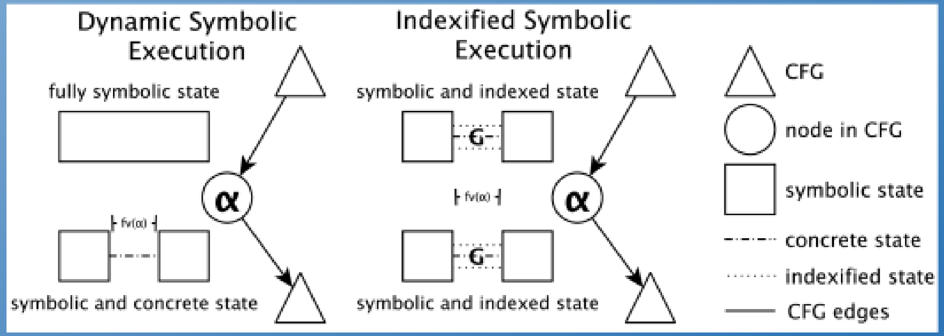


- Need an efficient representation of values
  - Tradeoff between precision and performance
  - Since representation needs to be small and finite
- Need an effective way of doing the interpretation
  - Knowing when to create a new instance of a routine (context sensitive)
  - Minimizing the reinterpretation of states
    - If merge doesn't change values, then no need to reinterpret
  - Tradeoff between precision and performance
- Need to handle the quirks of the language
  - Exceptions, callbacks from system methods
  - Reflection, native methods
- Need to make things efficient
  - To make this practical

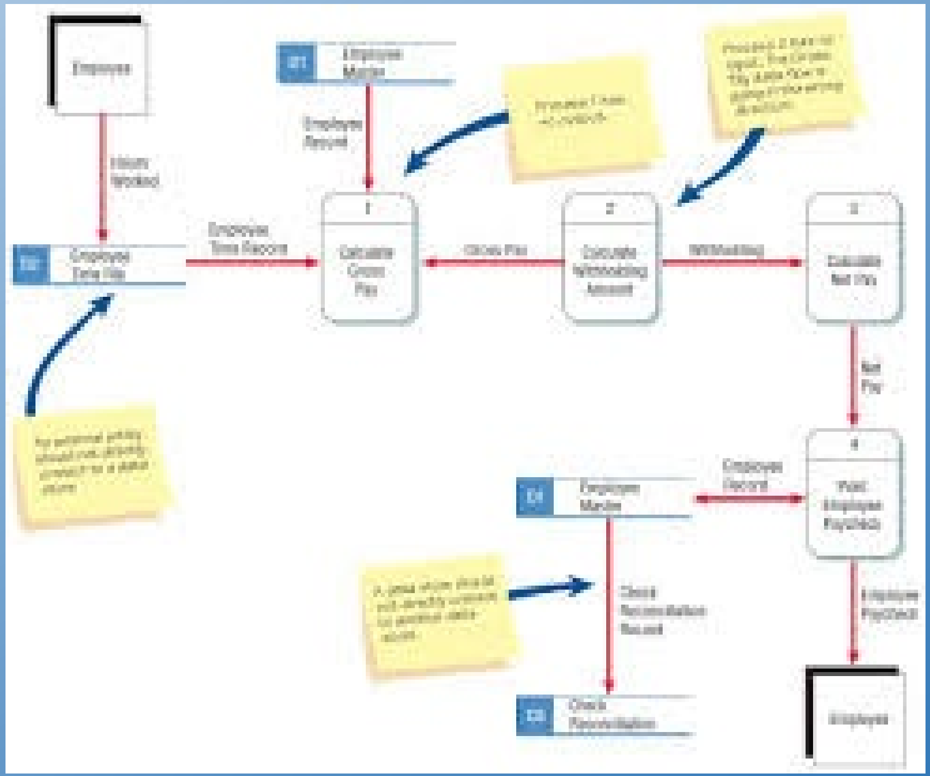


Efficient Abstract Interpretation

- Use a work queue algorithm
  - Start with program starting points (main, tests, …)
  - Create dummy code to execute tests, static initializers, …
- Given a program point with its values
  - Determine the set of next possible program points
  - And the possible values there
    - Stack, local variables
- Merge values as appropriate
  - From original values saved for that program point
    - Or just use the new values if the point was never executed
  - If anything changes or a new point, queue that program point
  - Try to queue alternatives in a logical order to minimize reevaluation
- Continue until nothing changes
  - Everything is finite; values only grow; this is guaranteed to terminate



Flow Analysis Errors

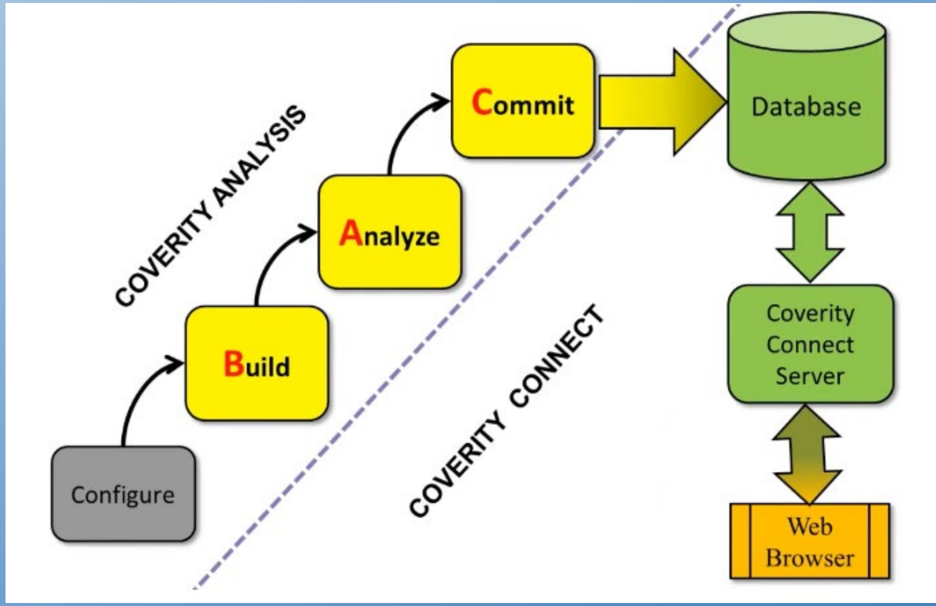


- Detected error indicates a possible program error
  - Conflict between expected and actual subtypes
    - Using annotations to define expected subtypes
  - Error program state reached
    - After some program state event-based transition
  - Can be wrong (like lint) since it is an approximation
    - The abstract execution might not be possible in the real program
  - Saved state information can provide program path leading to error
    - This is the counter-example we desire
    - It produces a graph of possible counter-examples
    - User can go through and determine which if any of these is really possible
- No errors found show the program is safe
  - Assuming the analysis is accurate
  - But false positives can be reasonably common (depends on property)
- Accuracy
  - Lack of accuracy can yield false negatives (safe when program isn’t)
  - One tries to avoid inaccuracy for safety conditions
  - But a little inaccuracy is probably okay for immediate feedback

Abstract Interpretation Difficulties

- This sounds straightforward, **but**:
  - Real programming languages and real programs are not that simple
  - Need to handle real programs and their complications
- Java: handling reflection, native methods, initializations, etc.
- Handling call backs from system routines
  - Swing/AWT event loop itself is hidden
- Handling exceptions (especially run time exceptions)
  - Thrown exceptions are relatively easy
  - Considering all potential run time exceptions is expensive & not very helpful
- Handling test cases (in addition to the main program)
  - With the various @Before ... annotations
- Handling implicit calls
  - Static initializers, boxing and unboxing, lambdas
  - Calls to Thread.start imply invoking Thread.run()
- Tracking program state & subtype changes
- Making it efficient enough to work as the user types

Flow Analysis Tools



- Quite a few exist
  - Mainly for fixed sets of properties
    - Optimized for those properties
  - Generally, these run in batch mode
  - And can take hours or days to run on something complex
- Examples
  - Amandroid for android checking
  - Commercial tools (C++): coverty, parasoft, understand, veracode, …
  - Often run with major check-ins, before releases
- Our Goal: Immediate feedback on safety errors

Immediate Feedback

- A lot of things are still done in batch
  - Compilation
  - Testing
  - Checker type checking
  - Checkstyle
  - Automatic bug repair
- I believe they would all be more effective if done immediately
  - Provide feedback as the developer types
  - Compiler feedback in todays IDEs shows this
- My research goal continues to be to find ways to achieve this
  - Continuous testing: not that useful since tests take too long and interact
  - ROSE for automatic bug repair while you debug
  - Code Bubbles checkstyle plugin to check as you edit
  - And abstract interpretation for checking safety properties as you edit
    - Fast data flow analysis has other applications as well

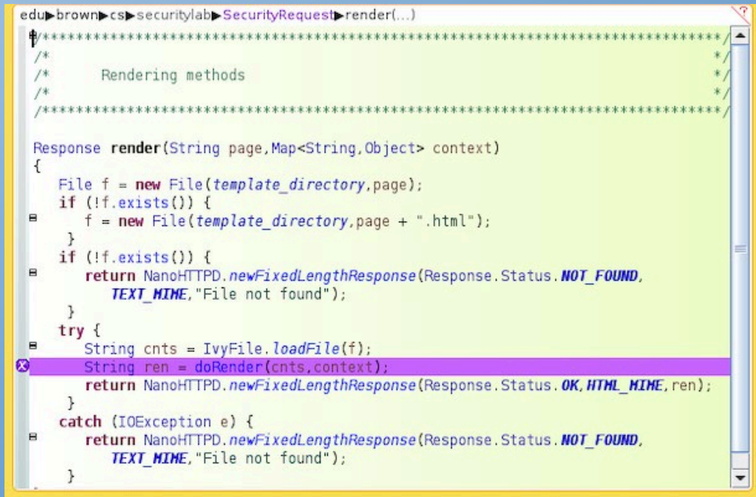
Safety Checking as you Program

- Goal: Show the state of safety conditions in real time
  - As the programmer writes or edits the code
  - Show problems to the programmer in the IDE
- This is the goal of our FAIT project
  - Runs in conjunction with Code Bubbles
  - Provides error indications at safety violations
  - Provides information on why this is a safety violation
    - Graph of paths leading to the violation
  - Provides other information
    - Backwards slice of a variable
  - Useful for other tools
    - ROSE uses it to do fault localization

FAIT: Efficient Flow Analysis in an IDE

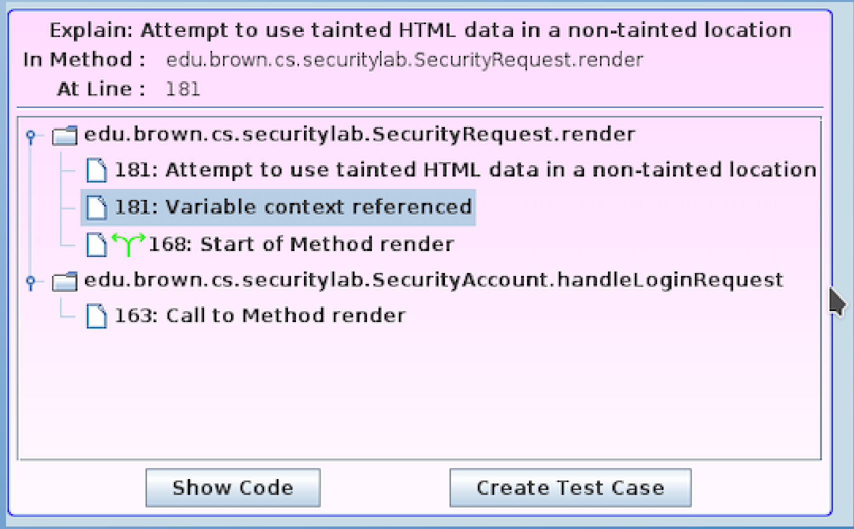
- Making it efficient and complete
  - Handle all the Java complexities such as reflection with some user input (via a resource file)
  - Special handling collections, maps, string buffers
  - Ignoring methods that do not matter with respect to the condition(s)
    - Based on user input in a resource file
  - Using unique immutable objects for efficient entity and value representations
- Making it concurrent
  - For much faster flow analysis
  - Work queue of methods to work on; work queue of locations in the method
  - Separate threads can work on separate methods simultaneously with little synchronization
- Making it incremental
  - To update as the user types
  - Detect what might have changed, mark those as invalid, add to work queues
- Handling both source and binary files simultaneously – parallel interpreters
  - Source for files being edited (needed to avoid continual save and compile)
  - Binary for everything else
- Tries to be both accurate and precise while being efficient
  - User input on what is important; what values to consider; what should be context sensitive

Code Bubbles FAIT Analyzer



- Runs as part of the environment
  - Updates anytime there are no compiler errors
    - Can't execute when there are errors
  - Provides immediate feedback
    - Typically, in under a second, with initial analysis in under a minute
- Requires the user to define subtypes and safety conditions
  - What they want to have checked
    - This varies from one system to another
    - Some basic ones predefined (Tainted data for SQL injection & XSS, for example)
  - Subtypes
    - Various states, rules for computation; again, some basic ones predefined
  - Safety conditions
    - Various states, program events, rules for changing state based on event

FAIT in Code Bubbles



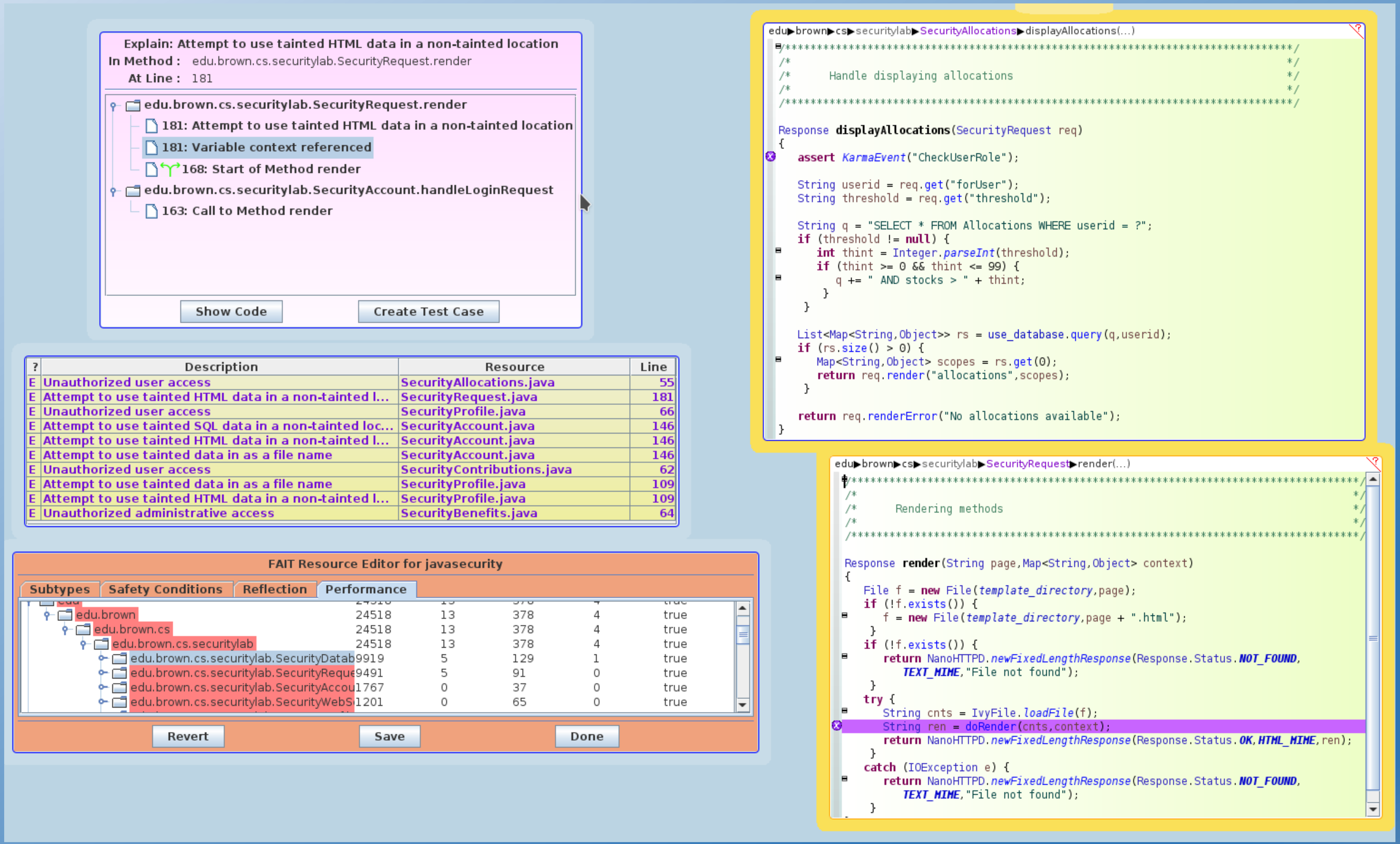
- Requires some annotation of the code
  - Using Java type annotations
    - Either direct or indirect (in resource file)
  - Using preannotated library routines (from resource file)
  - Source and sink annotations required for type checking
    - But not the intermediate values as Checker would require
    - Standard ones (e.g., Spring html server, Java html server) are predefined
    - Or can call dummy function (e.g., KarmaUtils.taint(data))
  - Calls to dummy function indicate state-based events
    - KarmaUtils.event("event")
- Requires additional information from a resource file
  - How to handle reflection, native methods
    - Reflection only in cases where the class is never instantiated otherwise
    - Most standard library native methods are taken care of
  - Note special case methods (e.g., those that never return)
  - What methods can be ignored (for better performance)
    - Or whole classes or whole packages
- Current Research on user interface to define and edit the resource file
  - This is a difficult user interface problem

FAIT in Code Bubbles

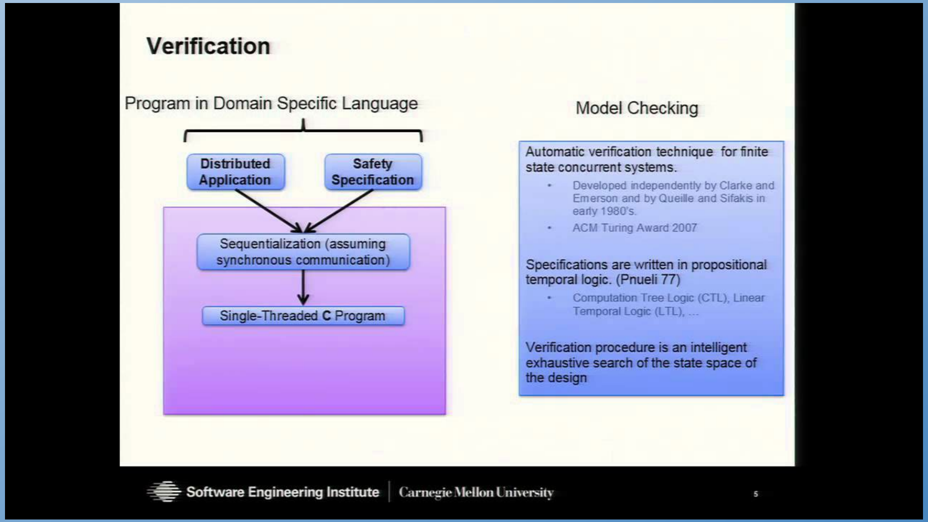
?	Description	Resource	Line
E	Unauthorized user access	SecurityAllocations.java	55
E	Attempt to use tainted HTML data in a non-tainted l...	SecurityRequest.java	181
E	Unauthorized user access	SecurityProfile.java	66
E	Attempt to use tainted SQL data in a non-tainted loc...	SecurityAccount.java	146
E	Attempt to use tainted HTML data in a non-tainted l...	SecurityAccount.java	146
E	Attempt to use tainted data in as a file name	SecurityAccount.java	146
E	Unauthorized user access	SecurityContributions.java	62
E	Attempt to use tainted data in as a file name	SecurityProfile.java	109
E	Attempt to use tainted HTML data in a non-tainted l...	SecurityProfile.java	109
E	Unauthorized administrative access	SecurityBenefits.java	64

- Reasonably precise for most safety conditions
  - But this varies with complexity of the code
  - And how complex one allows values to become (ranges and constraints)
  - Doesn’t detect multithreading errors (e.g., deadlocks and race conditions)
    - Doesn’t consider locks in general
  - Graphs let user explore output; resource files let user adjust program to achieve more precision
- Mostly accurate, but accuracy not guaranteed
  - Performance, reflection annotations, native code
    - User descriptions of these in resource files might introduce errors
  - Doesn’t consider all possible run time exceptions
- Fast enough to be useful
  - Updates whenever code is error free
  - Presents current errors to user
  - Can be queried in real time
    - Provide a back slice of how a value might be set from the editor
    - Provide information to ROSE on possible locations leading to a fault
    - Provide a trace of how an error might have occurred

FAIT in Code Bubbles

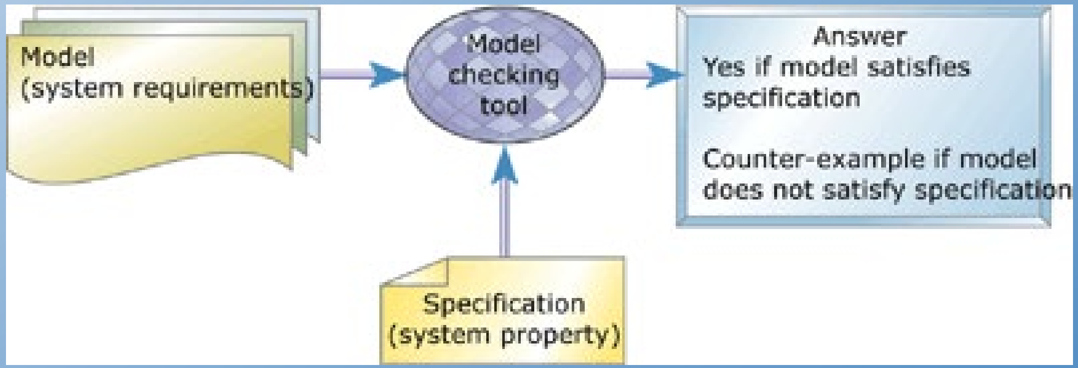


Model Checking



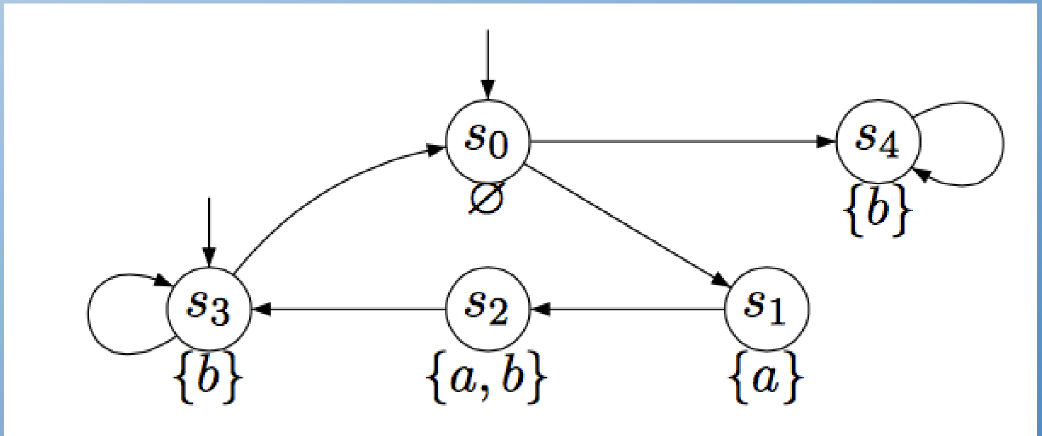
- Abstract interpretation is limited
  - Doesn’t consider interleaving executions
  - Doesn’t consider external events
  - Doesn’t consider locking
  - Doesn’t consider the external world
- Model Checking is a more general approach
  - Designed to handle interactions with the external world
  - Designed to handle thread-based interleaving
  - Can handle locking
  - Used extensively for embedded systems

Model Checking



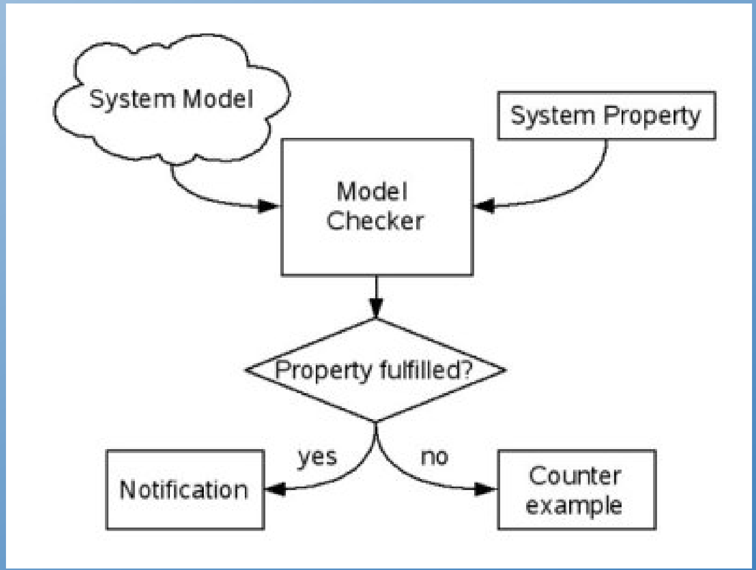
- Originally created for checking hardware
  - Small number of internal states
  - Hardware is inherently finite
- Then used for checking embedded systems
  - Embedded code is usually relatively simple
    - And usually modeled or written as a finite state automata
  - Interactions with the real world are important
  - These are often safety-critical (e.g., pinball fuses; train crashes)
- Then use to check arbitrary software systems
  - Checking individual safety conditions in a program

Model Checking



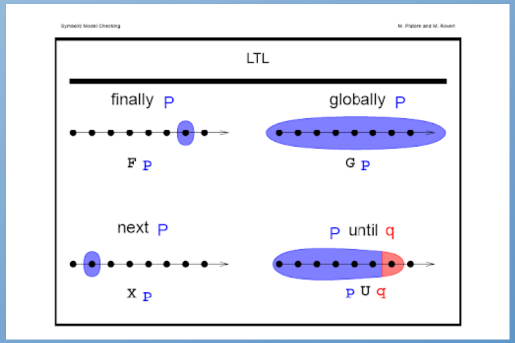
- Can be done in terms of automata
  - One automata for program, one for safety condition
  - Find all possible pairs of states
    - Run the two automata in step (cross-product)
    - Events from program automata drive condition automata
    - Outside events can also change the condition automata
  - This is effectively what is done by abstract interpretation state checking
- Usually done in terms of temporal logic formulas
  - Boolean formulas
  - With additional operators for time (different sets are used)
    - X is true forever (X always occurs)
    - X is true in the next state
    - X is true in some future state (X eventually occurs)
    - X is true in some state after Y is true

Model Checking



- Create a single formula that is true if the program fails
  - Represent a program state s as a Boolean vector
    - Finite number of variables, each with a finite representation
    - Includes the program counter (e.g., the program point)
    - Assume a finite number of threads – each thread has its own state
      - Program state is the concatenation of the thread states
  - Define the starting state of the program s_0
  - Create a logical relation representing program execution
    - R(s1,s2) is true if program can transition from state s1 to state s2
    - Basically, R is the OR of the effect of each instruction on the program state
  - Create a temporal logic formula representing the condition C(s1)
    - Over the program state – map from the FSA if needed
  - Create a formula that says an execution from s_0 leads a state where C holds
    - Temporal logic (repeatedly use program relation)
  - This is large, but can be created mechanically
- Prove that formula holds
  - The proof provides a counter example and hence a potentially buggy execution
  - Find a set of values that satisfies this formula
  - Convert that into the appropriate execution

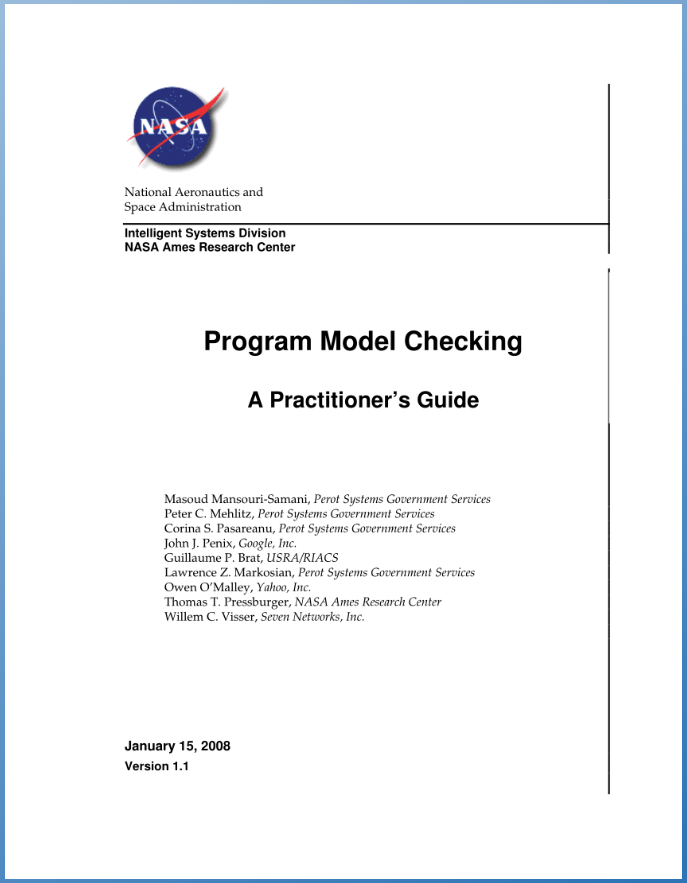
Model Checking



- Various technique exist to do the checking efficiently
  - Ordered binary decision diagrams
    - Cute data structure that can greatly simplify the search
    - Can handle program states space of size 2^100 or so
  - Newer technologies for 3SAT and similar problems
    - Can generally handle these as well
  - These are all batch processes however
- Both can yield a proof that shows how the program can fail
  - Which can be translated into a counterexample
  - Which can be translated back to the actual program
  - To provide feedback to the user

Practical Model Checking

- The whole process can be semi-automated
- JavaPathFinder is used by NASA for example
  - Starts with annotated Java code
  - And conditions to check
- Either says the model is correct
  - Or shows an execution leading to an error
    - Counterexample
  - If the reported execution isn’t possible
    - The model can be made more robust
    - Adding additional annotations or constraints
    - Extending the way values or the program is modeled
  - Often, a proof involves multiple such changes


National Aeronautics and
Space Administration

Intelligent Systems Division
NASA Ames Research Center

Program Model Checking

A Practitioner’s Guide

Masoud Mansouri-Samani, *Perot Systems Government Services*
Peter C. Mehlitz, *Perot Systems Government Services*
Corina S. Pasareanu, *Perot Systems Government Services*
John J. Penix, *Google, Inc.*
Guillaume P. Brat, *USRA/RIACS*
Lawrence Z. Markosian, *Perot Systems Government Services*
Owen O’Malley, *Yahoo, Inc.*
Thomas T. Pressburger, *NASA Ames Research Center*
Willem C. Visser, *Seven Networks, Inc.*

January 15, 2008
Version 1.1

PROJECT

- You should have a working version of your project
  - Not fully functional
  - But demonstratable
  - Something you can feel good about
- Choose a date for project presentations
  - One of 12/3, 12/5, 12/10 (2 presentations a day)
  - Or I will choose one for you
  - eMail me your options and priorities and I will assign dates