Lightweight Formal Methods: The What, Why, and How

John Baugh

Civil, Construction, and Environmental Engineering
North Carolina State University, Raleigh, NC
jwb@ncsu.edu

Workshop on Correctness and Reproducibility for Climate and Weather Software, NCAR, November 9-10, 2023

North Carolina State University

Department of Civil, Construction, and Environmental Engineering (ccee.ncsu.edu)

Enrollments: about 800 undergraduate and 300 graduate students

Regular course: *CE 537 Computer Methods and Applications* (35 times)

Coping with Software Complexity

Historically useful strategies

- **Abstraction:** generalizing concrete details, i.e., preserving the information that's relevant in a given context.
- **Separation of concerns:** finding parts of a problem that can be solved separately.
- **Engineering tools:** devising analysis and evaluation models, common design templates.
- **Progressive codification:** identifying, organizing, and systematizing useful patterns.

Building Models of Software

Model - a simplified representation of reality used to provide insight.

Building Models of Software

Model - a simplified representation of reality used to provide insight.

How is modeling useful?

> “It’s the process of organizing knowledge about a given system.”
>
> – B. Zeigler

Building Models of Software

Model - a simplified representation of reality used to provide insight.

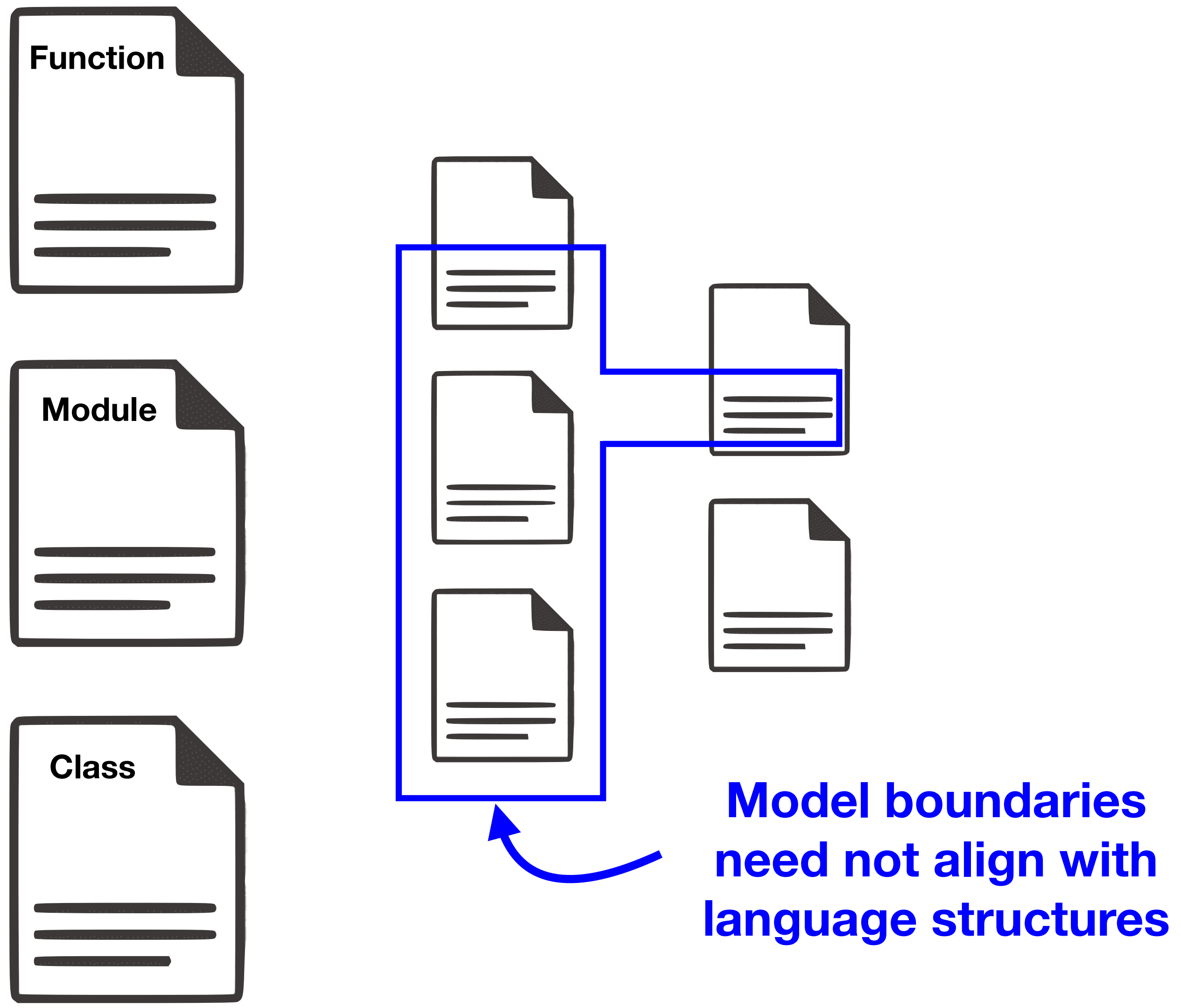
How is modeling useful?

> “It’s the process of organizing knowledge about a given system.”
>
> – B. Zeigler

What are the system boundaries?

> “A system is what is distinguished as a system.”
>
> – B. Gaines

What are the boundaries?



What Every Computer Scientist Should Know About Floating-Point Arithmetic

DAVID GOLDBERG

Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304

Floating-point arithmetic is considered an esotoric subject by many people. This is rather surprising, because floating-point is ubiquitous in computer systems: Almost every language has a floating-point datatype; computers from PCs to supercomputers have floating-point accelerators; most compilers will be called upon to compile floating-point algorithms from time to time; and virtually every operating system must respond to floating-point exceptions such as overflow This paper presents a tutorial on the aspects of floating-point that have a direct impact on designers of computer systems. It begins with background on floating-point representation and rounding error, continues with a discussion of the IEEE floating-point standard, and concludes with examples of how computer system builders can better support floating point.

Categories and Subject Descriptors: (Primary) C.0 [**Computer Systems Organization**]: General—*instruction set design*; D.3.4 [**Programming Languages**]: Processors—*compilers*, *optimization*; G.1.0 [**Numerical Analysis**]: General—*computer arithmetic*, *error analysis*, *numerical algorithms* (Secondary) D.2.1 [Software Engineering]: Requirements/Specifications—*languages*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics* D.4.1 [**Operating Systems**]: Process Management—*synchronization*

About This Talk

Lightweight formal methods

1. What are they?
    - A path to formal methods and a lightweight tool called Alloy

About This Talk

Lightweight formal methods

1. What are they?
    - A path to formal methods and a lightweight tool called Alloy
2. Why are they useful? Some rationale and examples:
    - An extension to a storm surge code used in production
    - Verifying ELLPACK and CSR sparse matrix operations
    - Refinement checking a Laplace solver in Coarray Fortran

About This Talk

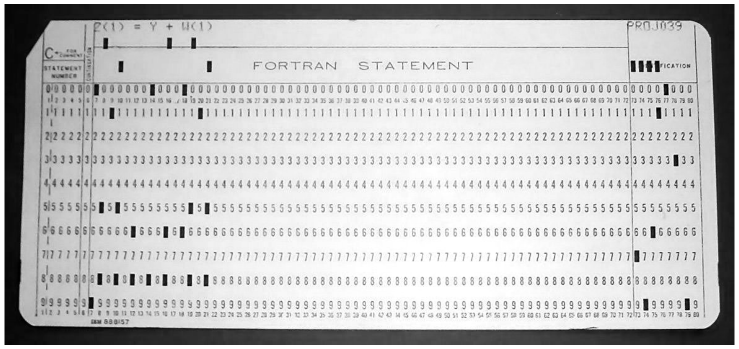
Lightweight formal methods

1. What are they?
    - A path to formal methods and a lightweight tool called Alloy
2. Why are they useful? Some rationale and examples:
    - An extension to a storm surge code used in production
    - Verifying ELLPACK and CSR sparse matrix operations
    - Refinement checking a Laplace solver in Coarray Fortran
3. How can we employ them?
    - Alloy, heap invariants, and pipe network analysis

1. The ***What***

As an Undergraduate

"A second course in Fortran programming?"

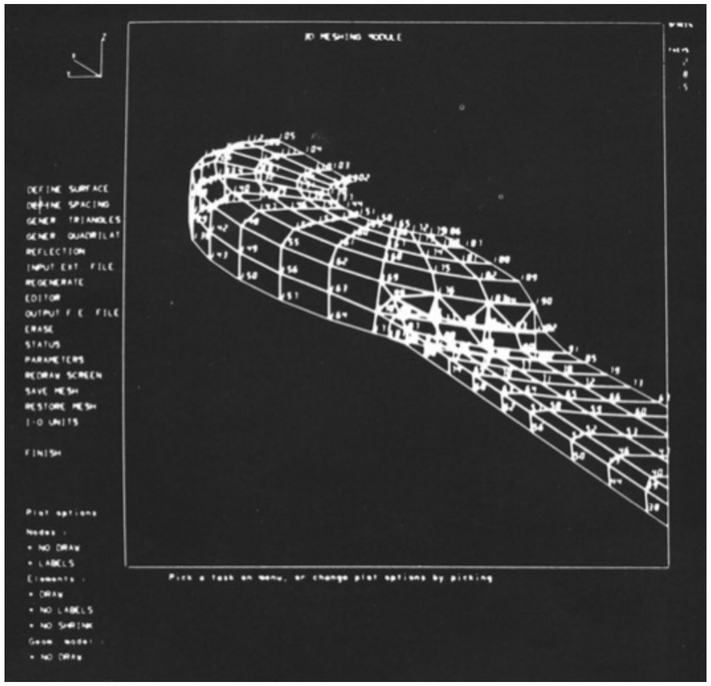


Hollerith Card
(one line of code per card)

IBM Keypunch Machine
(used into the early '80s)

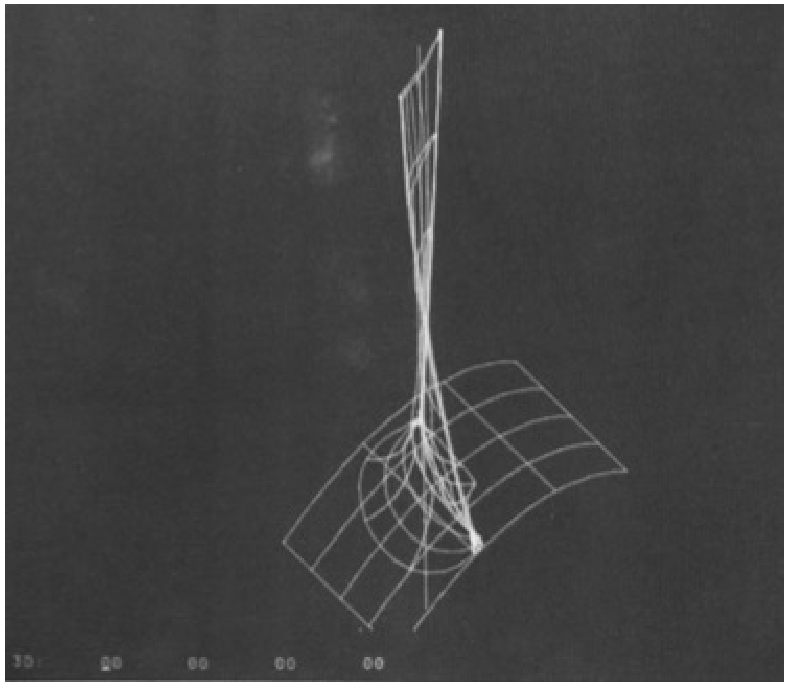
At a National Lab

"Add a new spline type for surface patches."



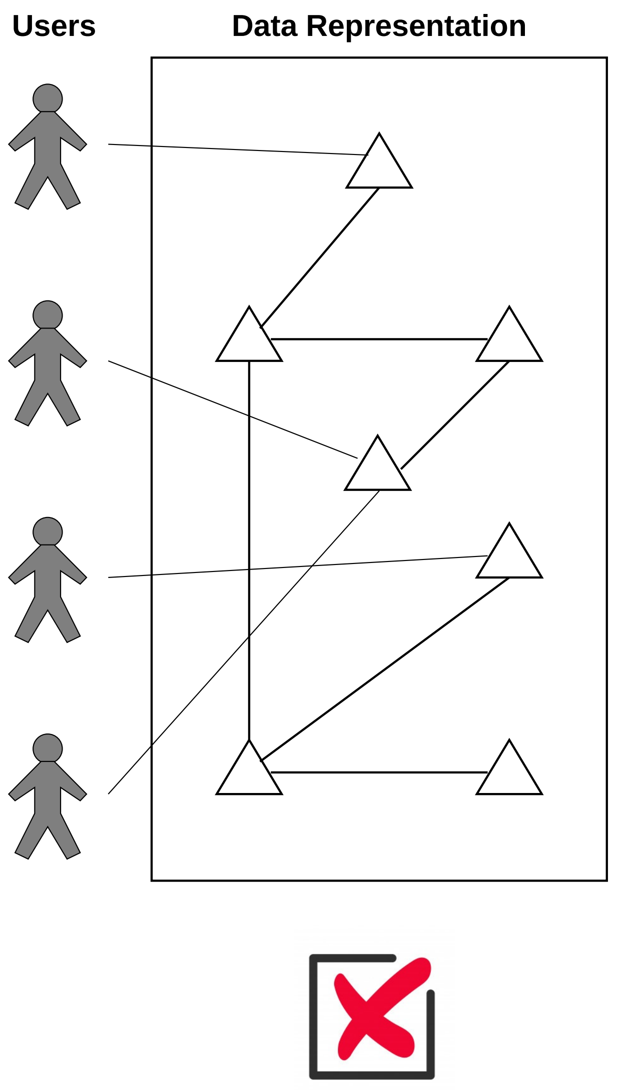
Pick a task on menu, or change plot options by picking

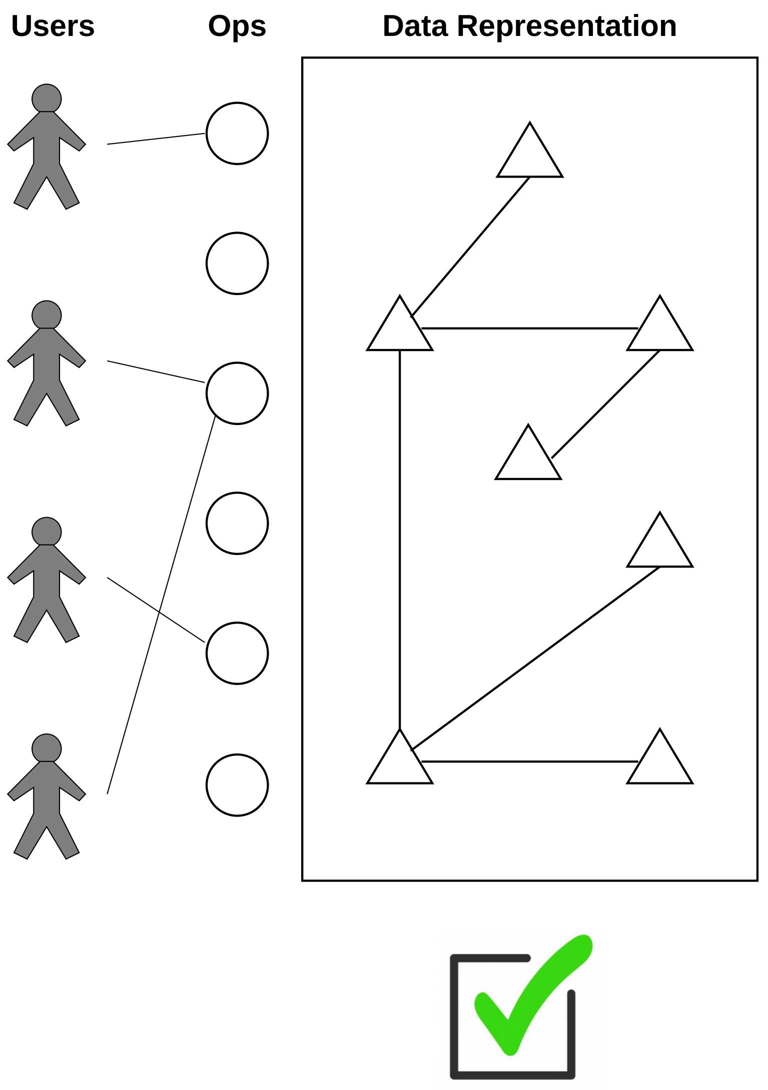
Meshed Surface Construction



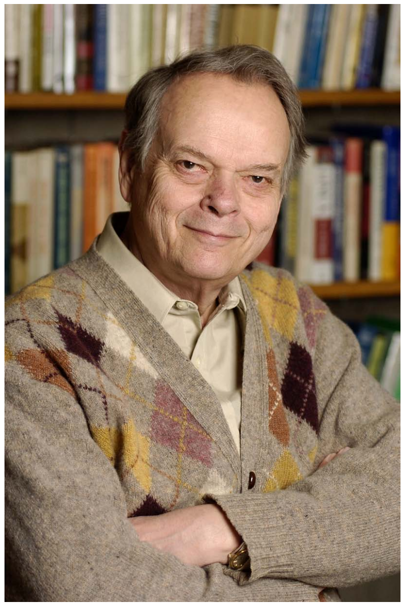
Turbine Blade Geometry

Data Abstraction?





Program Proving?



John Reynolds

Carnegie Mellon University

Programming language semantics, separation logic

Syllabus from his 1980s-era course on program proving:

The fundamental goal is the ability to write concise, clearly documented, and logically correct programs. Students will write short programs **without executing them on the computer**, and these exercises will be evaluated for style as well as logical correctness.

Floyd-Hoare Logic

Hoare's notation:

$$\{P\}\ S\ \{Q\}$$

where:

- S is a program statement (possibly a compound one)
- P and Q are the *precondition* and *postcondition*, respectively

Floyd-Hoare Logic

Hoare's notation:

$$\{P\}\ S\ \{Q\}$$

where:

- S is a program statement (possibly a compound one)
- P and Q are the *precondition* and *postcondition*, respectively

Interpretation:

If one executes S beginning with any state described by P, and if S terminates, then S will produce a state described by Q.

Floyd-Hoare Logic

Hoare's notation:

$$\{P\}\ S\ \{Q\}$$

where:

- S is a program statement (possibly a compound one)
- P and Q are the *precondition* and *postcondition*, respectively

Interpretation:

If one executes S beginning with any state described by P, and if S terminates, then S will produce a state described by Q.

Partial correctness:

A safety property. Termination is a separate argument.

Simple Iterative Programs

Problem: Consider the program below for performing integer multiplication by repeated addition:

```
$\{y \geq 0\}$                    $\leftarrow$ *Precondition*
let z = 0, k = 0
    {whileinv $I$: $z = x \times k$ and $k \leq y$}        $\leftarrow$ *Loop invariant*
    while $k \neq y$
        k = k + 1; z = z + x
    end
end
$\{z = x \times y\}$                $\leftarrow$ *Postcondition*
                                  (since $z = x \times k$ and $k = y$)
```

Iterative Loops: A Checklist

$\{P\}$

$\{$**whileinv** I: the loop invariant$\}$

while B S **end**

$\{Q\}$

1. *Achieve I*

   Show that the loop invariant I is true before the loop begins

2. $\{I \wedge B\}\ S\ \{I\}$

   Show that the loop body, S, maintains the loop invariant

3. $I \wedge \neg B \Rightarrow Q$

   Show that the desired result is true upon exiting the loop

4. For total correctness, of course, we must also show termination

Mechanical Verification

Generate verification conditions in Julia and discharge them using Z3, an SMT-based theorem prover (satisfiability modulo theories):

```
julia> vc(P, S, Q)
4-element VectorZ3.ExprAllocated:
```

P

```
  (=> (>= y 0)
      (and (= 0 0) (= 0 0) (>= y 0)))
```

Achieve I

```
  (=> (and (= z 0) (= k 0) (>= y 0))
      (and (= z (* x k)) (<= k y)))
```

$\{I \wedge B\}\ \ S\ \ \{I\}$

```
  (=> (and (= z (* x k)) (<= k y) (distinct k y))
      (and (= (+ z x) (* x (+ k 1))) (<= (+ k 1) y)))
```

$I \wedge \neg B \Rightarrow Q$

```
  (=> (and (= z (* x k)) (<= k y) (not (distinct k y)))
      (= z (* x y)))
```

Perspective

Formal methods are studied, not as an end in themselves, but to reveal

- what constitutes a precise specification of program behavior.
- what constitutes a rigorous argument that a program meets such a specification.

Perspective

Formal methods are studied, not as an end in themselves, but to reveal

- what constitutes a precise specification of program behavior.
- what constitutes a rigorous argument that a program meets such a specification.

"How do we connect the dots in an argument
that a program does what it purports to do?"

What we’re talking about is *static analysis*

General idea

Try and compute approximate but sound guarantees about the behavior of a program without executing it.

Tools

Various approaches analogous to the traditional divide in logic between *proof theory* and *model theory*.

What we’re talking about is *static analysis*

General idea

Try and compute approximate but sound guarantees about the behavior of a program without executing it.

Tools

Various approaches analogous to the traditional divide in logic between *proof theory* and *model theory*.

- Theorem provers: deductive, axioms and inference rules
- Model checkers: finite state machines and temporal logic
- Model finders: find an instance of a logical formula

What we’re talking about is *static analysis*

General idea

Try and compute approximate but sound guarantees about the behavior of a program without executing it.

Tools

Various approaches analogous to the traditional divide in logic between *proof theory* and *model theory*.

- Theorem provers: deductive, axioms and inference rules
  - Z3, CVC5, PVS, ACL2, Coq, Isabelle, Agda, Lean
- Model checkers: finite state machines and temporal logic
  - NuSMV, SPIN, TLA+, BLAST, FDR4, LTSA, UPPAAL
- Model finders: find an instance of a logical formula
  - Alloy, Alloy*, αRby, ProB

How should we employ tools?

Disappearing Formal Methods, Rushby, SRI

- formal machinery should “disappear” into familiar environments

How should we employ tools?

Disappearing Formal Methods, Rushby, SRI

- formal machinery should “disappear” into familiar environments

Lightweight Formal Methods, Jackson, MIT, and Wing, Columbia

- direct use in modeling and analysis
- emphasis on ease of use and focused application
- **intended to influence design**

How should we employ tools?

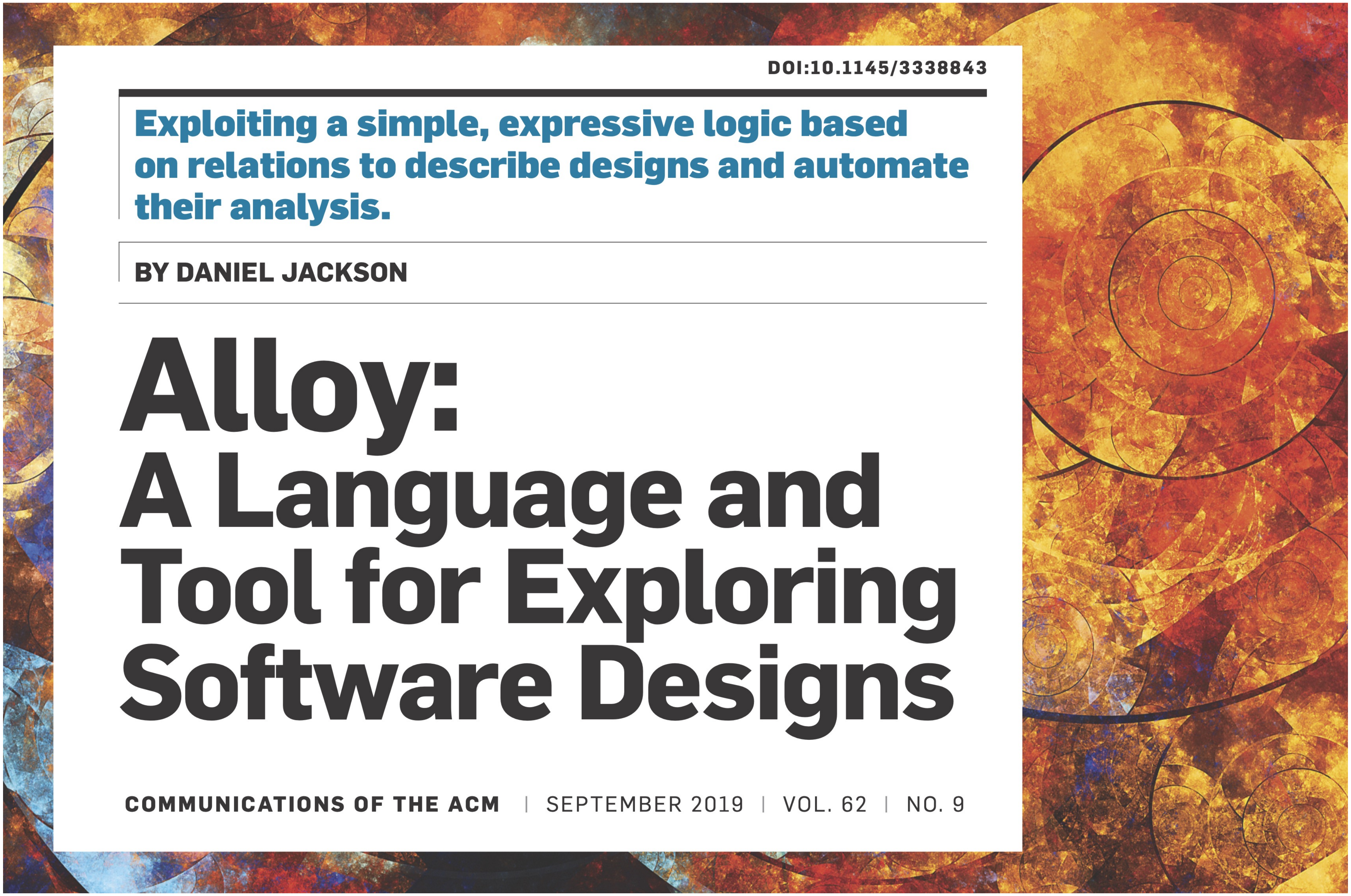
Disappearing Formal Methods, Rushby, SRI

- formal machinery should “disappear” into familiar environments

Lightweight Formal Methods, Jackson, MIT, and Wing, Columbia

- direct use in modeling and analysis
- emphasis on ease of use and focused application
- **intended to influence design**

“Code is a poor medium for exploring abstractions.” – Jackson



DOI:10.1145/3338843

Exploiting a simple, expressive logic based on relations to describe designs and automate their analysis.

BY DANIEL JACKSON

Alloy: A Language and Tool for Exploring Software Designs

COMMUNICATIONS OF THE ACM | SEPTEMBER 2019 | VOL. 62 | NO. 9

https://cacm.acm.org/magazines/2019/9/238969-alloy/

Lightweight Modeling in Alloy

A declarative modeling language

- First-order logic, relational calculus, and transitive closure
- Inspired by Z and model checkers like SMV

Lightweight Modeling in Alloy

A declarative modeling language

- First-order logic, relational calculus, and transitive closure
- Inspired by Z and model checkers like SMV

Automatic, push-button analysis

- Supports iterative development and analysis

Lightweight Modeling in Alloy

A declarative modeling language

- First-order logic, relational calculus, and transitive closure
- Inspired by Z and model checkers like SMV

Automatic, push-button analysis

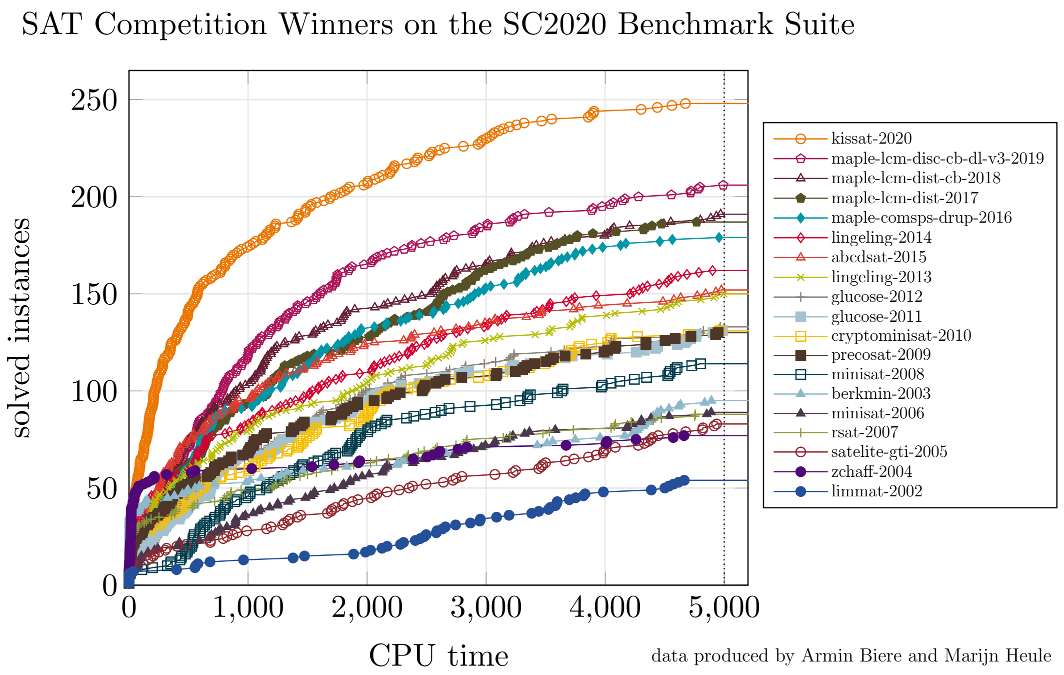
- Supports iterative development and analysis

Compilation to SAT

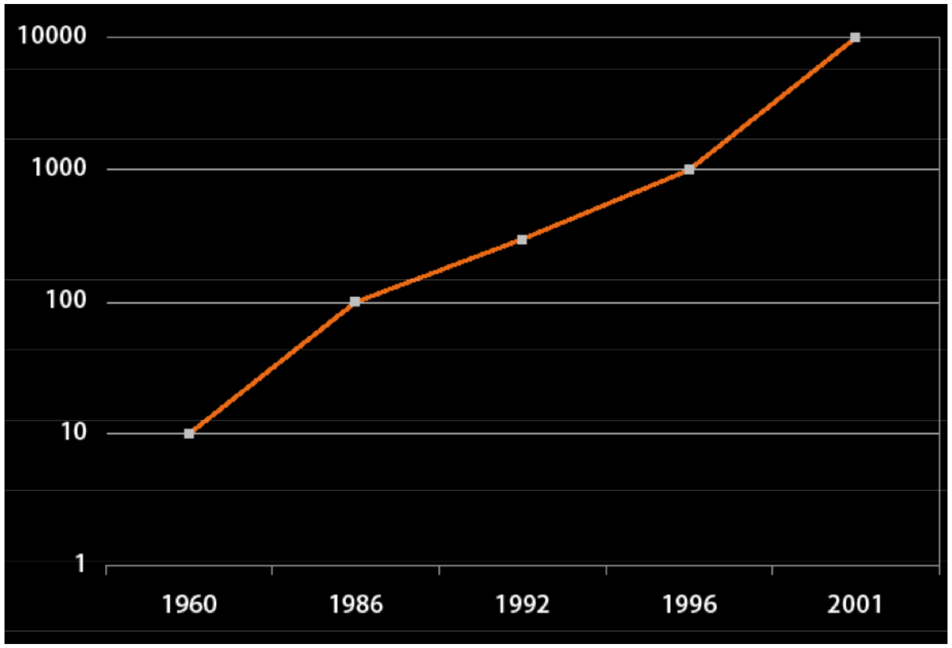
- Boolean satisfiability problem: $(x \vee y) \wedge (x \vee \neg y)$ is satisfiable when x is true

SAT Solver Performance

Annual Competitions (2002–20)



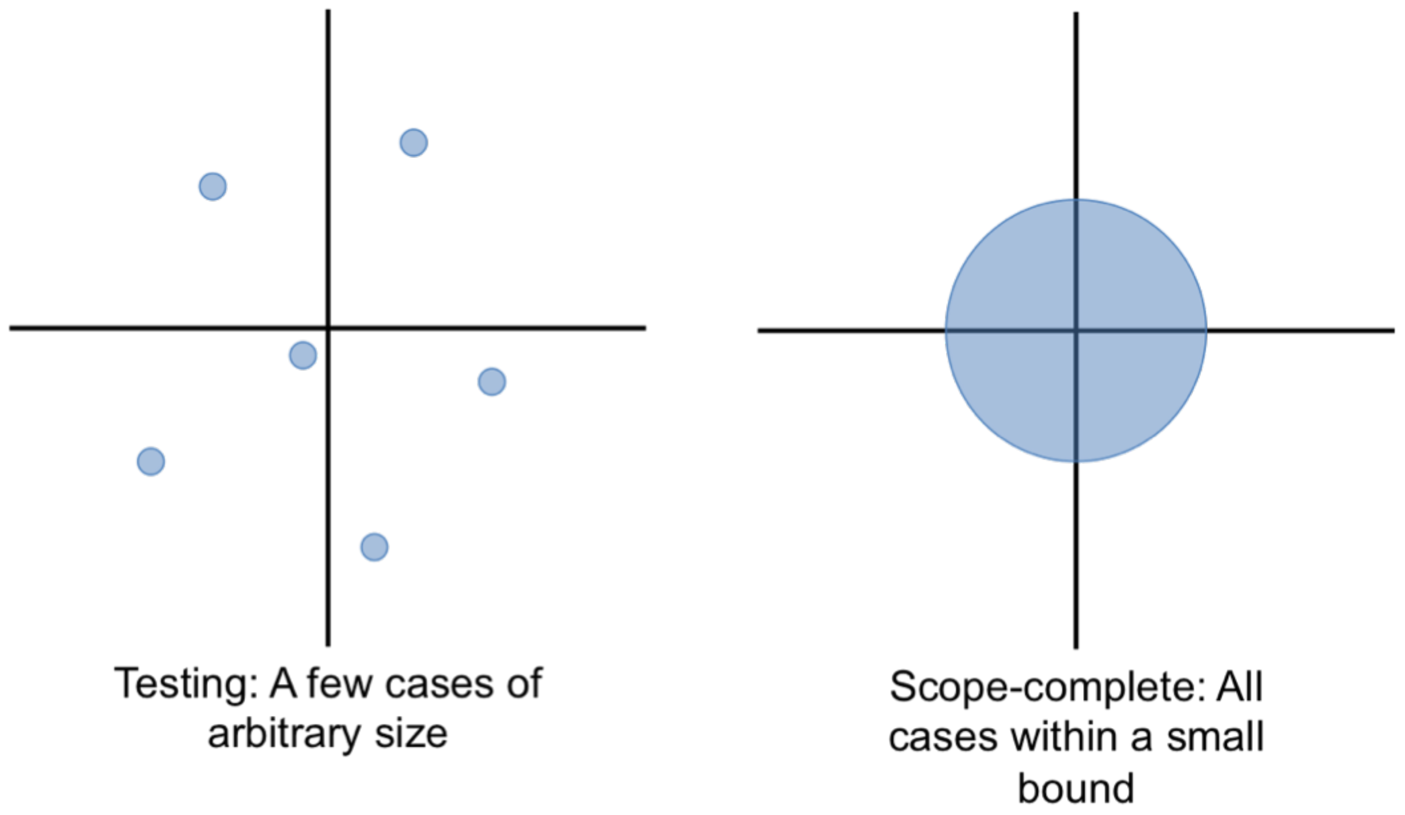
Historically



Greg Dennis and Rob Seater, *Alloy Tutorial*, alloytools.org

Small Scope Hypothesis

A high proportion of bugs can be found by testing a component on all inputs within a small scope.


Testing: A few cases of arbitrary size
Scope-complete: All cases within a small bound

Alloy Case Studies

With thousands of applications in categories like:

Enterprise modeling, auctions, electronic commerce, security, access control and security policies, feature modeling and analysis, domain specific languages and modeling, train control, file systems, software architecture, refactoring, program verification, databases, model-driven development, network protocols, testing and automated test case generation, configuration and reconfiguration, data structure repair, requirements, and teaching

https://alloytools.org/citations/case-studies.html

Alloy Case Studies

With thousands of applications in categories like:

Enterprise modeling, auctions, electronic commerce, security, access control and security policies, feature modeling and analysis, domain specific languages and modeling, train control, file systems, software architecture, refactoring, program verification, databases, model-driven development, network protocols, testing and automated test case generation, configuration and reconfiguration, data structure repair, requirements, and teaching

https://alloytools.org/citations/case-studies.html

And success in finding 1) safety-critical flaws in a **neutron radiotherapy** installation, 2) bugs in **Chord**, a prominent peer-to-peer distributed protocol, and 3) vulnerabilities in **WebAuth**, for Kerberos authentication, and so on.

Discovered using Alloy by a team at U. Washington,[1] a researcher at AT&T,[2] and a research group at UC Berkeley and Stanford.[3]

2. The ***Why***

About Scientific Software

Challenges

- Meeting quality and reproducibility standards, productivity [Wilson, 2006; Faulk et al., 2009]
- Numerous empirical studies of software “thwarting attempts at repetition or reproduction of scientific results” [Storer, 2017]
- Subsequent retractions of papers in scientific journals

About Scientific Software

Challenges

- Meeting quality and reproducibility standards, productivity [Wilson, 2006; Faulk et al., 2009]
- Numerous empirical studies of software "thwarting attempts at repetition or reproduction of scientific results" [Storer, 2017]
- Subsequent retractions of papers in scientific journals

Domain characteristics

- Software developed by domain experts
- Lack of test oracles: novel findings, difficult to validate
- Focus on performance and hardware utilization

Paths to Improvement

Broad categories of suggested approaches [Storer, 2017]

- development processes, e.g., agile methods
- quality assurance practices including testing, inspections, and continuous integration
- design approaches such as component architectures and design patterns

Paths to Improvement

Broad categories of suggested approaches [Storer, 2017]

- development processes, e.g., agile methods
- quality assurance practices including testing, inspections, and continuous integration
- design approaches such as component architectures and design patterns

Among quality assurance practices, **formal methods** are included

- . . . with the caveat that they have received considerably less attention in the scientific programming community, possibly due to “the additional challenge of verifying programs that manage floating point data”

Is there a role for tools like Alloy?

The essence of scientific software:

- Structure
  - Rich state in the form of spatial, geometric, material, topological, and other attributes
- Behavior
  - Explicit parallelism in a variety of forms
  - Continuous processes encoded as finite systems

State-based formal methods in scientific computation. Baugh and Dyer. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z: 6th International Conference, ABZ 2018,* pages 392–396, Cham, 2018.

Is there a role for tools like Alloy?

The essence of scientific software:

- Structure
  - Rich state in the form of spatial, geometric, material, topological, and other attributes
- Behavior
  - Explicit parallelism in a variety of forms
  - Continuous processes encoded as finite systems

In principle, such characteristics are a match for state-based formalisms like Alloy.

But what about the reals?

State-based formal methods in scientific computation. Baugh and Dyer. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z: 6th International Conference, ABZ 2018,* pages 392–396, Cham, 2018.

Separating Concerns

We naturally think of continuous processes:

e.g., circulation of ocean currents

But what does the computational apparatus underlying ocean circulation models really look like?

- purely analytic functions? ✗
- an amalgam of data structures, algorithms, and . . .
  numerical expressions ✓

Separating Concerns

We naturally think of continuous processes:

e.g., circulation of ocean currents

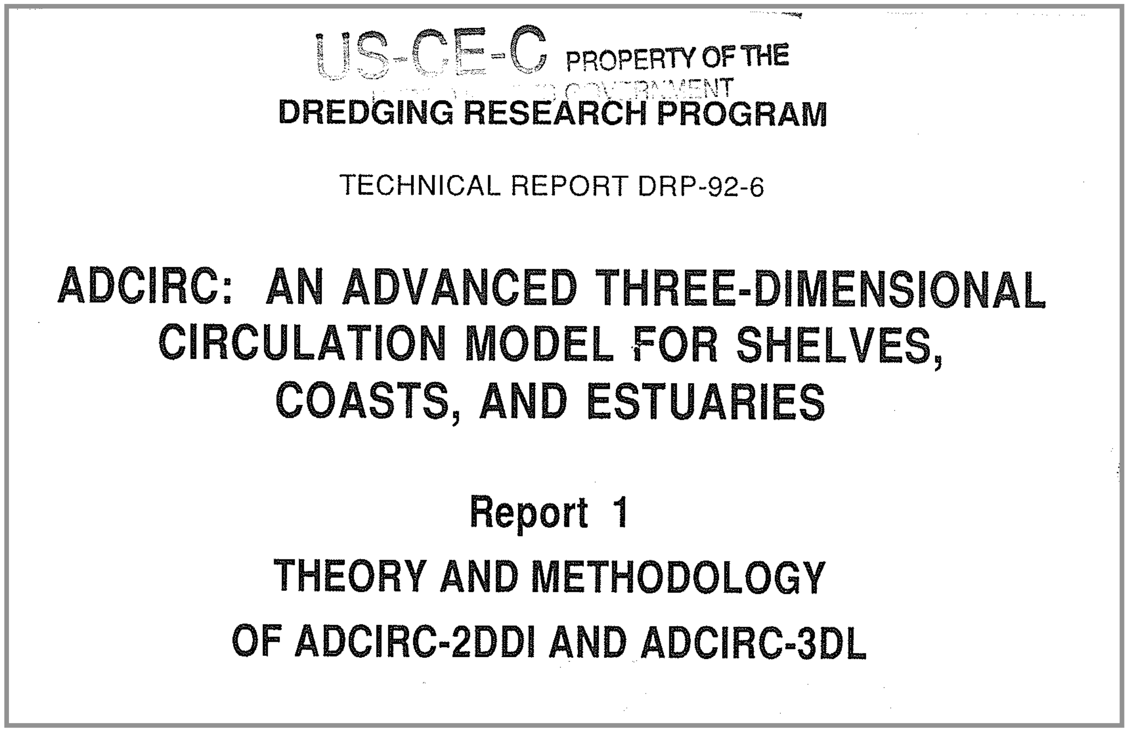
But what does the computational apparatus underlying ocean circulation models really look like?

- purely analytic functions? ✗
- an amalgam of data structures, algorithms, and . . .
  numerical expressions ✓

$$\boxed{\begin{array}{c}\text{scientific}\\ \text{programs}\end{array}} = \boxed{\begin{array}{c}\text{numerical}\\ \text{expressions}\end{array}} + \boxed{\begin{array}{c}\text{interstitial}\\ \text{machinery}\end{array}}$$

The Theory Report

Scientific computing lends itself well to a *refinement approach*, as most programs begin with a mathematical specification—often as a theory report—that serves as a guide, to one extent or another, in the implementation of high performance code.



US-CE-C PROPERTY OF THE GOVERNMENT

DREDGING RESEARCH PROGRAM

TECHNICAL REPORT DRP-92-6

ADCIRC: AN ADVANCED THREE-DIMENSIONAL CIRCULATION MODEL FOR SHELVES, COASTS, AND ESTUARIES

Report 1

THEORY AND METHODOLOGY OF ADCIRC-2DDI AND ADCIRC-3DL

Focused Application

$$\begin{tikzcd}
\boxed{\text{PDEs}} \arrow[d, Rightarrow, "\textit{discretization}"', "\textit{FEM, FD, FV}"] \\
\boxed{\text{Finite System of Equations}} \arrow[d, Rightarrow, "\textit{structure}"', "\textit{behavior}"] \\
\boxed{\text{Specification}} \arrow[d, Rightarrow, "\textit{object model}"', "\textit{parallelism}"] \\
\boxed{\text{Refinement}}
\end{tikzcd}$$

Lightweight in another sense: can draw useful conclusions about scientific software without simultaneously reproducing the sometimes deep, semantic proofs of numerical analysis.

Working with Numerical Expressions

Traditional number systems

$\mathbb{N} = \{0, 1, 2, \ldots\}$	the natural numbers
$\mathbb{Z} = \{m - n \mid m, n \in \mathbb{N}\}$	the integers
$\mathbb{Q} = \{m/n \mid m, n \in \mathbb{Z}, n \neq 0\}$	the rational numbers
$\mathbb{R}$	the real numbers
$\mathbb{C}$	the complex numbers

The sum and product on $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{Q}$ are those they inherit from $\mathbb{R}$.

$\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$ are fields, but not $\mathbb{N}$, since not all of its elements have a multiplicative inverse.

$$\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$$

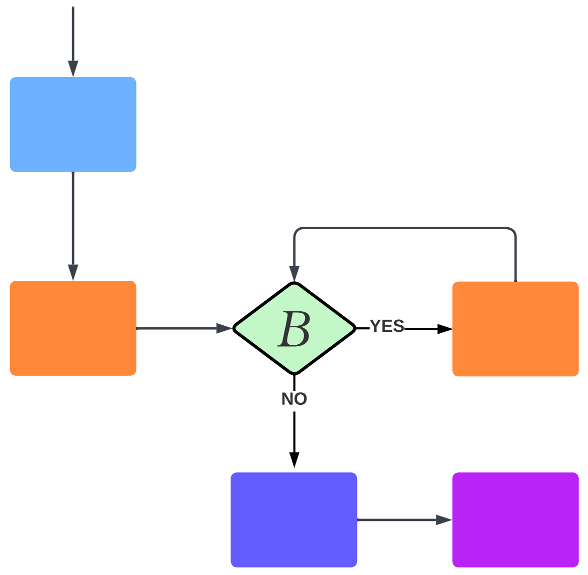
Only a subset of properties may be needed to draw useful conclusions.

Predicate Abstraction

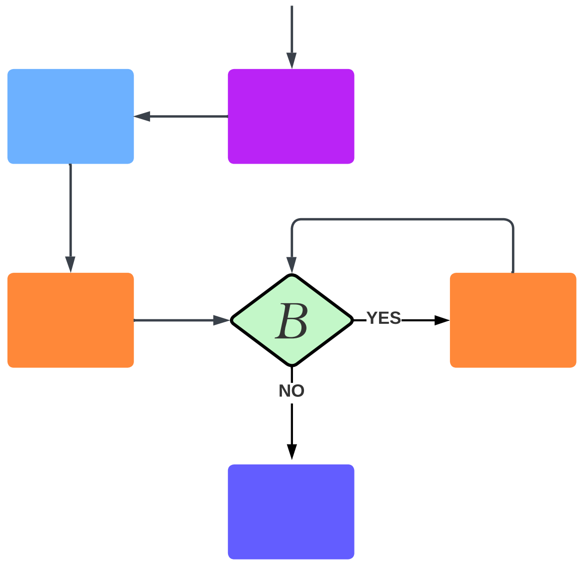
Floating point computations may determine the path of execution of a program, e.g.,

$$\textbf{if } V_{ss}(e) > V_{min} \textbf{ then } \ldots$$

If two models perform the same test and are being compared, we may substitute a common term, B, to yield a boolean program:



(a)



(b)

Other Abstraction Techniques

Numerical abstract domains

- May choose an abstract domain depending on the properties that need to be inferred.
- From *abstract interpretation*, a general theory of the approximations of program semantics.
- E.g., for safety arguments, can replace with interval arithmetic, as an **overapproximation**.

Other Abstraction Techniques

Numerical abstract domains

- May choose an abstract domain depending on the properties that need to be inferred.
- From *abstract interpretation*, a general theory of the approximations of program semantics.
- E.g., for safety arguments, can replace with interval arithmetic, as an **overapproximation**.

In other words:

We may not need a decision procedure for **real numbers**.

(and it may not be helpful even if we have one)

Example 1

Verifying an Extension to ADCIRC

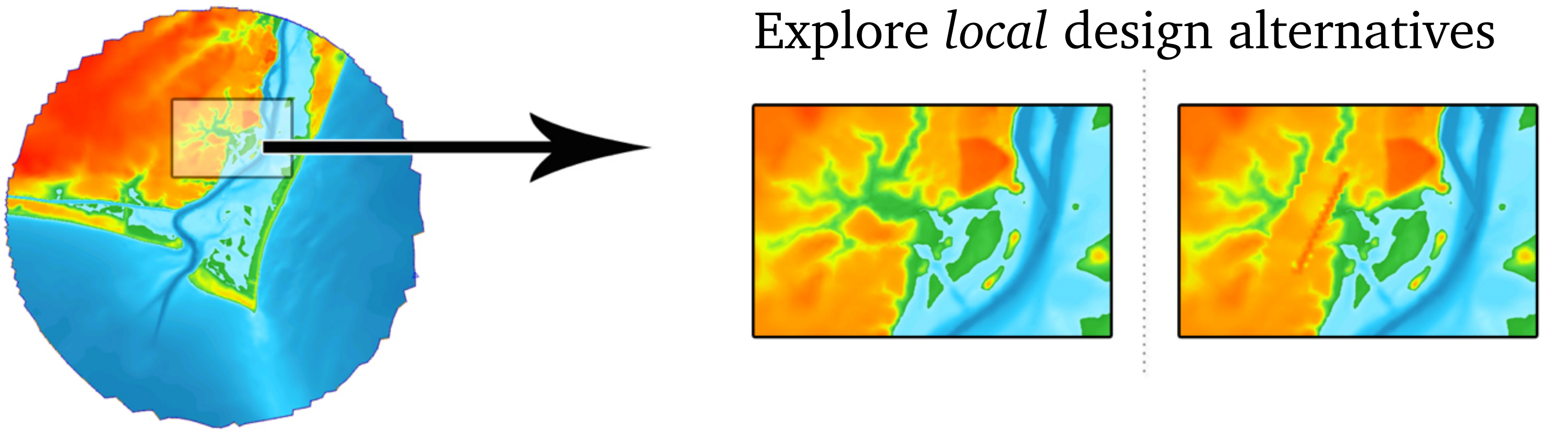
Formal methods and finite element analysis of hurricane storm surge: A case study in software verification. Baugh and Altuntas. *Science of Computer Programming*, 158:100–121, 2018.

ADCIRC Users

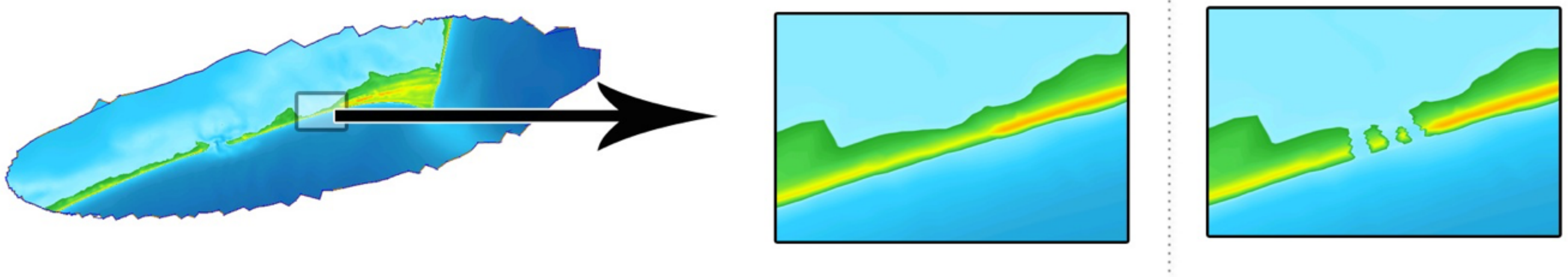
- U.S. Army Corps of Engineers
  - designing the $15B flood mitigation system in La.
- Federal Emergency Management Agency
  - evaluating flood risk on U.S. East & Gulf coasts
- National Oceanic and Atmospheric Administration
  - operational forecasting of tides and tropical storms
- U.S. Coast Guard
  - informing operational missions during hurricanes
- Nuclear Regulatory Commission
  - assessing flood risk to coastal nuclear power plants

Subdomain Modeling

Explore *local* design alternatives



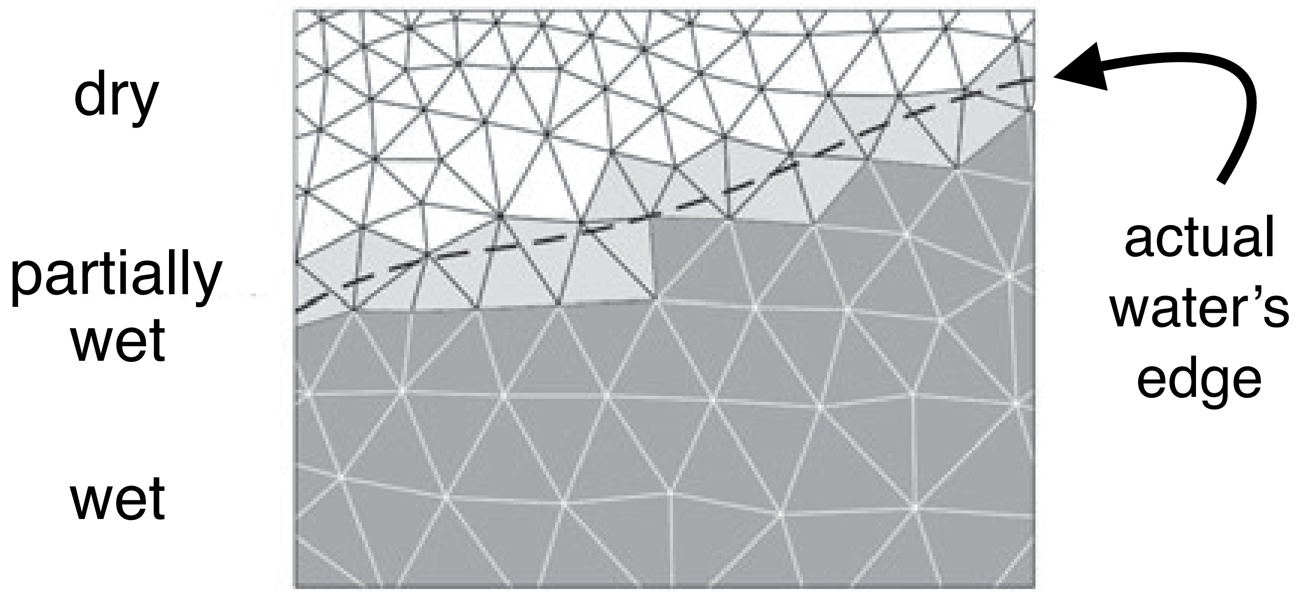
Investigate *local* failure scenarios



An exact reanalysis technique for storm surge and tides in a geographic region of interest. Baugh, Altuntas, Dyer, & Simon. *Coastal Engineering, 97,* 60–77, 2015.

Contingent Processes

- **Wetting and drying:** allows the propagation of overland flows
- Derived from simplified physics, encoded with **empirical rules:**



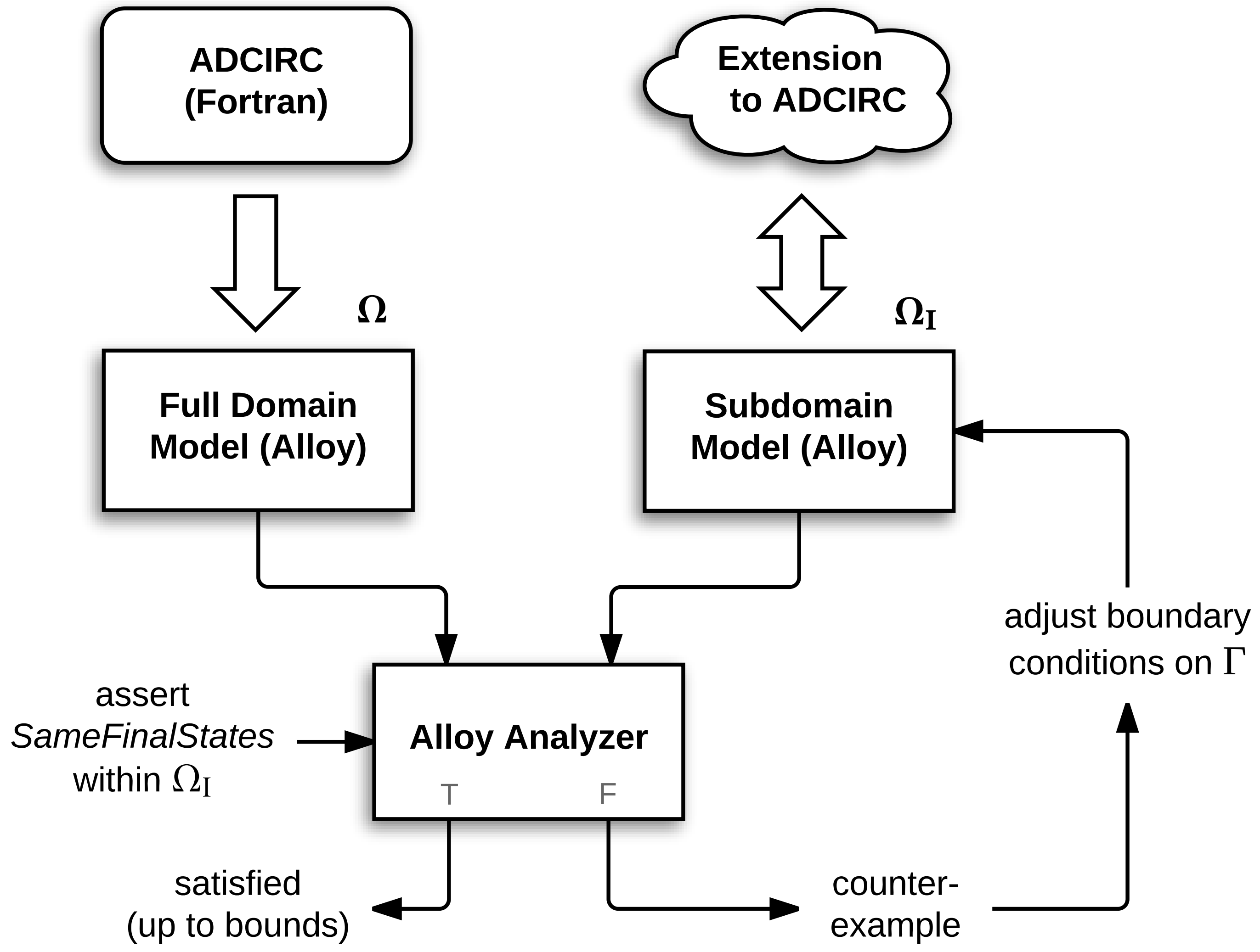
(Medeiros and Hagen, 2013)

Result is the wet-dry status of each node: wet or dry?

Wetting and Drying Algorithm

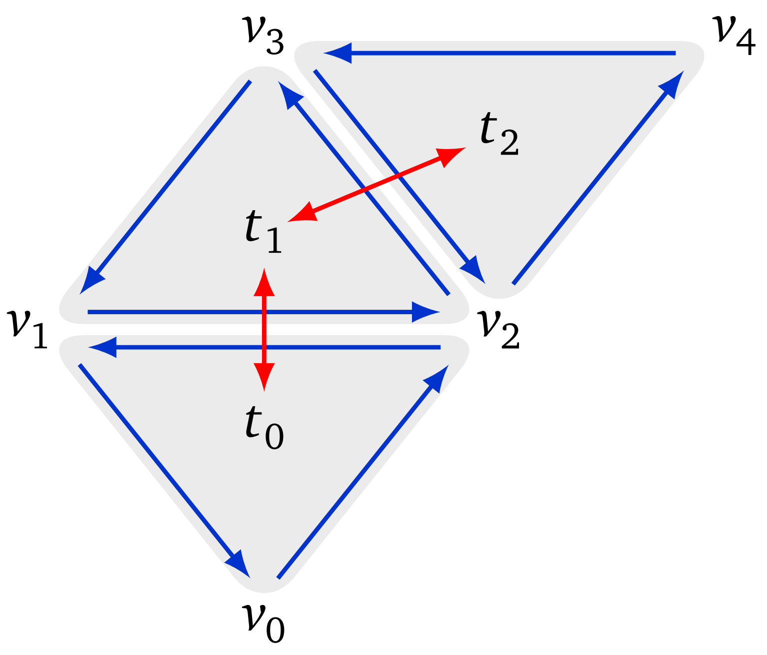
```
0: for $e$ in $elements$ do                                    ▷ initialization: start with all elements being wet
       $wet_e \leftarrow$ true
1: for $n$ in $nodes$ do                                       ▷ make nodes with low water column height dry
       if $W_n$ and $H_n < H_{min}$ then
           $W_n \leftarrow$ false, $W_n^t \leftarrow$ false
2: for $e$ in $elements$ do                                    ▷ propagate wetting unless flow is slow
       if $\neg W_i$ for exactly one node $i$ on $e$ and $V_{ss}(e) > V_{min}$ then
           $W_i^t \leftarrow$ true
3: for $e$ in $elements$ do                                    ▷ let water build up on incline
       find nodes $i$ and $j$ of $e$ with highest water surface elevations $\eta_i$ and $\eta_j$
       if $\min(H_i, H_j) < 1.2H_{min}$ then
           $wet_e \leftarrow$ false
4: for $n$ in $nodes$ do                                       ▷ make landlocked nodes dry
       if $W_n^t$ and $n$ on only inactive elements then
           $W_n^t \leftarrow$ false
5: for $n$ in $nodes$ do                                       ▷ set the final wet-dry state for nodes
       $W_n \leftarrow W_n^t$
```

Verification Approach



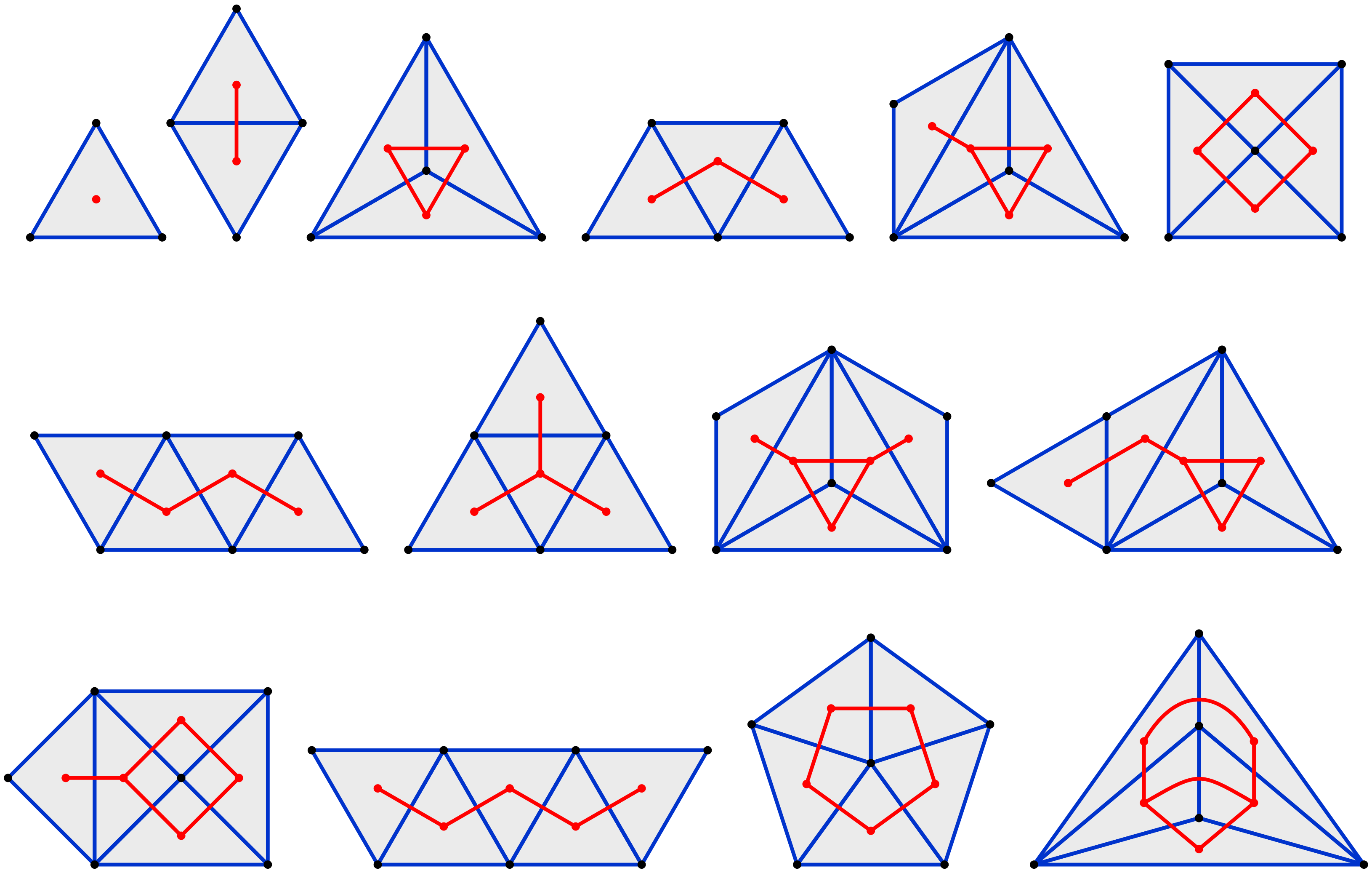
Representing a Mesh

```
sig Mesh {
  triangles: some Triangle,
  adj: Triangle → Triangle
}

sig Triangle {
  edges: Vertex → Vertex
}

sig Vertex {}
```

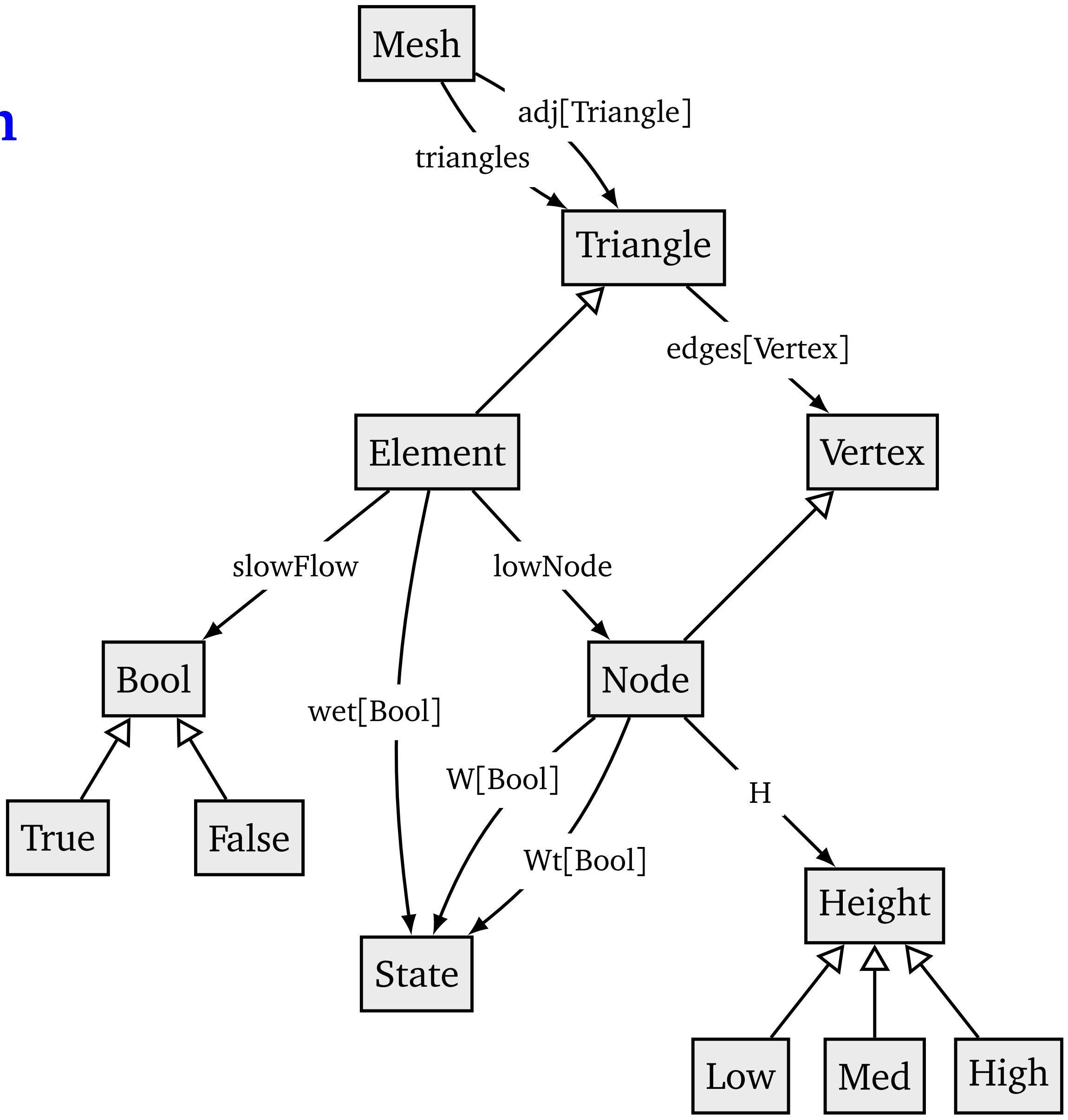


Require that triangles have three directed edges and be oriented, that meshes be connected, oriented, and non-overlapping, and that cut points be prevented.

A Menagerie of Mesh Topologies



Model Diagram



Observations

Alloy facilitates experimentation with

- the amount of state needed along the interface
- the manner in which that state is used to enforce boundary conditions

Observations

Alloy facilitates experimentation with

- the amount of state needed along the interface
- the manner in which that state is used to enforce boundary conditions

Alloy's strength is model *finding*

- mesh topologies are defined implicitly by declarative properties: no algorithm need be devised to produce them (as in a testing scenario, for instance)

Example 2

Sparse Matrix Computations

Bounded verification of sparse matrix computations. Dyer, Altuntas, and Baugh. In *Proceedings of the Third International Workshop on Software Correctness for HPC Applications, Correctness'19*, pages 36–43. IEEE/ACM, 2019.

Sparse Matrix Operations

Discretization of PDEs into finite systems of equations

- often working with sparse matrix formats: Ellpack (ELL), Compressed Sparse Row (CSR), etc.
- **interdependencies: sparse format $\longleftrightarrow$ solver**
- direct assembly from meshes, application of boundary conditions, manipulation by wetting and drying schemes

Sparse Matrix Operations

Discretization of PDEs into finite systems of equations

- often working with sparse matrix formats: Ellpack (ELL), Compressed Sparse Row (CSR), etc.
- **interdependencies: sparse format $\longleftrightarrow$ solver**
- direct assembly from meshes, application of boundary conditions, manipulation by wetting and drying schemes

A symbolic representation of sparse matrices, together with a new idiom for stateful behavior

- multiplication, transpose, translation between formats, etc.

Verification Approach

Notion of conformance: **Inclusion**

$$\begin{tikzcd} A \arrow[r, "OP_A"] & A' \\ C \arrow[u, "\alpha"] \arrow[r, "OP_C"'] & C' \arrow[u, "\alpha"'] \end{tikzcd}$$

- **Data Refinement**

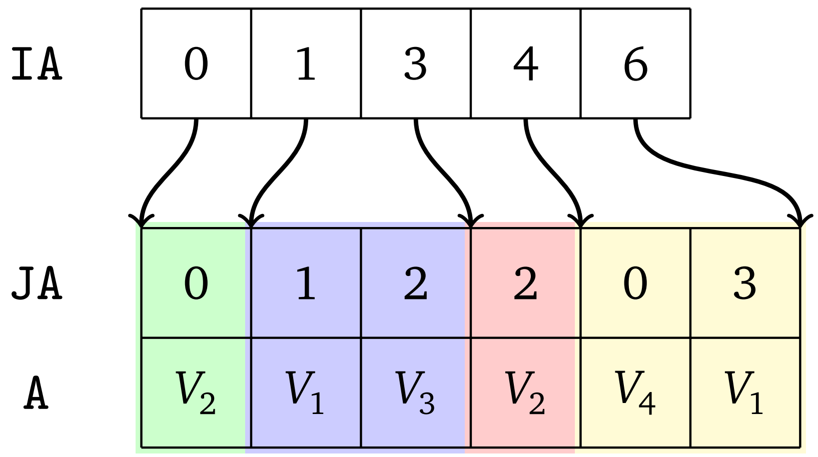
  A detailed concrete system simulates a more abstract one.

- **Weak commutativity**

$$I(C) \wedge OP_C(C, C') \wedge \alpha(C, A) \wedge \alpha(C', A') \Rightarrow OP_A(A, A')$$

Refinement from Dense to CSR Format

$$\begin{pmatrix} V_2 & Z & Z & Z \\ Z & V_1 & V_3 & Z \\ Z & Z & V_2 & Z \\ V_4 & Z & Z & V_1 \end{pmatrix}$$



```
sig Value {}
one sig Zero extends Value {}

sig Matrix {
   rows, cols: Int,
   vals: Int → Int → lone Value
}
```

```
sig CSR {
   rows, cols: Int,
   IA, JA: Int → lone Int,
   A: Int → lone Value
}
```

Observations

- Easy to extract and spot-check fragments from large code bases
  - “Surrounding state” generated by model finding
  - Bugs/inconsistencies found between code and documentation

Observations

- Easy to extract and spot-check fragments from large code bases
  - “Surrounding state” generated by model finding
  - Bugs/inconsistencies found between code and documentation
- Model structure similar to imperative code via *tabular idiom*

Observations

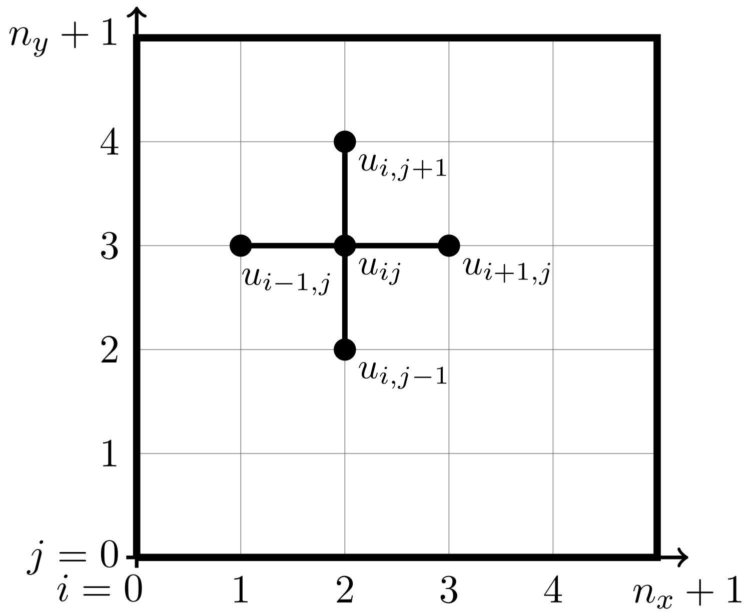
- Easy to extract and spot-check fragments from large code bases
  - "Surrounding state" generated by model finding
  - Bugs/inconsistencies found between code and documentation
- Model structure similar to imperative code via *tabular idiom*
- Most analyses take on the order of seconds to minutes

Example 3

Laplace Solver in Coarray Fortran

An HPC practitioner's workbench for formal refinement checking. Benavides, Baugh, and Gopalakrishnan. In *Languages and Compilers for Parallel Computing*, LCPC 2022, pages 64–72, 2023. Springer, Lecture Notes in Computer Science, vol. 13829.

Jacobi Iteration

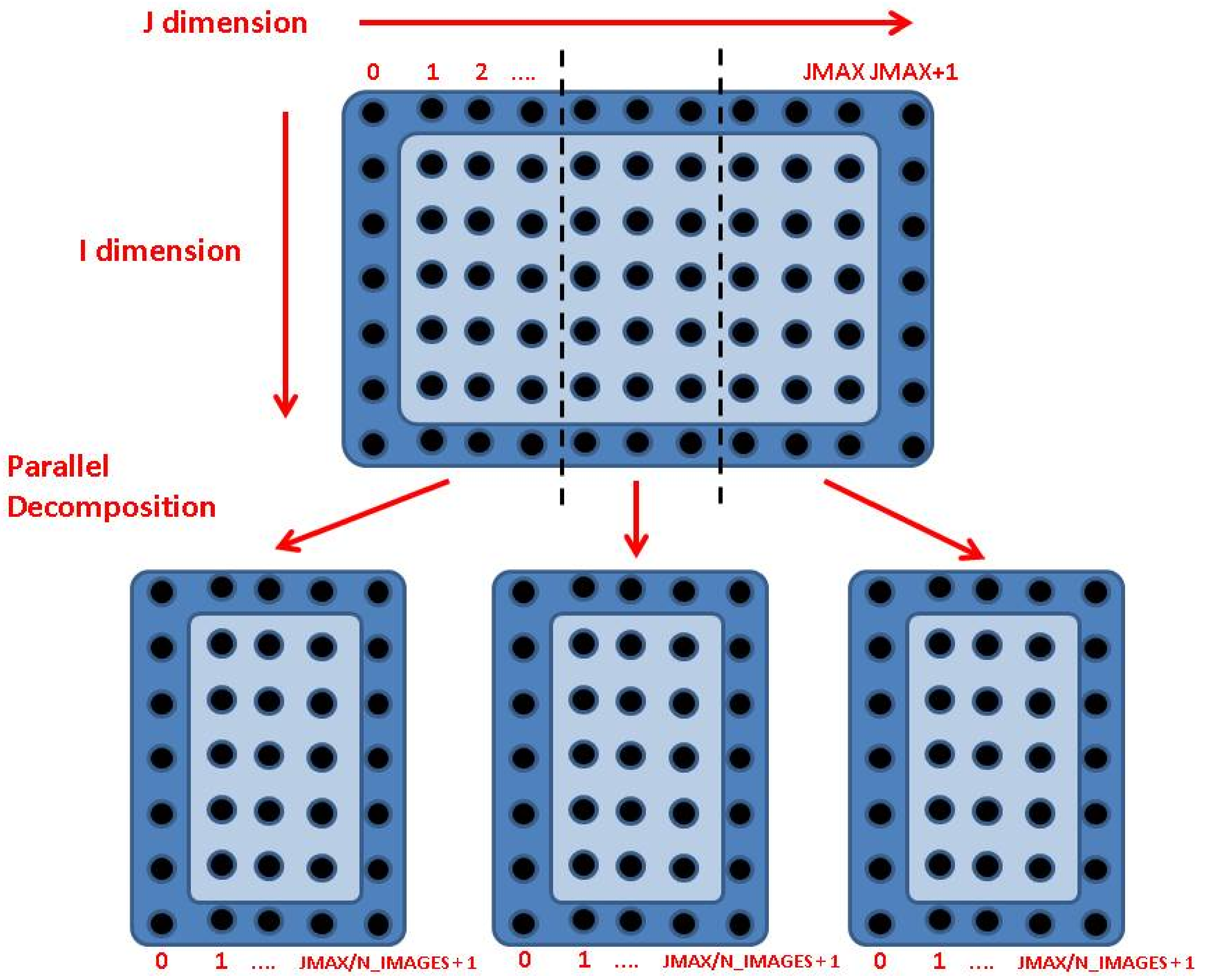


Fixed point pattern for Jacobi iteration:

$$u_{ij}^{k+1} = \frac{1}{4}(u_{i+1,j}^{k} + u_{i-1,j}^{k} + u_{i,j+1}^{k} + u_{i,j-1}^{k})$$

How might one verify an implementation in Coarray Fortran?

Domain Decomposition



Martin, J.M.R. Testing and verifying parallel programs using data refinement. In *Communicating Process Architectures 2017 & 2018*, pp. 491–500. IOS Press (2019).

Implementation and Approach

Coarray Fortran (Partitioned Global Address Space)

- Partitioned: programmer controls data layout across **images**
- Global: can directly access remote memory

Implementation and Approach

Coarray Fortran (Partitioned Global Address Space)

- Partitioned: programmer controls data layout across **images**
- Global: can directly access remote memory

Halo exchange

- Duplicate columns at the **image** interfaces
- Allows work in stages: computation followed by communication

Implementation and Approach

Coarray Fortran (Partitioned Global Address Space)

- Partitioned: programmer controls data layout across **images**
- Global: can directly access remote memory

Halo exchange

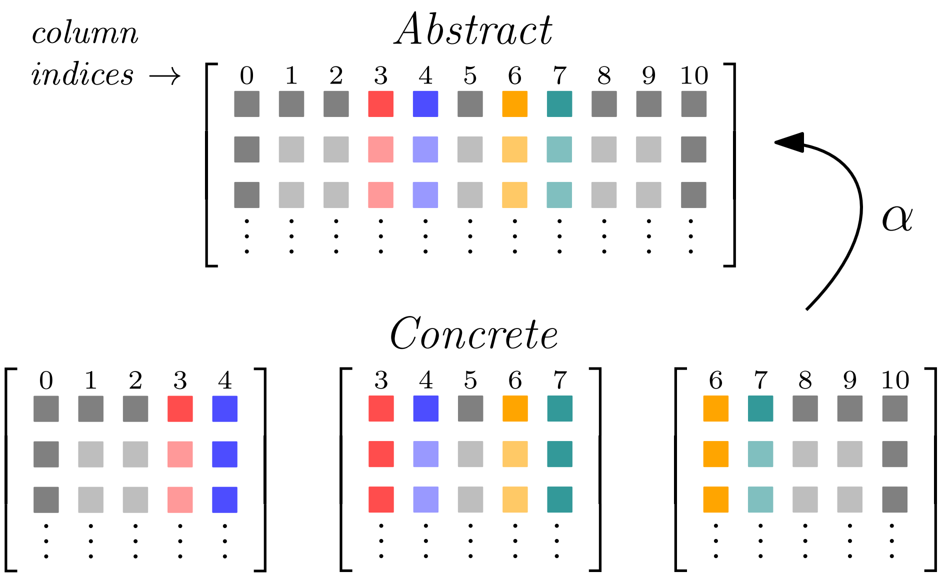
- Duplicate columns at the **image** interfaces
- Allows work in stages: computation followed by communication

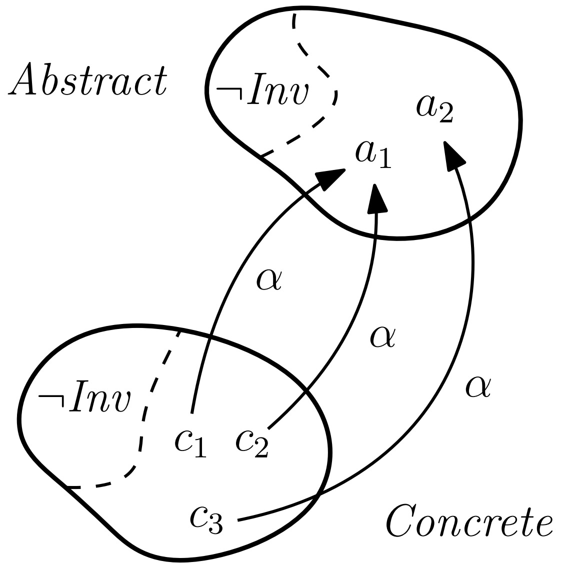
Verification approach

- Data refinement, formalize what is meant by an **image**

Abstraction Function and Invariants

Relate concrete and abstract state spaces by an abstraction function α:





```
// a sequence of images
sig Coarray {
  mseq: seq Matrix
}
```

Find concrete invariants: uniform shape of images, overlapping columns at the interfaces for border exchanges, etc.

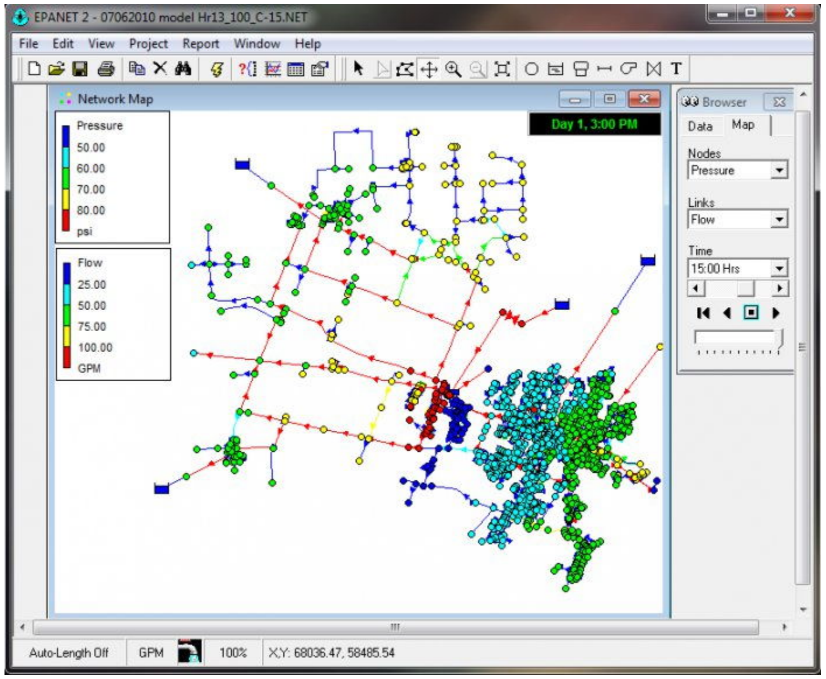
3. The ***How***

An Engineering Application

Public water systems depend on **water distribution networks** to provide an uninterrupted supply of safe drinking water.

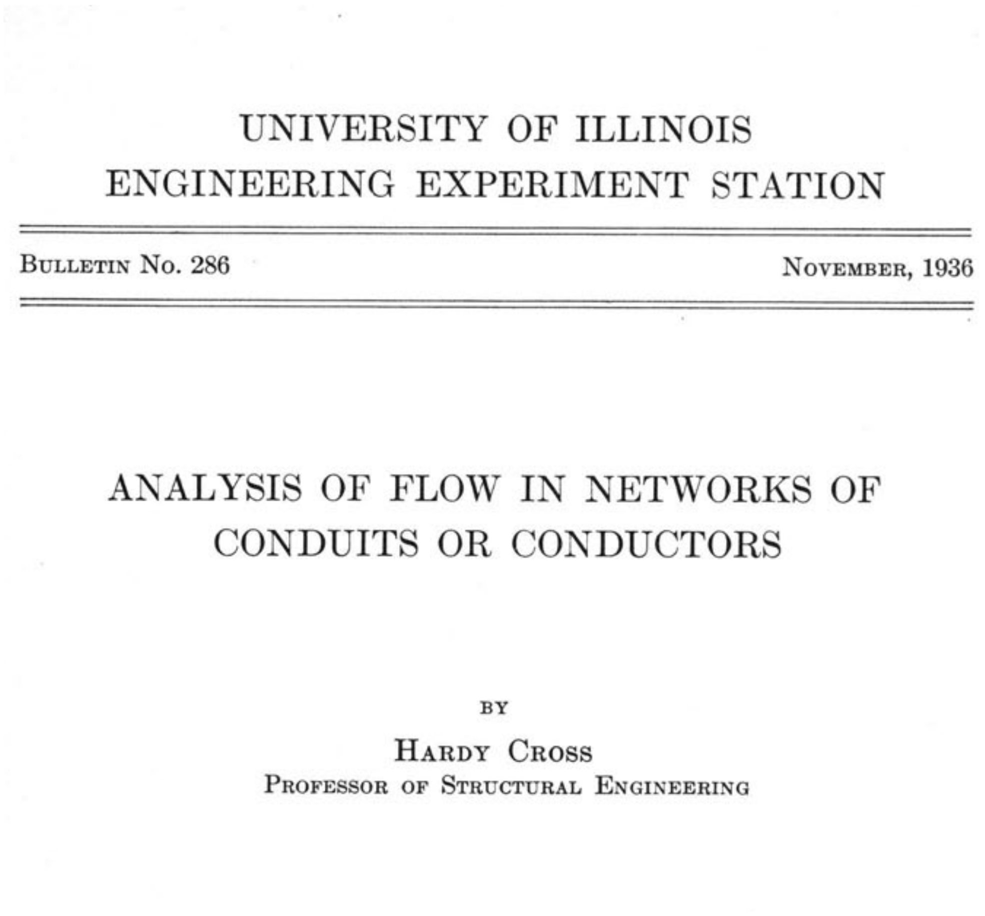
Engineers attempt to design low-cost systems that meet the hydraulic requirements of flow and pressure.

EPANET is a widely-used software system developed by the US EPA to model and simulate such systems.



Historical Context

In the 1930s, a breakthrough by Hardy Cross in pipe network analysis:



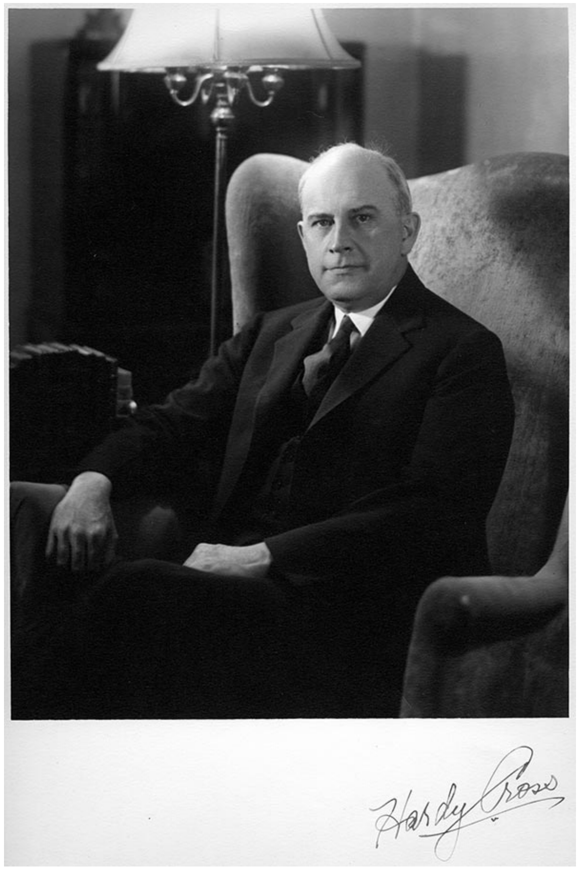
UNIVERSITY OF ILLINOIS
ENGINEERING EXPERIMENT STATION

BULLETIN No. 286 NOVEMBER, 1936

ANALYSIS OF FLOW IN NETWORKS OF
CONDUITS OR CONDUCTORS

BY

HARDY CROSS
PROFESSOR OF STRUCTURAL ENGINEERING



The Hardy Cross Method

A **hand method** that revolutionized municipal water supply design

- Iteratively determines the flow in pipe network systems
- Nonlinear relationship between head loss and flow had been a challenging problem
- Remains as the method taught to most civil engineering students

First computer implementation

In 1957, **Hoag and Weinberg** adapted the Hardy Cross method for solving the network flow problem to the digital computer and applied the method to the water distribution system of the city of Palo Alto, California.

Basic Principles

“The physical conditions controlling engineering relations often consist of two groups of laws which are quite independent of each other.”

Conservation of mass

- *Continuity of flow:*

  the total flow reaching any junction equals the total flow leaving it

Conservation of energy

- *Continuity of potential:*

  the total change in potential along any closed path is zero

Iterative Approach

Balancing heads (or *loop* method)

1. Assume any distribution of flow satisfying continuity.
2. Compute in each pipe the (nonlinear) loss of head.
3. Set up in each circuit a counterbalancing flow to balance the head.
4. Compute the revised flows and repeat the procedure.

Iterative Approach

Balancing heads (or *loop* method)

1. Assume any distribution of flow satisfying continuity.
2. Compute in each pipe the (nonlinear) loss of head.
3. Set up in each circuit a counterbalancing flow to balance the head.
4. Compute the revised flows and repeat the procedure.

Computational aspects?

What would it mean to verify correctness?

Hardy Cross Verification

Problem: Would want to show ...

```
{**whileinv** $I$: *continuity of flow*}          ← *Loop invariant*
**while** *heads in any loops are unbalanced*
    *revise flows in each loop while maintaining invariant*
**end**
{*continuity of flow* $\wedge$ *all heads balanced*}          ← *Postcondition*
```

Counterbalancing flow around a loop to balance the head:

$$\Delta Q = -\frac{\sum rQ\,|Q|^{n-1}}{\sum rn\,|Q|^{n-1}}$$

Hardy Cross Verification

Problem: Would want to show ...

```
{**whileinv** $I$: *continuity of flow*}          ← *Loop invariant*
**while** *heads in any loops are unbalanced*
    *revise flows in each loop while maintaining invariant*
**end**
{*continuity of flow* $\wedge$ *all heads balanced*}          ← *Postcondition*
```

We must show that **whileinv** I is indeed invariant:

Let $I = \textit{continuity of flow}$ then

$$I \wedge \textit{revise flows in each loop} \Rightarrow I$$

Hoag and Weinberg: Improved Convergence

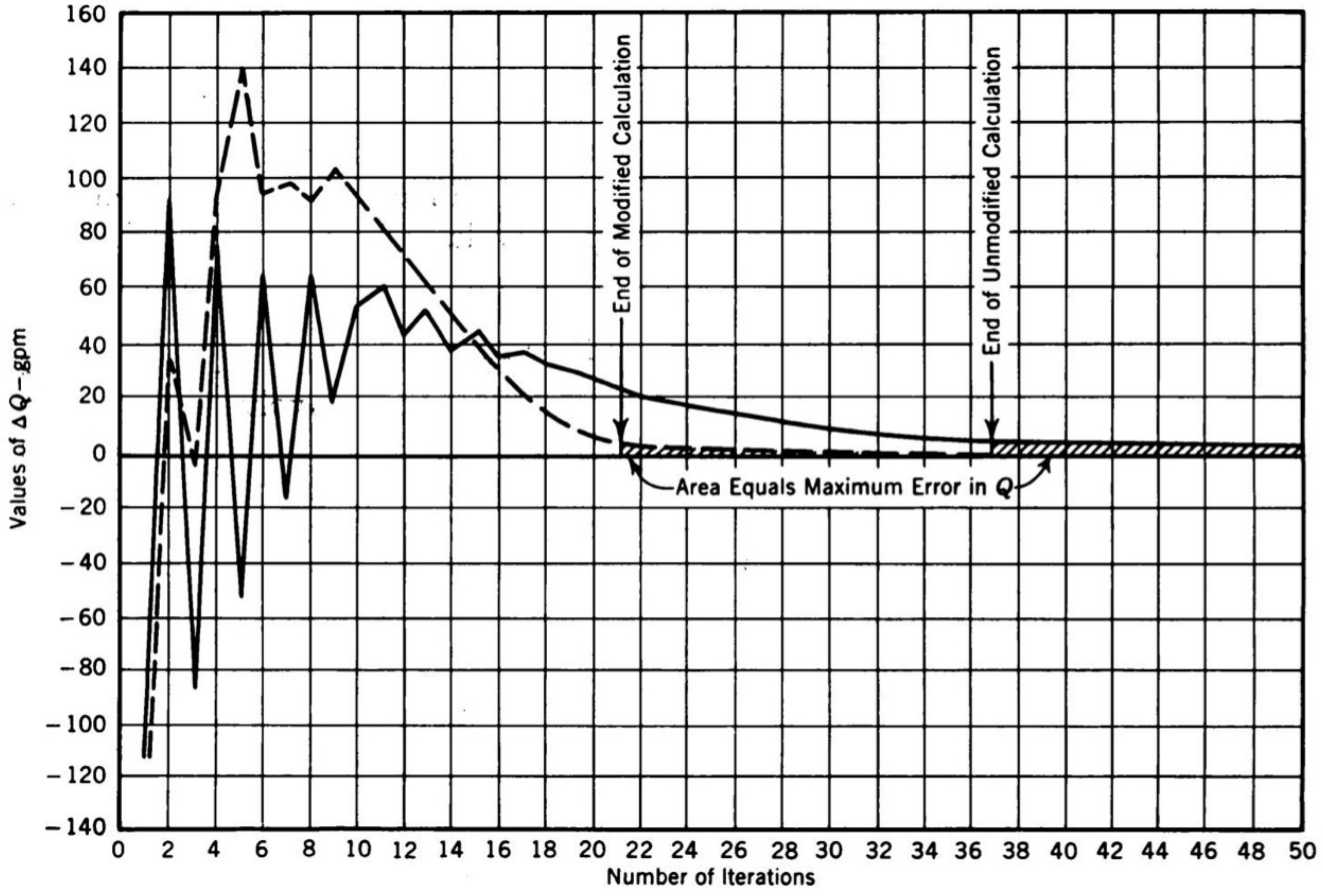


Fig. 3. Curves Showing Typical Fluctuation of Flow Rate Corrections for Solutions With Modified and Unmodified Programs

Prove Convergence?

As with Newton-Raphson and other nonlinear solvers, the Hardy Cross method may fail to converge.

Does that mean verification is impossible?

Prove Convergence?

As with Newton-Raphson and other nonlinear solvers, the Hardy Cross method may fail to converge.

Does that mean verification is impossible?

It’s certainly possible to implement the Hardy Cross method, and the result may be either correct or incorrect, right?

Prove Convergence?

As with Newton-Raphson and other nonlinear solvers, the Hardy Cross method may fail to converge.

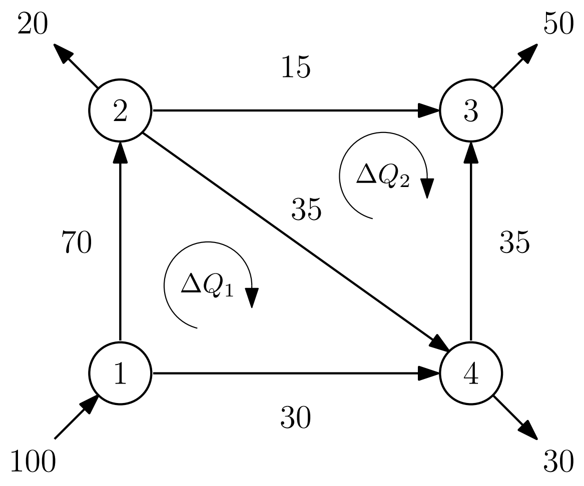
Does that mean verification is impossible?

It's certainly possible to implement the Hardy Cross method, and the result may be either correct or incorrect, right?

We can recognize that convergence is a termination condition, and prove safety:

If the program terminates, its results are correct.

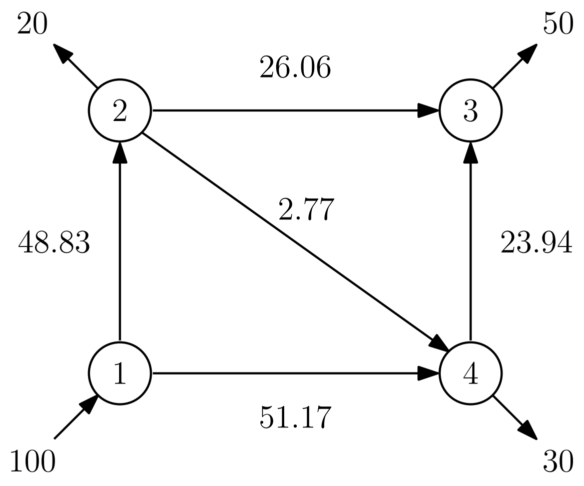
What a step in the algorithm looks like:



Initial flows

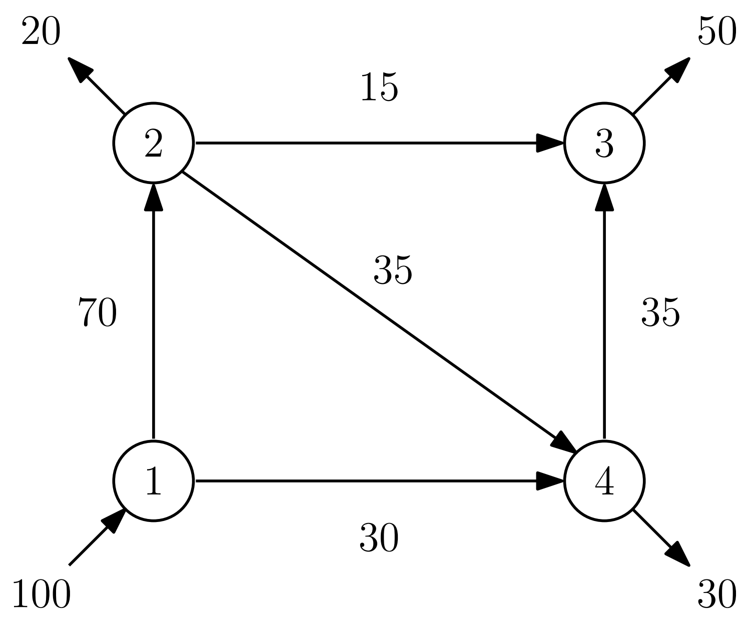
$\Rightarrow$

$$\boxed{\begin{array}{c}\Delta Q_1 = -21.17 \\ \Delta Q_2 = 11.06\end{array}}$$

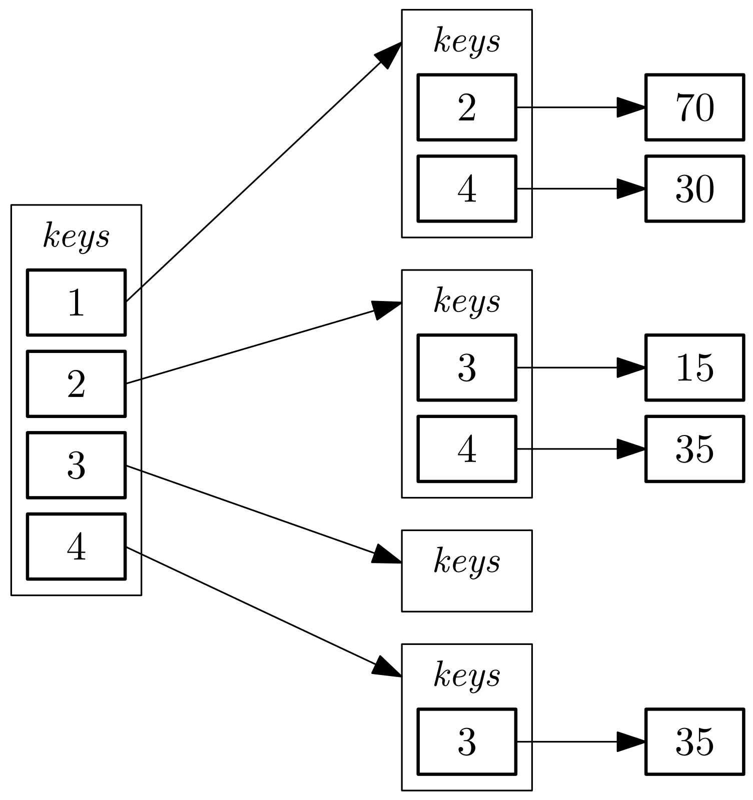


Updated flows

Simple Network Representation



Directed Graph



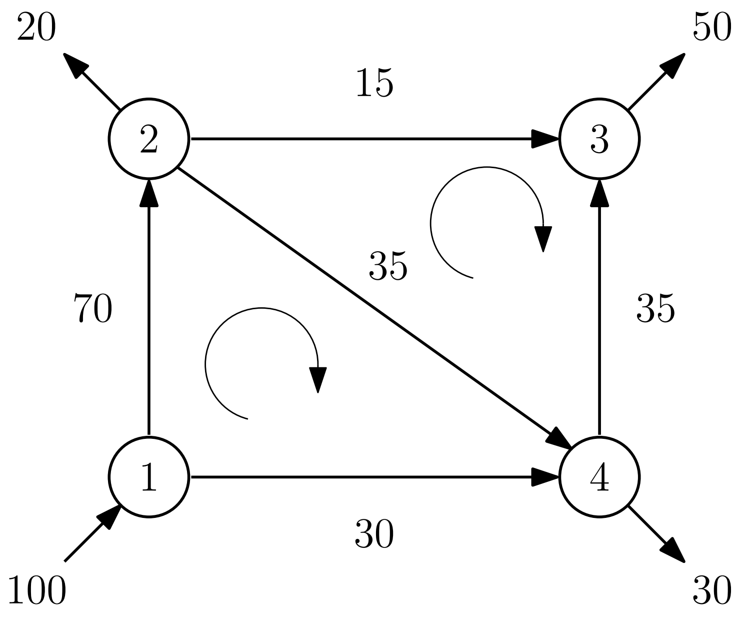
Adjacency Structure

Network Representation in Julia

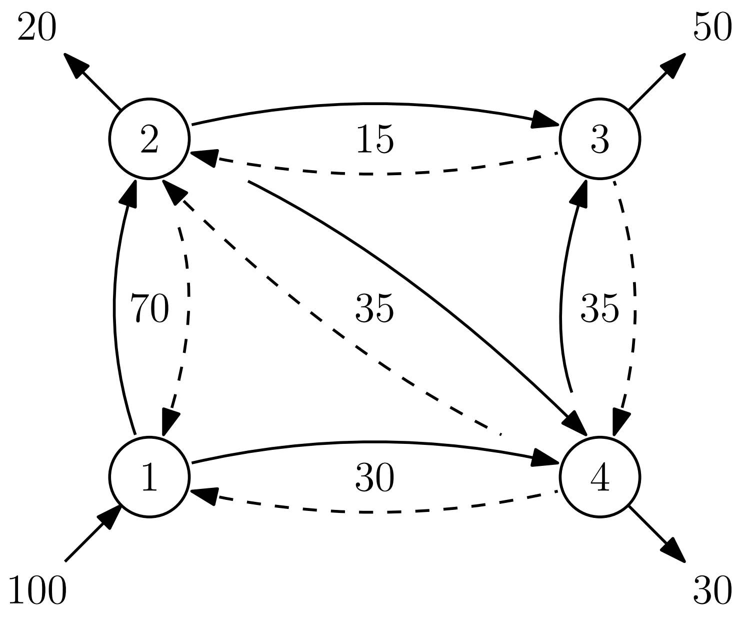
```
# nodal flows                  # edge flows
n = Dict()                     e = Dict([i => Dict() for i in 1:4])

n[1] = 100                     e[1][2] = 70
n[2] = -20                     e[1][4] = 30
n[3] = -50                     e[2][3] = 15
n[4] = -30                     e[2][4] = 35
                               e[4][3] = 35
```

```
# helper functions

out_edges(d, i) = d[i]

in_edges(d, j) = Dict([i => d'[j] for (i, d') in d
                                       if j in keys(d')])
```

Forward and Reverse Edges



Forward edges only



Forward and reverse edges

Modeling Exercise

Wish to show

Let $I = \textit{continuity of flow}$ then

$$I \wedge \textit{revise flows in each loop} \Rightarrow I$$

Modeling Exercise

Wish to show

> Let I = *continuity of flow* then
>
> $I \wedge \textit{revise flows in each loop} \Rightarrow I$

And do so in the context of an **object model** that captures details:

- **Alloy** model with forward and reverse edges, edge signs, etc.
- Network representation that's a little more complex, for efficiency
- Loop invariant I will not hold if we have conceptual errors

Modeling Exercise

Wish to show

Let I = *continuity of flow* then

$I \wedge \textit{revise flows in each loop} \Rightarrow I$

And do so in the context of an **object model** that captures details:

- **Alloy** model with forward and reverse edges, edge signs, etc.
- Network representation that's a little more complex, for efficiency
- Loop invariant I will not hold if we have conceptual errors

Carry over the global invariants found into actual code, e.g., Julia:

"A view of object models as heap invariants"

Working with Alloy

Atoms are Alloy’s primitive entities

- indivisible, immutable, uninterpreted

Working with Alloy

Atoms are Alloy’s primitive entities

- indivisible, immutable, uninterpreted

Relations associate atoms with one another

- sets of tuples
- tuples are sequences of atoms

Working with Alloy

Atoms are Alloy’s primitive entities

- indivisible, immutable, uninterpreted

Relations associate atoms with one another

- sets of tuples
- tuples are sequences of atoms

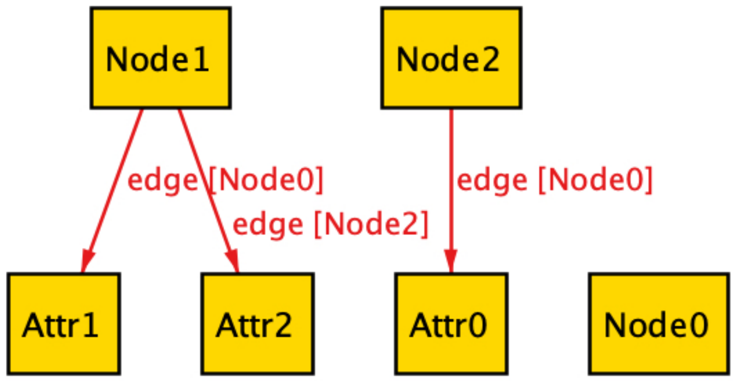
Every value in Alloy logic is a relation

- relations, sets, and scalars are all the same thing

Alloy Model and a Snapshot

An Alloy model:

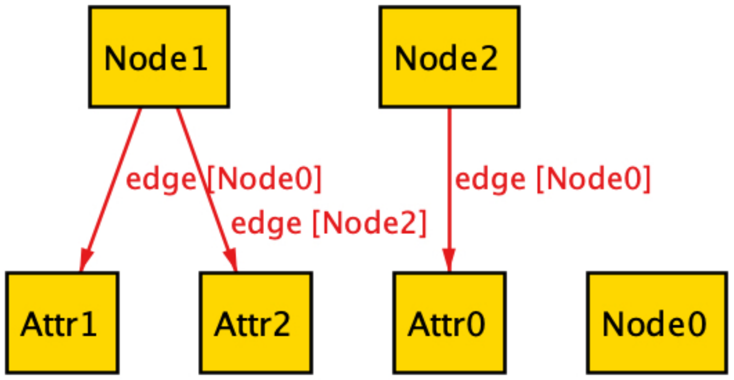
```
sig Attr {}

sig Node {
  edge: Node->Attr
}
```



Alloy Model and a Snapshot

An Alloy model:

```
sig Attr {}

sig Node {
  edge: Node->Attr
}
```



Sigs define unary relations:

```
Attr = {(Attr0),
        (Attr1),
        (Attr2)}

Node = {(Node0),
        (Node1),
        (Node2)}
```

Fields define *n*-ary relations:

```
edge = {(Node1, Node0, Attr1),
        (Node1, Node2, Attr2),
        (Node2, Node0, Attr0)}
```

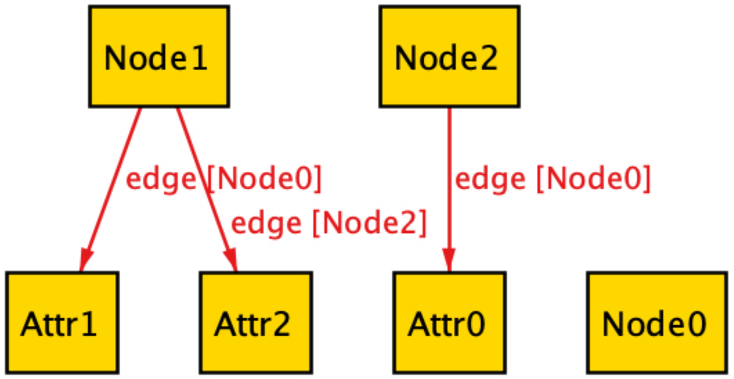
Dot join and box join operators:

```
Node1.edge[Node2] = {(Attr2)}
```

Alloy Model and a Snapshot

An Alloy model:

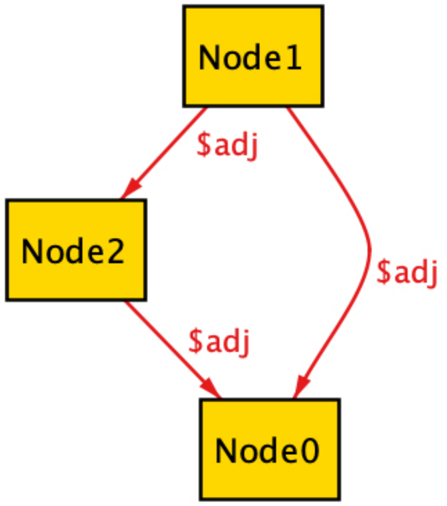
```
sig Attr {}

sig Node {
  edge: Node->Attr
}
```



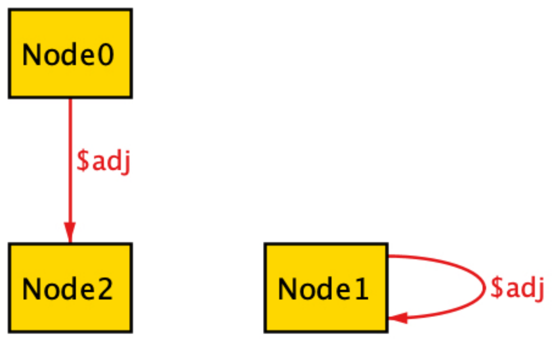
Visualize adjacency:

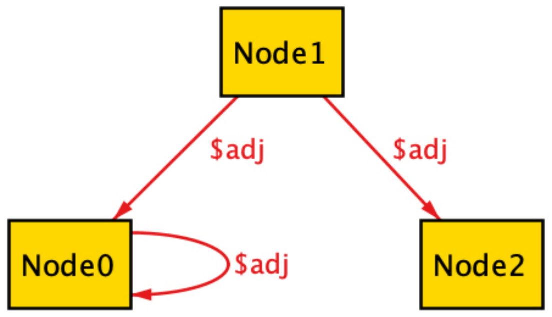
```
fun adj: Node->Node {
  edge.Attr
}
```

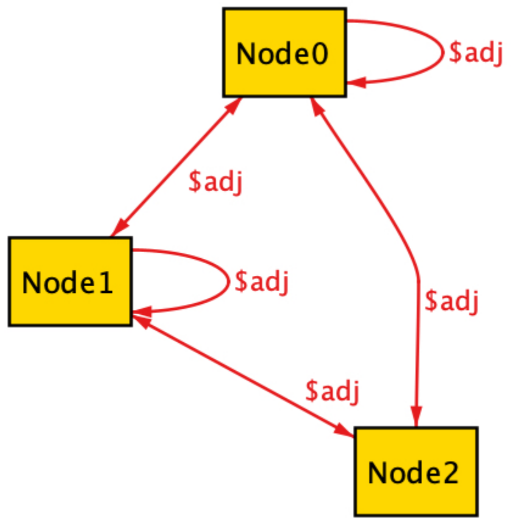
(a function of no arguments is treated as a named relation, a skolem constant)



Snapshots: More Instances

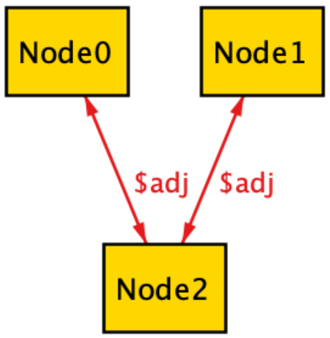


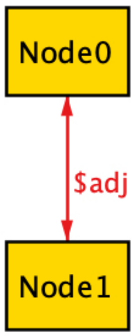


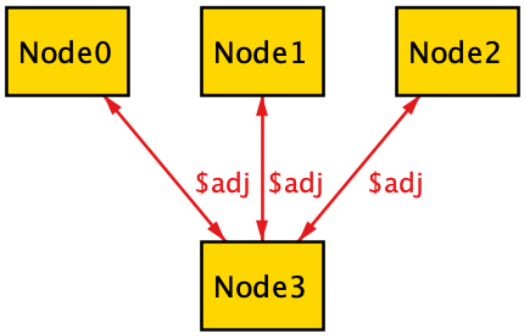


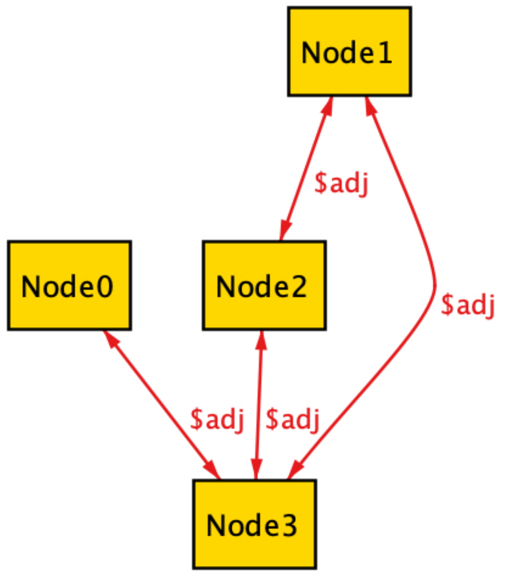
Snapshots: More Instances (after adding facts)

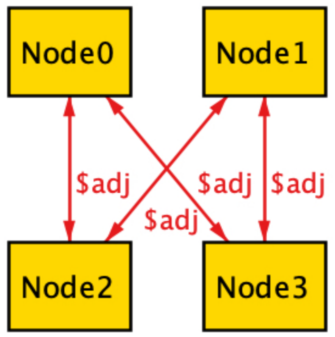
```
fact {
  -- adj is symmetric (~ is transpose)
  adj = ~adj

  -- no loops (& is intersection, iden is identity)
  no iden & adj

  -- graph is connected (* is transitive closure)
  all n: Node | Node in n.*adj
}
```

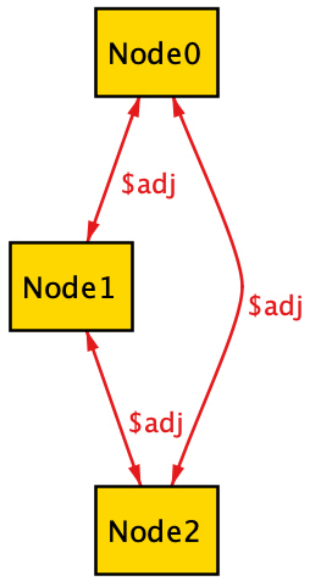


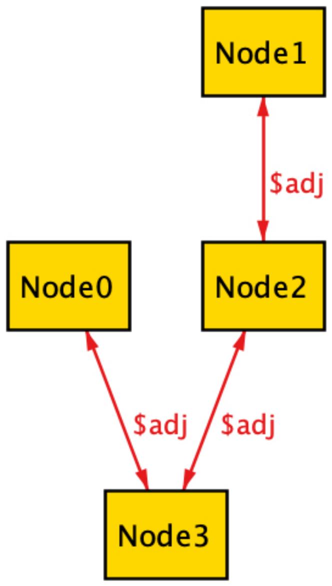


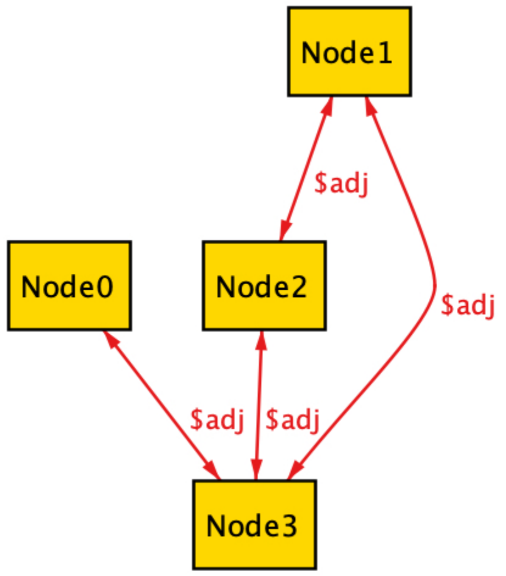



Node1
$adj
Node0
Node2
$adj
$adj
$adj
Node3

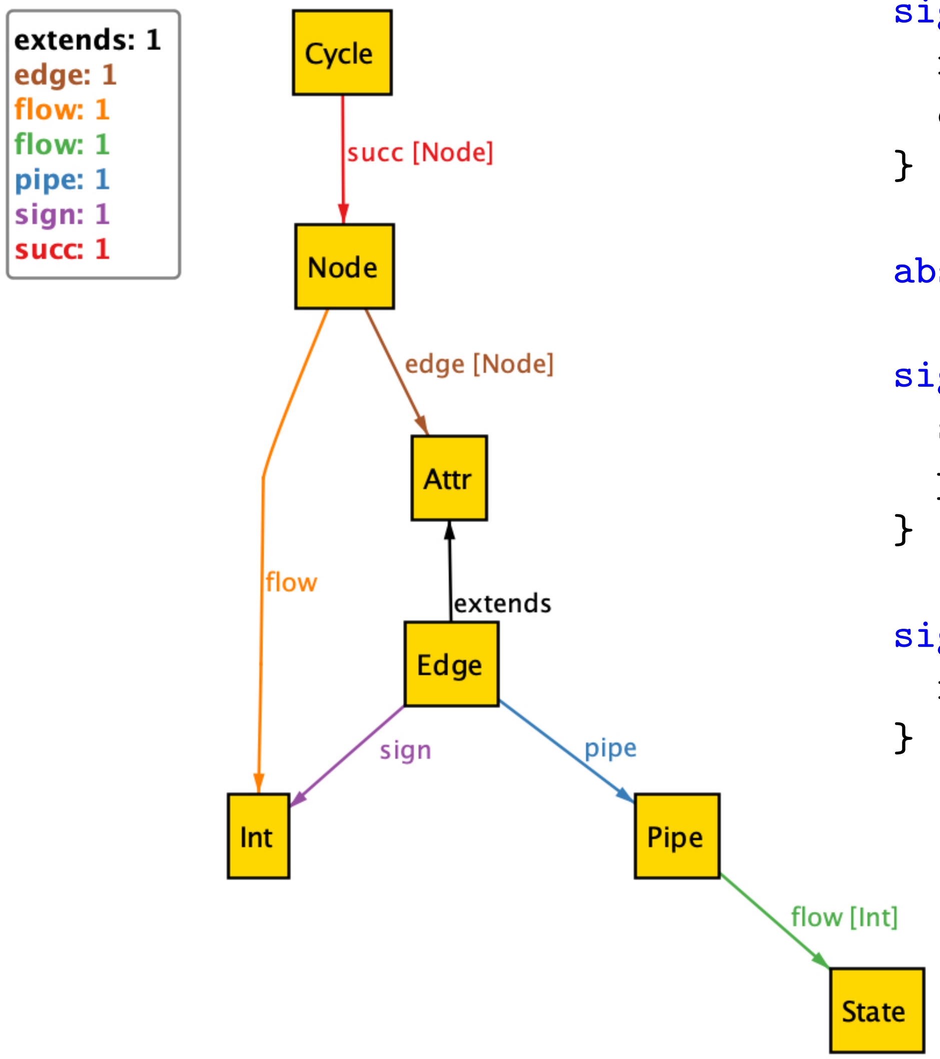

Node0
Node1
$adj
$adj
$adj
$adj
$adj
Node2
Node3


Node0
$adj
Node1
$adj
$adj
Node2


Node1
$adj
Node0
Node2
$adj
$adj
Node3


Node1
$adj
Node0
Node2
$adj
$adj
$adj
Node3

Extending the Model



```
sig Node {
  flow: Int,
  edge: Node->Attr
}

abstract sig Attr {}

sig Edge extends Attr {
  sign: Int,
  pipe: Pipe
}

sig Pipe {
  flow: Int one -> State
}
```

A Sample of Assertions (Heap Invariants)

Edge signs:

```
-- edge signs must be either -1 or +1
fact { all e: Edge | e.sign in -1 ++ 1 }

-- anti-parallel edges have opposite signs
fact { all a, b: Node | a->b ++ b->a in adj implies
         add[a.edge[b].sign, b.edge[a].sign] = 0
}
```

Multiplicities:

```
-- an edge is referenced by just one vertex pair
fact { all e: Edge | one edge.e }

-- a pipe is shared by two anti-parallel edges
fact { all p: Pipe | antiparallel_pair[edge.(pipe.p)] }
```

Flow Computations

```
// Net flow at a node

fun net_flow [u: Node, s: State] : Int {
  sub[u.flow, sum v: Node |
                 let e = u.edge[v] | mul[e.sign, e.pipe.flow.s]]
}

// Increment flow in a pipe

pred inc_flow [u, v: Node, q: Int, s, s": State] {
  let e = u.edge[v] |
    e.pipe.flow.s" = add[e.pipe.flow.s, mul[e.sign, q]]
```

Verifying the Loop Invariant

```
-- balancing heads in an arbitrary loop
pred revise_flows [s, s": State] {
    some i: Int | inc_flow[Cycle.succ, i, s, s"]
}

-- loop invariant: continuity of flow
pred inv [s: State] {
    all u: Node | net_flow[u, s] = 0
}

-- check and see if the invariant holds
assert invariant_holds {
  let s = so/first, s" = s.next |
    inv[s] and revise_flows[s, s"] implies inv[s"]
}
```

Results of the Exercise

An object model and associated heap invariants that allow us to conclude that continuity of flow is assured.

Results of the Exercise

An object model and associated heap invariants that allow us to conclude that continuity of flow is assured.

Though a simple problem, it demonstrates that

a) Much can be done without a general theory of reals

   – integers are a ring, and that’s all we need (no need for multiplicative inverses here)

Results of the Exercise

An object model and associated heap invariants that allow us to conclude that continuity of flow is assured.

Though a simple problem, it demonstrates that

a) Much can be done without a general theory of reals

  – integers are a ring, and that's all we need (no need for multiplicative inverses here)

b) Boundaries of declarative models can easily grow and shrink

  – we specify *what* loops are, but could just as easily define *how* to construct them (and check the construction process)

Conclusion

Lightweight formal methods

Are they useful?

Conclusion

Lightweight formal methods

Are they useful?

(all tools are useful, in some context, or they're not tools)

Conclusion

Lightweight formal methods

Are they useful?

(all tools are useful, in some context, or they're not tools)

Okay, are they useful *enough*?

Versatile, so proficiency can be developed and maintained.

Conclusion

Lightweight formal methods

Are they useful?

(all tools are useful, in some context, or they're not tools)

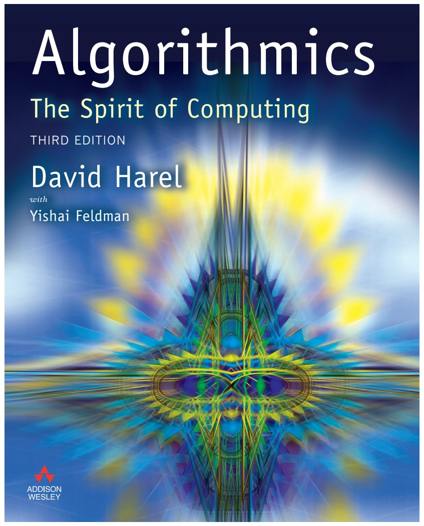
Okay, are they useful *enough*?

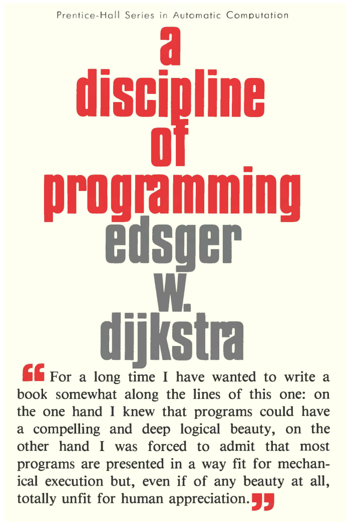
Versatile, so proficiency can be developed and maintained.

Complementary to historically useful strategies:

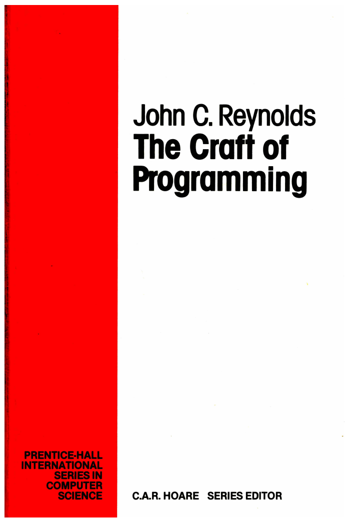
Abstraction, separation of concerns, engineering tools, and progressive codification

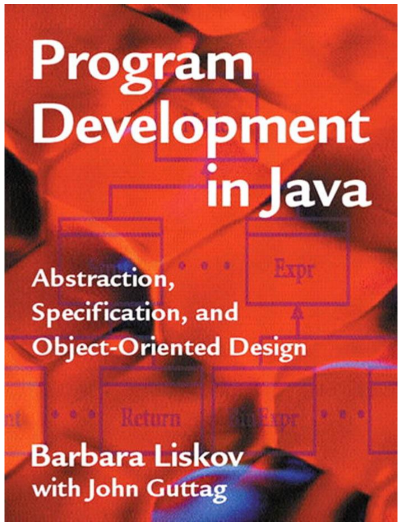
Some Books Informing This Perspective ... Thank You!

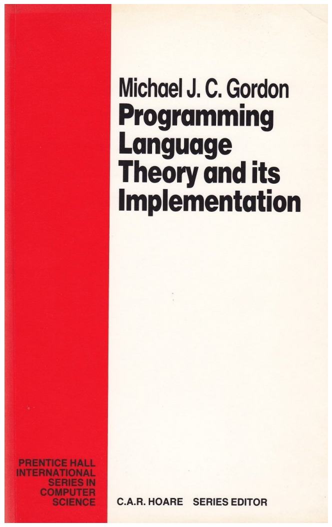

Algorithmics
The Spirit of Computing
THIRD EDITION
David Harel
with
Yishai Feldman
ADDISON WESLEY

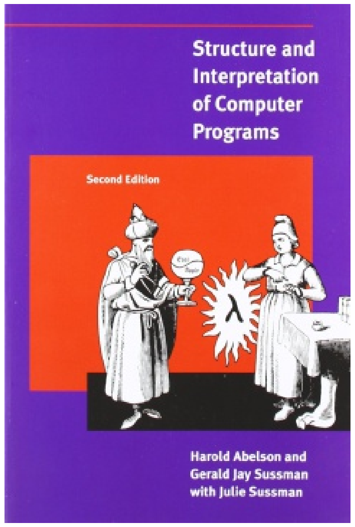

Prentice-Hall Series in Automatic Computation
a discipline of programming
edsger w. dijkstra

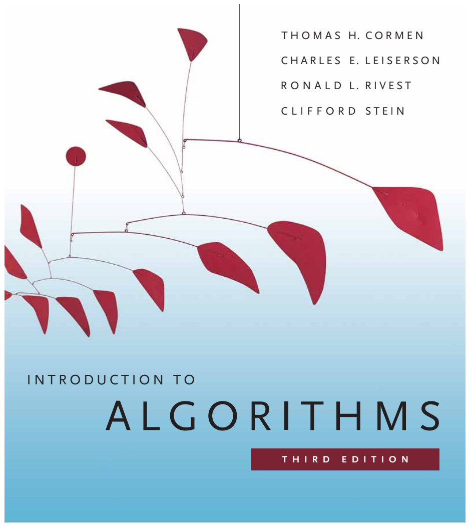
"For a long time I have wanted to write a book somewhat along the lines of this one: on the one hand I knew that programs could have a compelling and deep logical beauty, on the other hand I was forced to admit that most programs are presented in a way fit for mechanical execution but, even if of any beauty at all, totally unfit for human appreciation."

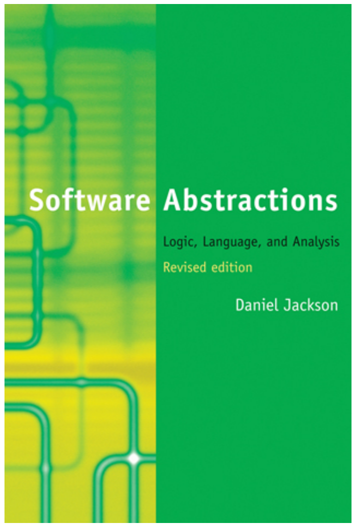

John C. Reynolds
The Craft of Programming
PRENTICE-HALL INTERNATIONAL SERIES IN COMPUTER SCIENCE
C.A.R. HOARE SERIES EDITOR


Program Development in Java
Abstraction, Specification, and Object-Oriented Design
Barbara Liskov
with John Guttag


Michael J. C. Gordon
Programming Language Theory and its Implementation
PRENTICE HALL INTERNATIONAL SERIES IN COMPUTER SCIENCE
C.A.R. HOARE SERIES EDITOR

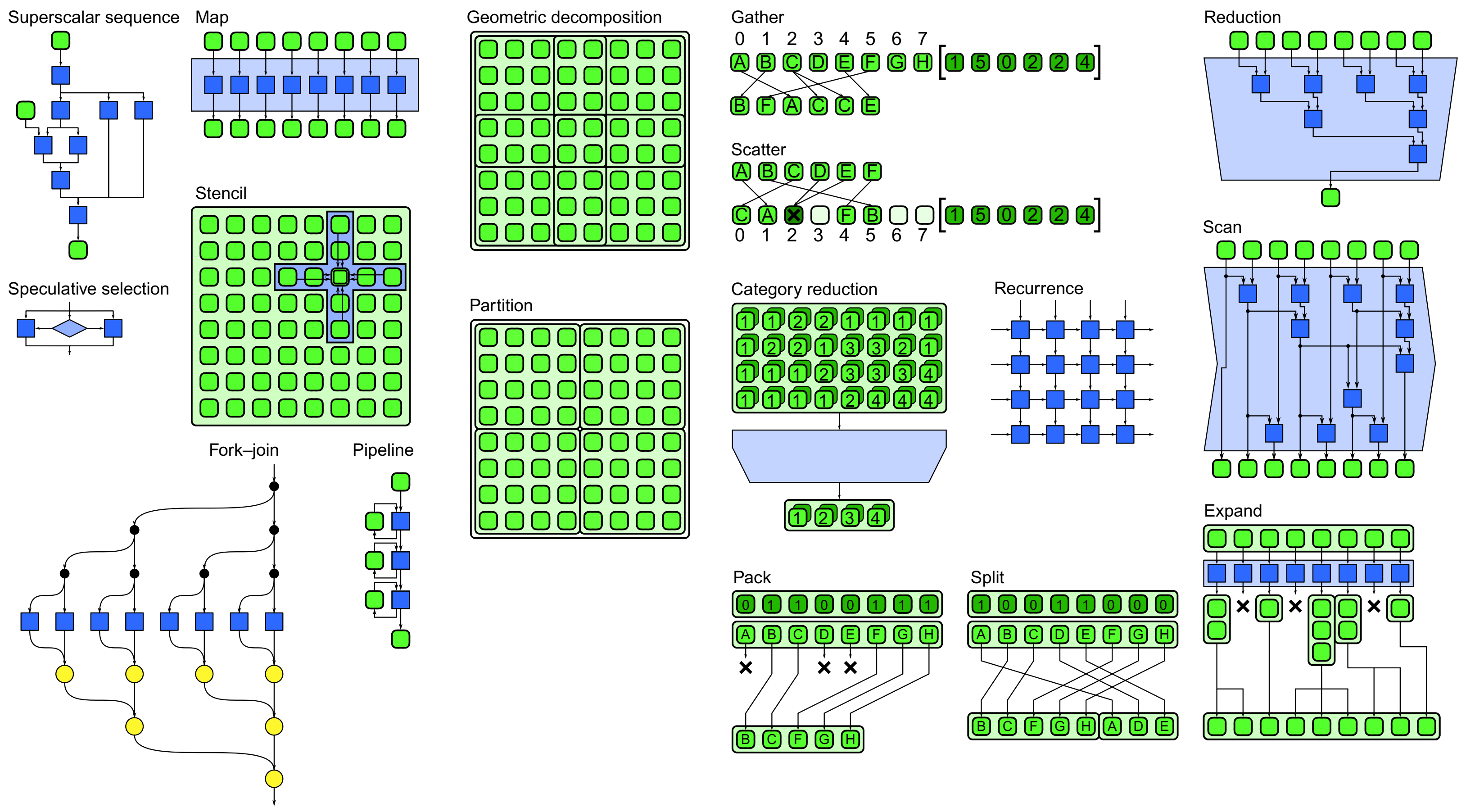

Structure and Interpretation of Computer Programs
Second Edition
Harold Abelson and
Gerald Jay Sussman
with Julie Sussman


THOMAS H. CORMEN
CHARLES E. LEISERSON
RONALD L. RIVEST
CLIFFORD STEIN
INTRODUCTION TO ALGORITHMS
THIRD EDITION


Software Abstractions
Logic, Language, and Analysis
Revised edition
Daniel Jackson

Thank You

Scientific Computing



McCool, M., Reinders, J., & Robison, A. (2012). *Structured parallel programming: patterns for efficient computation*. Elsevier.

Dijkstra's Perspective

The role of specifications: they act as a logical firewall between two different concerns.

- The *pleasantness* problem: the question of whether an engine meeting the specification is the engine we would like to have.

- The *correctness* problem: the question of how to design an engine meeting the specification.

See EWD 1058:

`https://www.cs.utexas.edu/users/EWD/transcriptions/EWD10xx/EWD1058.html`

Dijkstra's Perspective

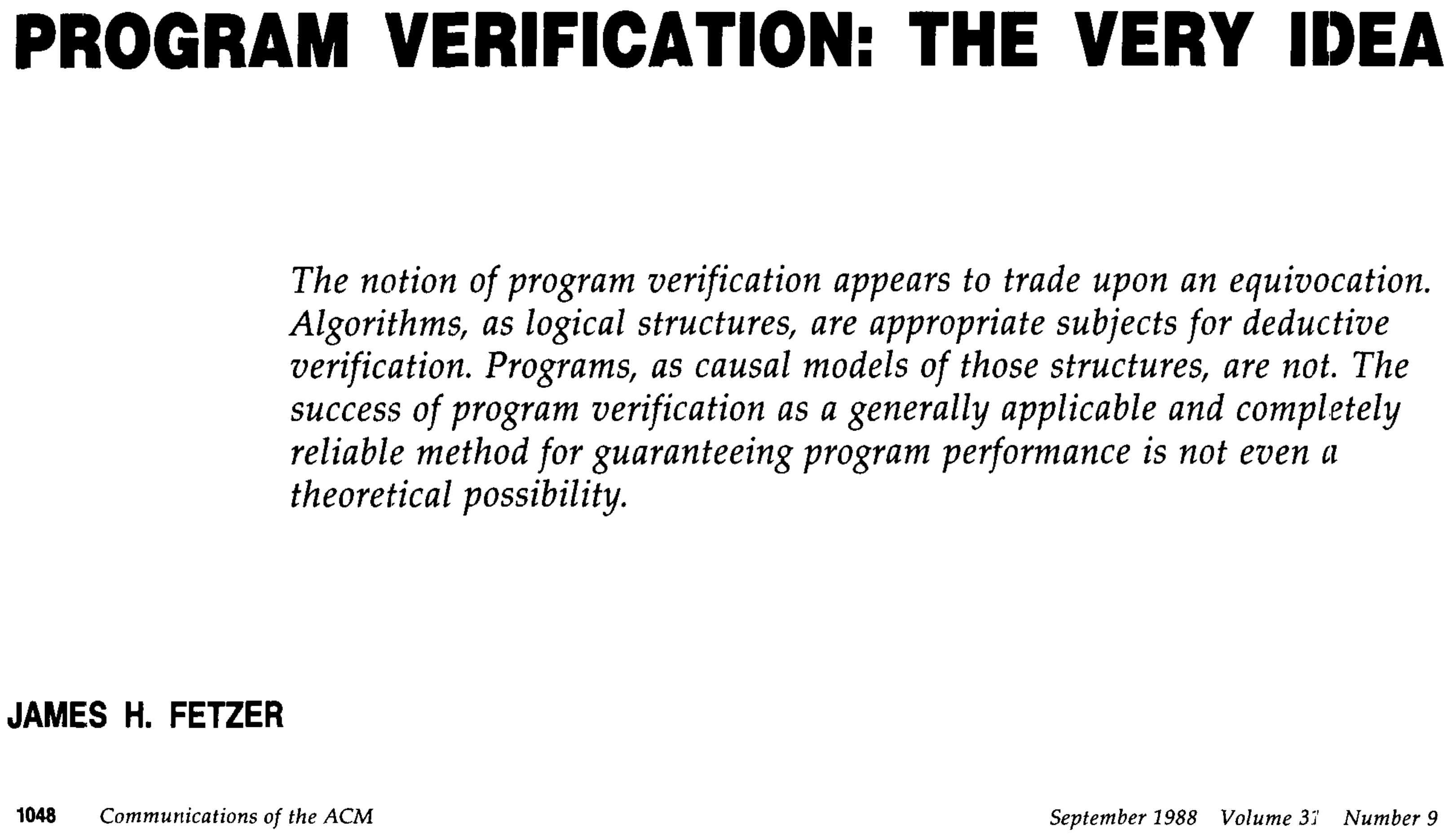
The role of specifications: they act as a logical firewall between two different concerns.

- The *pleasantness* problem: the question of whether an engine meeting the specification is the engine we would like to have.
  - "*Are we building the right product?*" (Boehm, 1979)
- The *correctness* problem: the question of how to design an engine meeting the specification.
  - "*Are we building the product right?*"

See EWD 1058:

`https://www.cs.utexas.edu/users/EWD/transcriptions/EWD10xx/EWD1058.html`

Polemics and Detractors



PROGRAM VERIFICATION: THE VERY IDEA

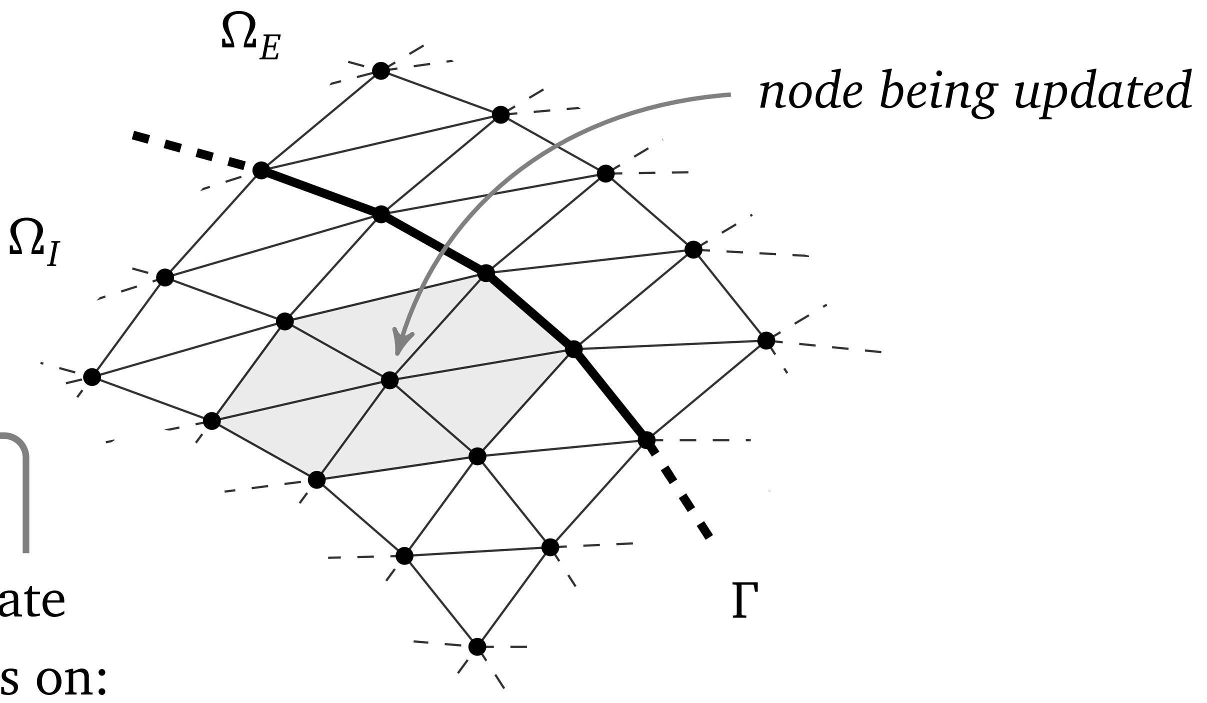
The notion of program verification appears to trade upon an equivocation. Algorithms, as logical structures, are appropriate subjects for deductive verification. Programs, as causal models of those structures, are not. The success of program verification as a generally applicable and completely reliable method for guaranteeing program performance is not even a theoretical possibility.

JAMES H. FETZER

Floating Point in the Larch Prover

```
FloatingPoint (smallest, largest,
               gap, rational): trait
  assumes FPAssumptions
  includes
    Rational,
    TotalOrder (F)
  introduces
    mag: F → Q
    approx: F, Q, Q → Bool
    -__, abs, __⁻¹: F → F
    __+__, __*__, __-__, __/__: F, F → F
  asserts
    F generated by float
    ∀ f, f1, f2: F, q, t: Q
      f1 ≤ f2 == rational(f1) ≤ rational(f2);
      mag(f) == abs(rational(f));
      approx(f, q, t) ==
        abs(q) ≤ largest
          ⇒ abs(rational(f) - q)
              ≤ (smallest +
                 (gap*(mag(f) + abs(q) + t)));
```

Wet-Dry States



element state depends on:

- incident nodes

node state depends on:

- water surface elevation
- incident elements and adjacent nodes

Physical Properties of Elements

```
0: for $e$ in $elements$ do
       $wet_e \leftarrow$ true
1: for $n$ in $nodes$ do
       if $W_n$ and $H_n < H_{min}$ then
           $W_n \leftarrow$ false, $W_n^t \leftarrow$ false
2: for $e$ in $elements$ do
       if $\neg W_i$ for one node $i$ on $e$ and $V_{ss}(e) > V_{min}$ then
           $W_i^t \leftarrow$ true
3: for $e$ in $elements$ do
       find nodes $i$ and $j$ of $e$ with highest water surface
       if $\min(H_i, H_j) < 1.2H_{min}$ then
           $wet_e \leftarrow$ false
4: for $n$ in $nodes$ do
       if $W_n^t$ and $n$ on only inactive elements then
           $W_n^t \leftarrow$ false
5: for $n$ in $nodes$ do
       $W_n \leftarrow W_n^t$
```

```
sig Element extends Triangle {
    wet: Bool one → State
}

sig State {}
```

accommodate state changes

Physical Properties of Elements

```
0: for $e$ in $elements$ do
       $wet_e \leftarrow$ true
1: for $n$ in $nodes$ do
       if $W_n$ and $H_n < H_{min}$ then
           $W_n \leftarrow$ false, $W_n^t \leftarrow$ false
2: for $e$ in $elements$ do
       if $\neg W_i$ for one node $i$ on $e$ and $V_{ss}(e) > V_{min}$ then
           $W_i^t \leftarrow$ true
3: for $e$ in $elements$ do
       find nodes $i$ and $j$ of $e$ with highest water surface
       if $\min(H_i, H_j) < 1.2H_{min}$ then
           $wet_e \leftarrow$ false
4: for $n$ in $nodes$ do
       if $W_n^t$ and $n$ on only inactive elements then
           $W_n^t \leftarrow$ false
5: for $n$ in $nodes$ do
       $W_n \leftarrow W_n^t$
```

```
sig Element extends Triangle {
    wet: Bool one → State,
    slowFlow: one Bool
    lowNode: Node
}
```

Physical Properties of Nodes

```
0: for $e$ in $elements$ do
       $wet_e \leftarrow$ true
1: for $n$ in $nodes$ do
       if $W_n$ and $H_n < H_{min}$ then
           $W_n \leftarrow$ false, $W_n^t \leftarrow$ false
2: for $e$ in $elements$ do
       if $\neg W_i$ for one node $i$ on $e$ ... then
           $W_i^t \leftarrow$ true
3: for $e$ in $elements$ do
       find nodes $i$ and $j$ of $e$ with highest water surface
       if $\min(H_i, H_j) < 1.2 H_{min}$ then
           $wet_e \leftarrow$ false
4: for $n$ in $nodes$ do
       if $W_n^t$ and $n$ on only inactive elements then
           $W_n^t \leftarrow$ false
5: for $n$ in $nodes$ do
       $W_n \leftarrow W_n^t$
```

```
sig Node extends Vertex {
    W, Wt: Bool one → State,
    H: Height
}

abstract sig Height {}

one sig Low, Med, High
    extends Height {}
```

Dynamics: Wetting and Drying

- Model each part of the algorithm by a predicate defining the state change
- Form a trace by chaining together parts and thereby constraining intermediate states
- Consider only a single time step:
  - begin with arbitrary wet-dry states, as though they had been produced in a prior time step
  - check correctness condition at the end of one step

Dynamics: Wetting and Drying

```
-- nodal wetting (propagate wetness across triangle if flow is not slow)
pred part2 [m: Mesh, s, s': State] {
  noElementChange[m, s, s']
  all n: m.nodes | n.W.s' = n.W.s
    and n.Wt.s' = (make_wet[m, n, s] implies True else n.Wt.s) }

-- define the conditions that cause a node to become wet
pred make_wet [m: Mesh, n: Node, s: State] {
  some e: m.elements | e.slowFlow = False and loneDryNode[n, e, s] }

pred loneDryNode [n: Node, e: Element, s: State] {
  n in dom[e.edges] and n.W.s = False and wetNodes[e, s] = 2 }

fun wetNodes [e: Element, s: State]: Int {
  #(dom[e.edges] <: W).s.True }
```

Results of the Exercise

Final wet-dry states from boundary nodes in a full run can be used to impose *intermediate* wet-dry states in subdomain runs

- No need to record intermediate wet-dry states of nodes and elements

Results of the Exercise

Final wet-dry states from boundary nodes in a full run can be used to impose *intermediate* wet-dry states in subdomain runs

- No need to record intermediate wet-dry states of nodes and elements

Thus, for actual simulations in ADCIRC, we can record a minimal amount of data and impose it as boundary conditions

- This is in fact how subdomain modeling is implemented in ADCIRC beginning with v51.42

Tabular Pattern for Nested, Bounded Iteration

Pattern defines an iteration table (iter) and time-indexed scalar variables (x, y), where ψ and ω define loop bounds:

```
some iter: Int→ Int→ Int, $x$, $y$, ...: Int→ univ {
   table[{i: $\psi$, j: $\omega$ | ...}, iter]
   all i: $\psi$ |
      all j: $\omega$ |
         let t = iter[i][j], t' = t.add[1] {
            $x$[t'] = ... $x$[t] ...
            $y$[t'] = ... $y$[t] ...
            ...
         }
}
```

Pseudo-code:

```
for $i$ in $\psi$ do
   for $j$ in $\omega$ do

      $x = \ldots x \ldots$
      $y = \ldots y \ldots$
      ...
```

Dense and CSR Transpose Fragments

```
pred transpose [m, m': Matrix] {
  m'.rows = m.cols
  m'.cols = m.rows
  m'.vals = { j, i: Int, v: Value | i → j → v in m.vals }
}
```

```
pred transpose [c, c': CSR] {
  ...
  all i: range[c.rows] |
    all k: range[c.IA[i], c.IA[i.add[1]]] |
      let t = iter[i][k], t' = t.add[1],
          j = c.JA[k],
          nxt = c'.IA[t][j] {
        c'.A[nxt] = c.A[k]
        c'.JA[nxt] = i
        c'.IA[t'] = c'.IA[t] ++ j → nxt.add[1]
      }
```

Pseudo-code:

```
for $i$ in $range(c.rows)$ do
    for $k$ in $range(\text{c.IA}[i], \text{c.IA}[i+1])$ do

        $j \leftarrow \text{c.JA}[k]$
        $nxt \leftarrow \text{c'.IA}[j]$
        $\text{c'.A}[nxt] \leftarrow \text{c.A}[k]$
        $\text{c'.JA}[nxt] \leftarrow i$
        $\text{c'.IA}[j] \leftarrow nxt + 1$
```

Loss of Head

We assume that we know the law determining the loss of head in any length of pipe for a given flow. This law usually takes the form

$$h = CV^n$$

where h is the change in head accompanying flow in any length of pipe, C is the loss in the pipe for unit velocity of flow, and V is the velocity. Since the quantity of water flowing in the pipe is AV, this relation may be rewritten

$$h = rQ^n$$

where r is the loss of head in the pipe for unit quantity of flow. The quantity r depends on the length and diameter of pipe and on its roughness.

The problem is to find the amount of water flowing in each pipe.

Further Details from the Paper

b) Compute in each pipe the loss of head $h = rQ^n$.

With due attention to ***sign*** (direction of potential drop), compute the total head loss around each elementary closed circuit

$$\sum h = \sum rQ^n$$

(c) Compute also in each such closed circuit the sum of the quantities $R = nrQ^{(n-1)}$ *without reference to sign*.

(d) Set up in each circuit a counterbalancing flow to balance the head in that circuit (to make $\sum rQ^n = 0$) equal to

$$\Delta = \frac{\sum rQ^n \ \ \text{(with due attention to direction of flow)}}{\sum nrQ^{(n-1)} \ \ \text{(without reference to direction of flow)}}$$

(e) Compute the revised flows and repeat the procedure.