Verification and Developing a Testing Regime

Presented to
ATPESC 2018 Participants

Anshu Dubey

Computer Scientist, Mathematics and Computer Science Division

Q Center, St. Charles, IL (USA)
Date 08/08/2018

EXASCALE COMPUTING PROJECT

NNSA National Nuclear Security Administration

U.S. DEPARTMENT OF ENERGY | Office of Science

Argonne NATIONAL LABORATORY

License, citation, and acknowledgments

License and Citation

- Requested citation: Anshu Dubey, Verification and developing a testing regime, tutorial, in Argonne Training Program on Extreme-Scale Computing (ATPESC) 2018. DOI: 10.6084/m9.figshare.6943091

Acknowledgements

- This work was supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research (ASCR), and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration..
- This work was performed in part at the Argonne National Laboratory, which is managed managed by UChicago Argonne, LLC for the U.S. Department of Energy under Contract No. DE-AC02-06CH11357

Argonne NATIONAL LABORATORY

ECP EXASCALE COMPUTING PROJECT

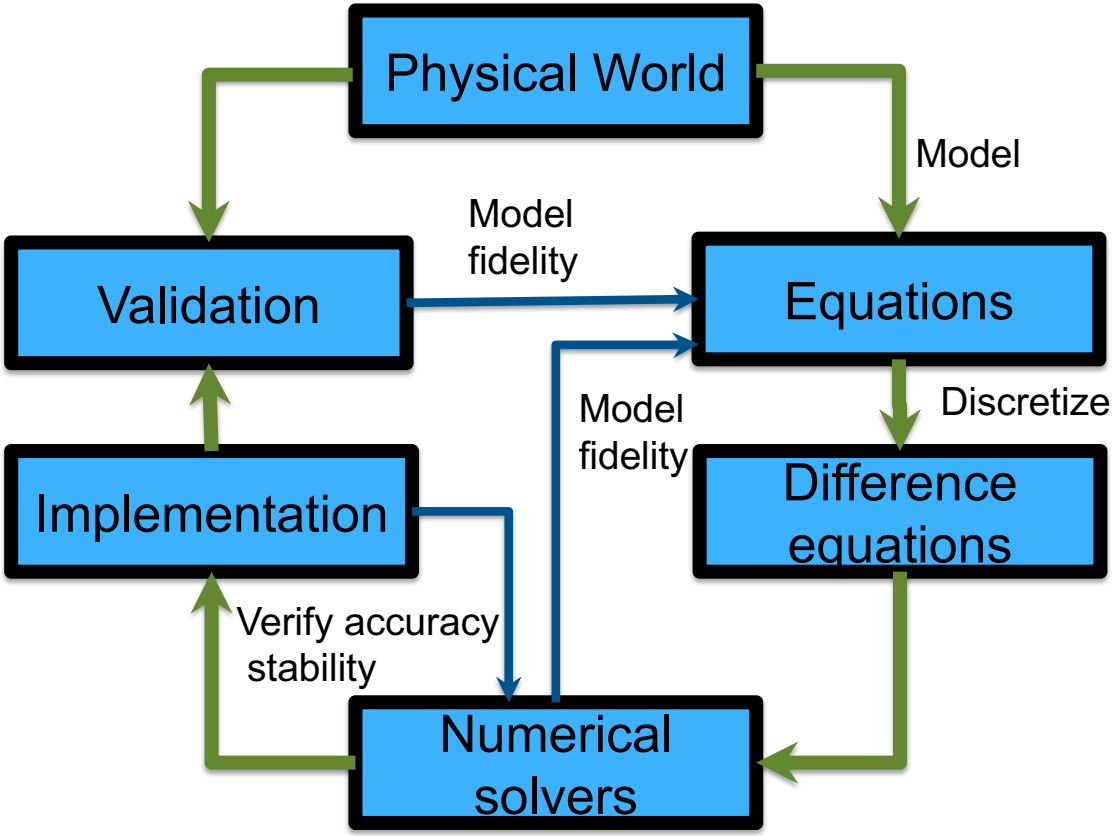
Verification

- Code verification uses tests
  - It is much more than a collection of tests
- It is the holistic process through which you ensure that
  - Your implementation shows expected behavior,
  - Your implementation is consistent with your model,
  - Science you are trying to do with the code can be done.

Challenge with Exploratory Software

- Verification implies one knows the outcome
  - The outcome is achieved or not achieved
- What if one doesn't exactly know the outcome?
  - Software is meant to understand the expected outcome

Challenge with Scientific Software



This is for simulations, but the philosophy applies to other computations too.

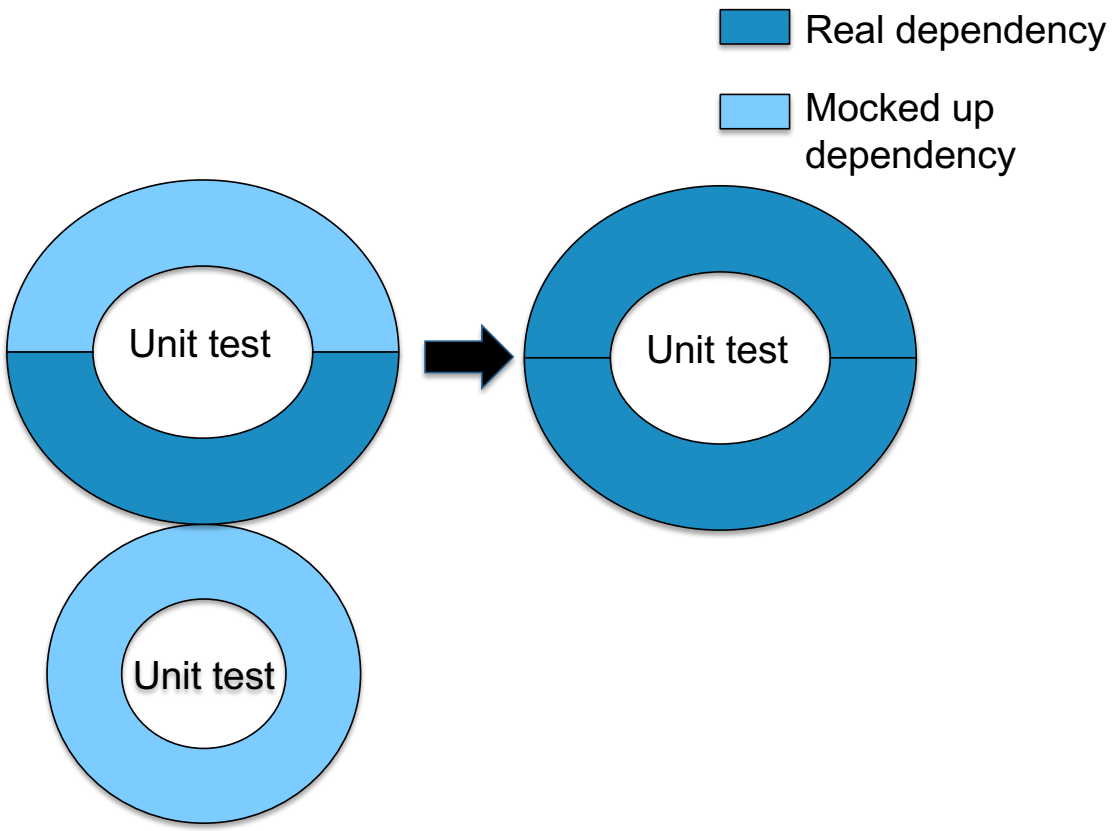
Many stages in the lifecycle have components that may themselves be under research => need modifications

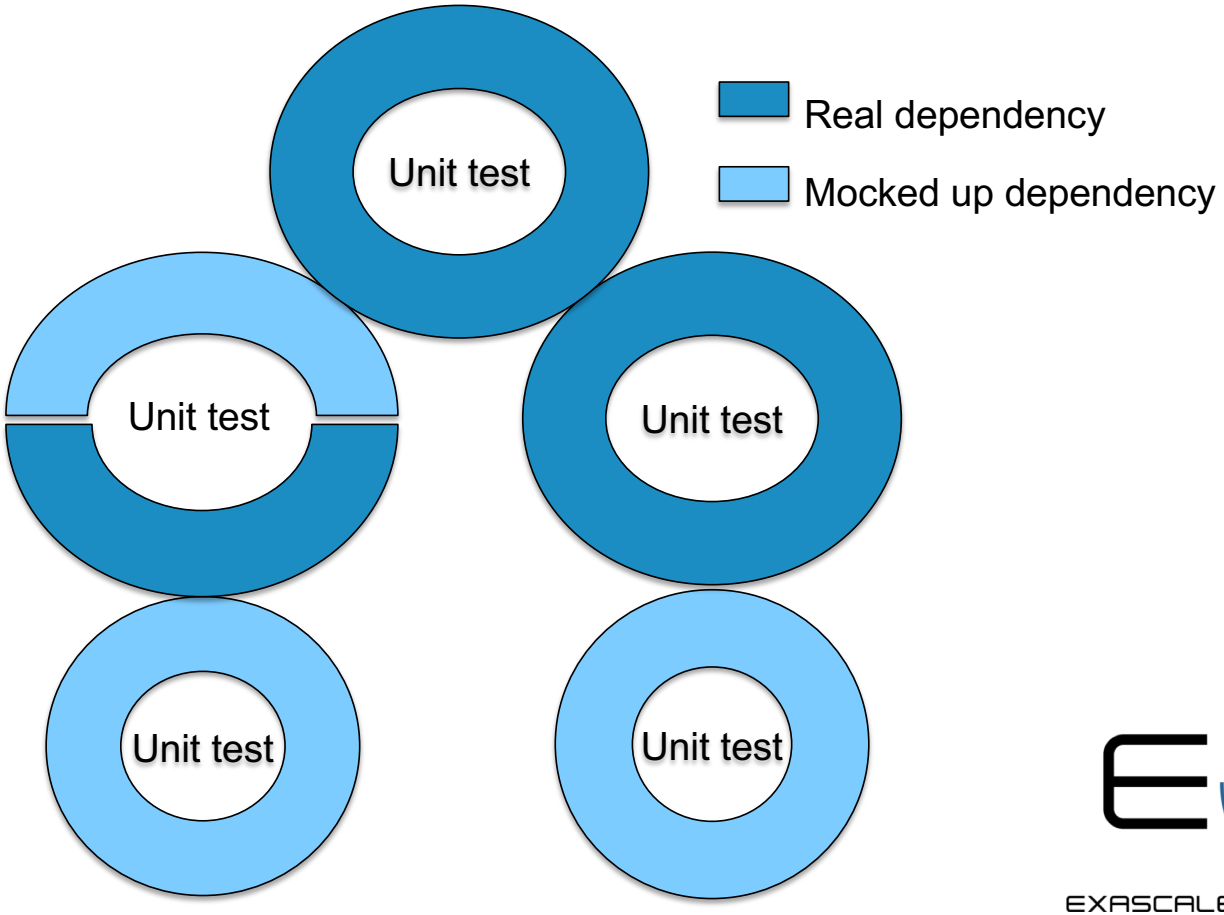
Other specific verification challenges

- Integration testing may have hierarchy too
- Particularly true of codes that allow composability in their configuration
- Codes may incorporate some legacy components
  - Its own set of challenges
    - No existing tests at any granularity
- Examples – multiphysics application codes that support multiple domains

Workarounds

- Approach the problem sideways
  - Components can be exercised against known simpler applications
  - Same applies to combination of components
- Build a scaffolding of verification tests to gain confidence



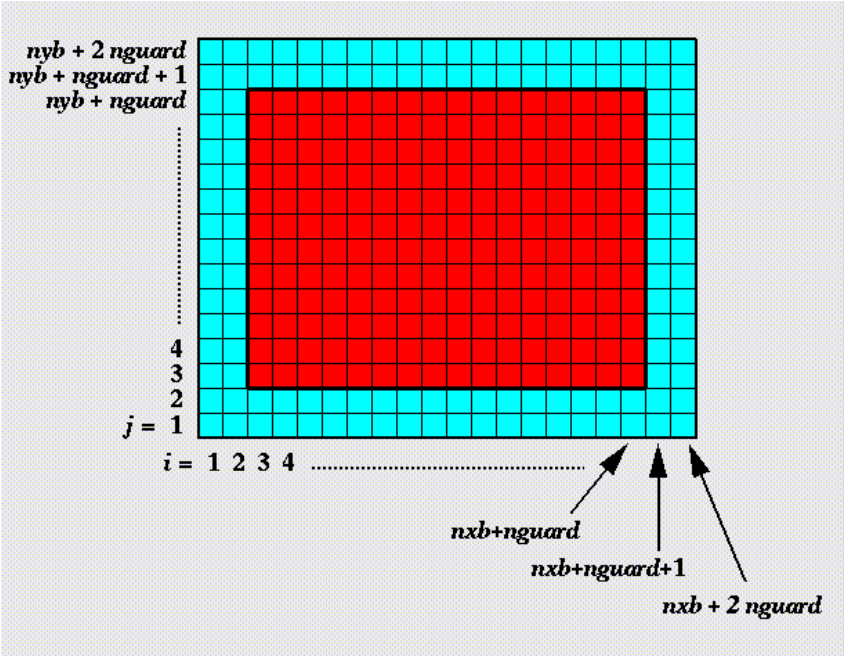

Real dependency
Mocked up dependency
Unit test
Unit test
Unit test
Unit test
Unit test

Test Development

- Development of tests and diagnostics goes hand-in-hand with code development
  - Non-trivial to devise good tests, but extremely important
  - Compare against simpler analytical or semi-analytical solutions
    - They can also form a basis for unit testing
- In addition to testing for “correct” behavior, also test for stability, convergence, or other such desirable characteristics

Example from Flash

- Against manufactured solution
- Grid ghost cell fill
  - Use a known analytical function to initialize domain
  - Use two variables A & B
  - Initialize A including guard cells and B excluding them
  - Apply guard cell fill to B



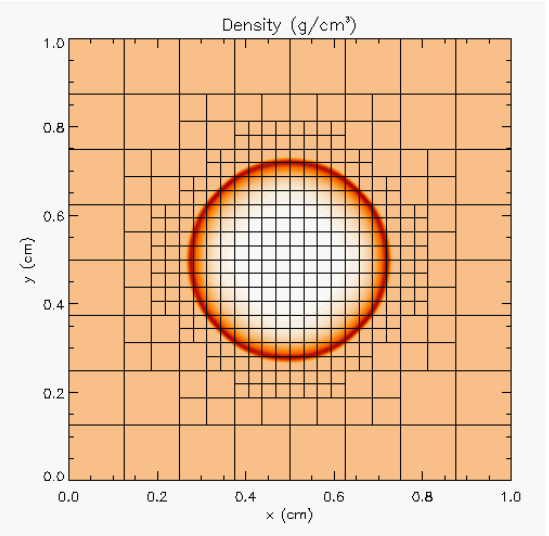
- Works for uniform and adaptive mesh

Example from Flash Equation of State

- Operates in three modes
  - Given density and internal energy get pressure
  - Given density and pressure get temperature
  - Given density of pressure get internal energy
- Use initial conditions from a known problem
- Apply EOS in two different modes – at the end all variables should be consistent within tolerance

Example from FLASH, scaffolding

- Sedov blast wave
- High pressure at the center
- Shock moves out spherically
- FLASH with AMR and hydro
- Known analytical solution



Though it exercises both mesh, hydro and eos, if mesh and eos are verified first, then this test verifies hydro

Building confidence

- First two unit tests are stand-alone
- The third test depends on Grid and Eos
  - Not all of Grid functionality it uses is unit tested
    - Flux correction in AMR
- If Grid and Eos tests passed and Hydro failed
  - If UG version failed then fault is in hydro
  - If UG passed and AMR failed the fault is likely in flux correction

Stages and types of verification

- During initial code development
  - Accuracy and stability
  - Matching the algorithm to the model
  - Interoperability of algorithms
- In later stages
  - While adding new major capabilities or modifying existing capabilities
  - Ongoing maintenance
  - Preparing for production

Stages and types of verification

- If refactoring
  - Ensuring that behavior remains consistent and expected
- All stages have a mix of automation and human-intervention

Note that the stages apply to the whole code as well as its components

Test Development

- Development of tests and diagnostics goes hand-in-hand with code development
  - Non-trivial to devise good tests, but extremely important
  - Compare against simpler analytical or semi-analytical solutions
    - They can also form a basis for unit testing
- In addition to testing for “correct” behavior, also test for stability, convergence, or other such desirable characteristics
- Many of these tests go into the test-suite

Use of test harnesses

- Essential for large code
  - Set up and run tests
  - Evaluate test results
- Easy to execute a logical subset of tests
  - Pre-push
  - Nightly
- Automation of test harness is critical for
  - Long-running test suites
  - Projects that support many platforms

Jenkins
C-dash
Custom
(FlashTest)

Policies on testing practices

- Must have consistent policy on dealing with failed tests
  - Issue tracking
    - How quickly does it need to be fixed?
    - Who is responsible for fixing it?
- Someone needs to be in charge of watching the test suite

Policies on testing practices

- When refactoring or adding new features, run a regression suite before checkin
  - Be sure to add new regression tests for the new features
- Require a code review before releasing test suite
  - Another person may spot issues you didn't
  - Incredibly cost-effective

Maintenance of a test suite

- Testing regime is only useful if it is
  - Maintained
    - Tests and benchmarks periodically updated
  - Monitored regularly
    - Can be automated
  - Has rapid response to failure
    - Tests should pass most of the time

How to evaluate project needs

And devise a testing regime

Why not always use the most stringent testing?

- Effort spent in devising tests and testing regime are a tax on team resources
- When the tax is too high…
  - Team cannot meet code-use objectives
- When is the tax is too low…
  - Necessary oversight not provided
  - Defects in code sneak through

Evaluating project needs

- Objectives: expected use of the code
- Team: size and degree of heterogeneity
- Lifecycle stage: new or production or refactoring
- Lifetime: one off or ongoing production
- Complexity: modules and their interactions

Commonalities

- Unit testing is always good
  - It is never sufficient
- Verification of expected behavior
- Understanding the range of validity and applicability is always important
  - Especially for individual solvers

Challenges with legacy codes

Checking for coverage

- Legacy codes can have many gotchas
  - Dead code
  - Redundant branches
- Interactions between sections of the code may be unknown
- Can be difficult to differentiate between just bad code, or bad code for a good reason
  - Nested conditionals

Code coverage tools are of limited help

An Approach

- Isolate a small area of the code
- Dump a useful state snapshot
- Build a test driver
  - Start with only the files in the area
  - Link in dependencies
    - Copy if any customizations needed
- Read in the state snapshot
- Verify correctness
  - Always inject errors to verify that the test is working

Methodology developed for the E3SM project, proving to be very useful

Selection of tests

- Two purposes
  - Regression testing
    - May be long running
    - Provide comprehensive coverage
  - Continuous integration
    - Quick diagnosis of error
- A mix of different granularities works well
  - Unit tests for isolating component or sub-component level faults
  - Integration tests with simple to complex configuration and system level
  - Restart tests
- Rules of thumb
  - Simple
  - Enable quick pin-pointing

Approach for Test Selection

- Build a matrix
  - Physics along rows
  - Infrastructure along columns
  - Alternative implementations, dimensions, geometry
- Mark <i,j> if test covers corresponding features
- Follow the order
  - All unit tests – including full module tests
  - Tests representing ongoing productions
  - Tests sensitive to perturbations
  - Most stringent tests for solvers
  - Least complex test to cover remaining spots

Example

	Hydro	EOS	Gravity	Burn	Particles
AMR	CL	CL		CL	CL
UG	SV	SV			SV
Multigrid	WD	WD	WD	WD	
FFT			PT		

Tests	Symbol
Sedov	SV
Cellular	CL
Poisson	PT
White Dwarf	WD

- A test on the same row indicates interoperability between corresponding physics
- Similar logic would apply to tests on the same column for infrastructure
- More goes on, but this is the primary methodology

Questions

Benefits of testing

- Promotes high-quality software that delivers correct results and improves confidence
- Increases quality and speed of development, reducing development and maintenance costs
- Maintains portability to a variety of systems and compilers
- Helps in refactoring
  - Avoid introducing new errors when adding new features
  - Avoid reintroducing old errors

How common are bugs?

Programs do not acquire bugs as people acquire germs, by hanging around other buggy programs. Programmers must insert them.
- Harlan Mills

- Bugs per 1000 lines of code (KLOC)
- Industry average for delivered software
  - 1-25 errors
- Microsoft Applications Division
  - 10-20 defects during in-house testing
  - 0.5 in released product

Code Complete (Steven McConnell)

Why testing is important: the protein structures of Geoffrey Chang

- Some inherited code flipped two columns of data, inverting an electron-density map
- Resulted in an incorrect protein structure
- Retracted 5 publications
  - One was cited 364 times
- Many papers and grant applications conflicting with his results were rejected

He found and reported the error himself

Why testing is important: the 40 second flight of the Ariane 5

- Ariane 5: a European orbital launch vehicle meant to lift 20 tons into low Earth orbit
- Initial rocket went off course, started to disintegrate, then self-destructed less than a minute after launch
- Seven variables were at risk of leading to an Operand Error (due to conversion of floating point to integer)
  - Four were protected
- Investigation concluded insufficient test coverage as one of the causes for this accident
- Resulted in a loss of $370,000,000.

Why testing is important: the Therac-25 accidents

- Therac-25: a computer-controlled radiation therapy machine
- Minimal software testing
- Race condition in the code went undetected
- Unlucky patients were struck with approximately 100 times the intended dose of radiation, ~ 15,000 rads
- Error code indicated that no dose of radiation was given, so operator instructed machine to proceed
- Recalled after six accidents resulting in death and serious injuries