Compatibility Testing

- Testing compatibility of components of a software
- Ensures that software will work (build/execute) with different version of the components

- Motivation:
  - Automated techniques unavailable
  - Large number of configurations make manual testing impossible
Solution

- Rachet system software
- Evaluation
- Future work
- Discussion

Rachet

1. Model configuration space
2. Determine coverage criteria
3. Produce test configuration and test plan
4. Execute test plan
   - Testing only that software builds
Model Configuration Space

- Model components of a software and their relationships to other components
- Model versions of each component
- Model relationships between versions of components

Model Configuration Space

[Diagram showing component annotations and constraints:]

Version Annotations

<table>
<thead>
<tr>
<th>Component</th>
<th>Versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>B₁, B₂, B₃</td>
</tr>
<tr>
<td>C</td>
<td>C₁, C₂</td>
</tr>
<tr>
<td>D</td>
<td>D₁, D₂, D₃</td>
</tr>
<tr>
<td>E</td>
<td>E₁, E₂, E₃, E₄</td>
</tr>
<tr>
<td>F</td>
<td>F₁, F₂, F₃, F₄, F₅</td>
</tr>
<tr>
<td>G</td>
<td>G₁</td>
</tr>
</tbody>
</table>

Constraints:

$(\text{ver}(C) = C₂) \rightarrow (\text{ver}(E) \geq E₁)$
Determine Coverage Criteria

- Exhaustive (EX)
  - Generate configurations exhaustively

- Configurations for building A:
  - \{B_1, C_1\}
  - \{B_2, C_1\}
  - \{B_1, C_2\}
  - \{B_2, C_2\}

Determine Coverage Criteria

- Direct Depends: “A component directly depends on another component if there is a path between the two components such that there isn’t any other component on the path”

- Directly Depends (DD)
  - Cover all direct dependencies by at least one configuration

- Configurations for building A:
  - \{B_1, C_1\}
  - \{B_2, C_2\}
Filter configurations

- Observation: “Multiple configurations might contain identical direct dependency sequence”
- Put configurations into a prefix tree (test plan)
- This reduces number of components to build
- For each configuration, nodes representing direct dependencies are added in order to the tree

Prefix Tree (Test Plan)
Rachet Test Execution Architecture

- Client/Server design
- Server controls the plan execution and distributes build tasks to clients
- Client connects to the server to ask for a single task and runs the given task
- Client has a cache which can be used to store dependencies
- VMWare is used to run the tests
- Server can send initial states to clients in the form VMWare files

Strategies for running test plan

- Need to cover all nodes in the test plan
- Parallel Depth-First:
  - Utilizes locally cached tasks
  - Find next task by doing a depth first search
- Parallel Breadth-First:
  - Aims to maximize number of tasks being executed simultaneously
  - Secondarily maximize use of cache
- Hybrid approach
  - Designed to maximize parallelism and reusability of cache
Test Plan

Evaluation

- **Research Questions:**
  - How does exhaustive coverage compare to direct dependency coverage
  - Loss of effectiveness with direct dependency coverage
  - Which test execution strategy is better

- **Test Ratchet with 2 subject applications**
  - InterComm
  - PETSc
CDG for test subjects

Component Versions

<table>
<thead>
<tr>
<th>Comp.</th>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>petsc</td>
<td>2.2.0</td>
<td>PETSc, the SUT</td>
</tr>
<tr>
<td>fc</td>
<td>1.5</td>
<td>InterComm, the SUT</td>
</tr>
<tr>
<td>blas</td>
<td>2.3.6, 2.5.1</td>
<td>Dynamic OOP language</td>
</tr>
<tr>
<td>lapack</td>
<td>1.0</td>
<td>Basic linear algebra subprograms</td>
</tr>
<tr>
<td>ap</td>
<td>2.0, 3.1.1</td>
<td>A library for linear algebra operations</td>
</tr>
<tr>
<td>pvm</td>
<td>0.7.9</td>
<td>High-level array management library</td>
</tr>
<tr>
<td>lam</td>
<td>3.2.6, 3.3.11, 3.4.5</td>
<td>Parallel data communication component</td>
</tr>
<tr>
<td>mch</td>
<td>6.5.9, 7.0.6, 7.1.3</td>
<td>A library for MPI (Message Passing Interface) standard</td>
</tr>
<tr>
<td>gf</td>
<td>1.2.7</td>
<td>A library for MPI</td>
</tr>
<tr>
<td>g77</td>
<td>4.0.3, 4.1.1</td>
<td>GNU Fortran 95 compiler</td>
</tr>
<tr>
<td>pf</td>
<td>3.3.6, 3.4.6</td>
<td>GNU Fortran 77 compiler</td>
</tr>
<tr>
<td>gxx</td>
<td>6.2</td>
<td>PGI Fortran compiler</td>
</tr>
<tr>
<td>pxx</td>
<td>3.3.6, 3.4.6, 4.0.3, 4.1.1</td>
<td>GNU C++ compiler</td>
</tr>
<tr>
<td>mpfr</td>
<td>2.2.0</td>
<td>PGI C++ compiler</td>
</tr>
<tr>
<td>gmp</td>
<td>4.2.1</td>
<td>A library for multiple-precision floating-point number computations</td>
</tr>
<tr>
<td>pc</td>
<td>4.0</td>
<td>A library for arbitrary precision arithmetic computation</td>
</tr>
<tr>
<td>gcc</td>
<td>6.2</td>
<td>PGI C compiler</td>
</tr>
<tr>
<td>fc</td>
<td>3.3.6, 3.4.6, 4.0.3, 4.1.1</td>
<td>GNU C compiler</td>
</tr>
<tr>
<td>fc</td>
<td>4.0</td>
<td>Fedora Core Linux operating system</td>
</tr>
</tbody>
</table>
Test Plan Statistics

<table>
<thead>
<tr>
<th>System</th>
<th>Type</th>
<th>Cfgs</th>
<th>Comp_{cfgs}</th>
<th>Comp_{plan}</th>
</tr>
</thead>
<tbody>
<tr>
<td>InterComm</td>
<td>Ex-Cover</td>
<td>3552</td>
<td>39840</td>
<td>9919</td>
</tr>
<tr>
<td>InterComm</td>
<td>DD-Cover</td>
<td>158</td>
<td>1642</td>
<td>677</td>
</tr>
<tr>
<td>PETSc</td>
<td>Ex-Cover</td>
<td>1184</td>
<td>14336</td>
<td>3493</td>
</tr>
<tr>
<td>PETSc</td>
<td>DD-Cover</td>
<td>90</td>
<td>913</td>
<td>309</td>
</tr>
</tbody>
</table>

- For InterComm Ex-Plan (Total 9919):
  - Successful: 461
  - Failed: 687
- For InterComm DD-Plan (Total 677):
  - Successful: 275
- Large number of components could not be tested

Results

- DD is 2.5 - 3 times faster than EX
Loss of effectiveness

- Is there any loss of effectiveness with using direct dependency coverage?
- All failures in InterComm EX-plan maps to corresponding failure in related DD-plan
- Results show that 8 failures (how many failures overall??) from PETSc EX-plan were not detected by the related DD-plan
- Attributed to insufficient information in the model

Comparison of Test Plan Strategies
Rachet behavior as successful builds grow (simulated)

Related Work

- GridUnit/InGrid
- Skoll and BuildFarm
- Opium and EDOS
- No industry products in this domain
Future Work

- Further optimize plan execution strategies
- Explore new types of criteria
- Adding cost models into plan execution strategies
- Explore a way to extract dependences automatically from package tools like Automake

Discussion

- Only good as the model
- Requires a project that uses a build tool
- Comparison of EX-Plan with DD-Plan
- Rachet can be applied to other languages