Effective and Scalable Software Compatibility Testing

Il-Chul Yoon, Alan Sussman,

Atif Memon, Adam Porter

Tandeep Sidhu
CMSC 737 - Student Presentation

1

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

2

Solution

- Rachet system software
- Evaluation
- Future work
- Discussion

3

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

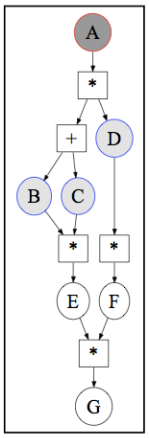
4

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

5

Model Configuration Space



Version Annotations

Component	Versions
A	A_1
B	B_1, B_2, B_3
C	C_1, C_2
D	D_1, D_2, D_3
E	E_1, E_2, E_3, E_4
F	F_1, F_2, F_3, F_4, F_5
G	G_1

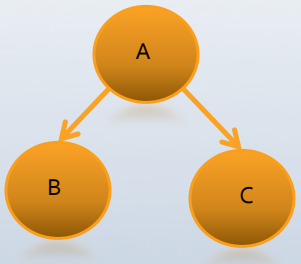
Constraints

$(ver(C) = C_2) \rightarrow (ver(E) \geq E_3)$

6

Determine Coverage Criteria

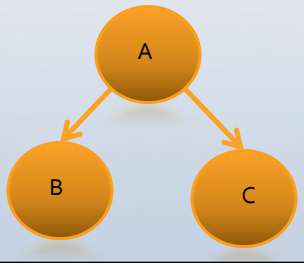
- Exhaustive (EX)
  - Generate configurations exhaustively
- Configurations for building A:
  - {B1,C1}
  - {B2,C1}
  - {B1,C2}
  - {B2,C2}



7

Determine Coverage Criteria

- Direct Depends: "A component directly depends on another component if there is path between the 2 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:
  - {B1,C1}
  - {B2,C2}



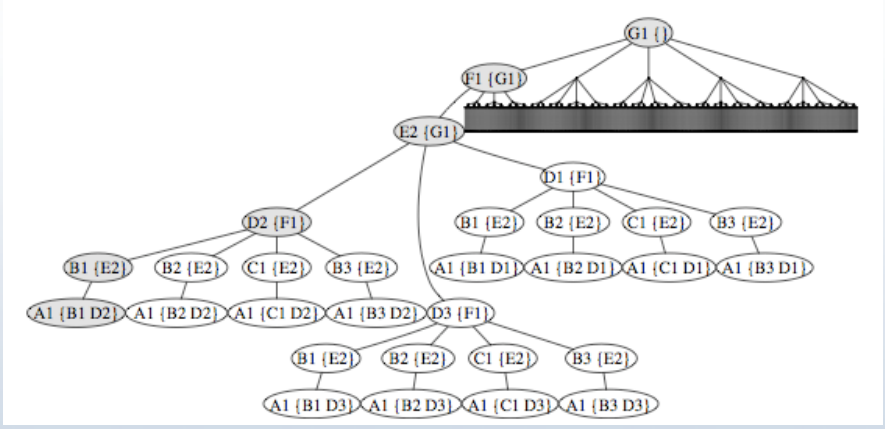
8

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

9

Prefix Tree (Test Plan)



10

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

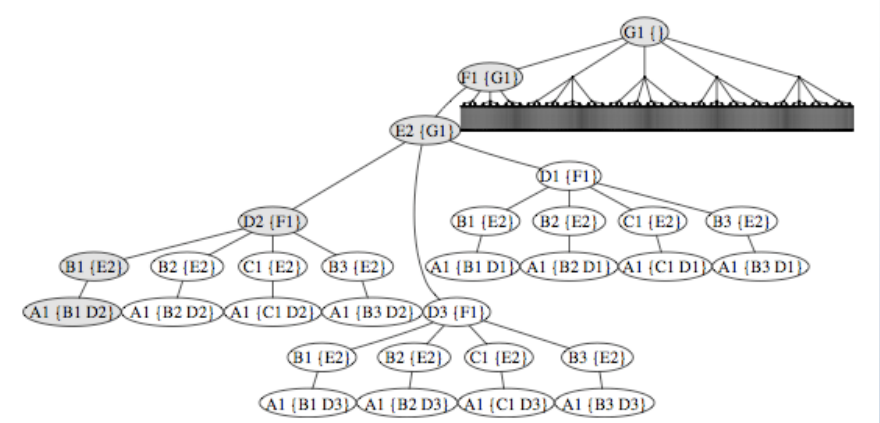
11

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

12

Test Plan



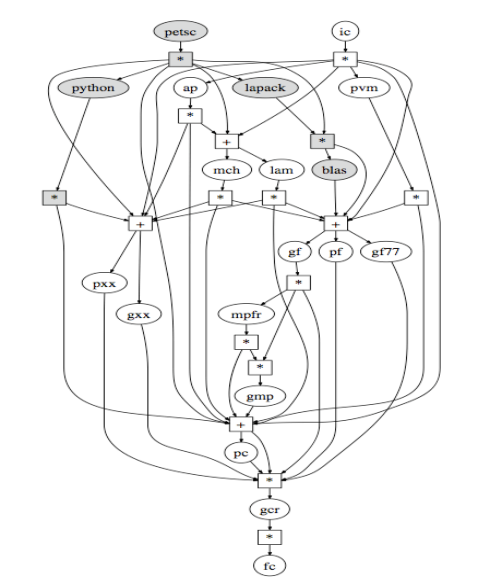
13

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 Rachet with 2 subject applications
  - InterComm
  - PETSc

14

CDG for test subjects



15

Component Versions

Comp.	Version	Description
petsc	2.2.0	PETSc, the SUT
ic	1.5	InterComm, the SUT
python	2.3.6, 2.5.1	Dynamic OOP language
blas	1.0	Basic linear algebra subprograms
lapack	2.0, 3.1.1	A library for linear algebra operations
ap	0.7.9	High-level array management library
pvm	3.2.6, 3.3.11, 3.4.5	Parallel data communication component
lam	6.5.9, 7.0.6, 7.1.3	A library for MPI (Message Passing Interface) standard
mch	1.2.7	A library for MPI
gf	4.0.3, 4.1.1	GNU Fortran 95 compiler
gf77	3.3.6, 3.4.6	GNU Fortran 77 compiler
pf	6.2	PGI Fortran compiler
gxx	3.3.6, 3.4.6, 4.0.3, 4.1.1	GNU C++ compiler
pxx	6.2	PGI C++ compiler
mpfr	2.2.0	A C library for multiple-precision floating-point number computations
gmp	4.2.1	A library for arbitrary precision arithmetic computation
pc	6.2	PGI C compiler
gcr	3.3.6, 3.4.6, 4.0.3, 4.1.1	GNU C compiler
fc	4.0	Fedora Core Linux operating system

16

Test Plan Statistics

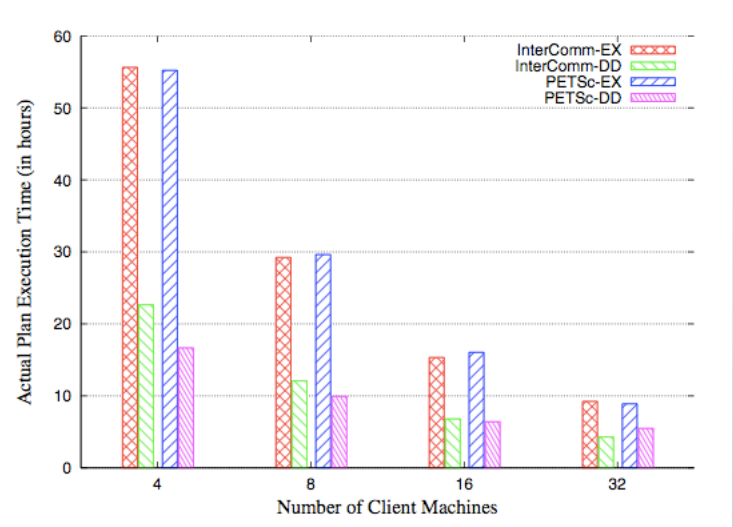
System	Type	Cfgs	Comp_{cfgs}	Comp_{plan}
InterComm	Ex-Cover	3552	39840	9919
InterComm	DD-Cover	158	1642	677
PETSc	Ex-Cover	1184	14336	3493
PETSc	DD-Cover	90	913	309

- 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

17

Results

- DD is 2.5 - 3 times faster than EX



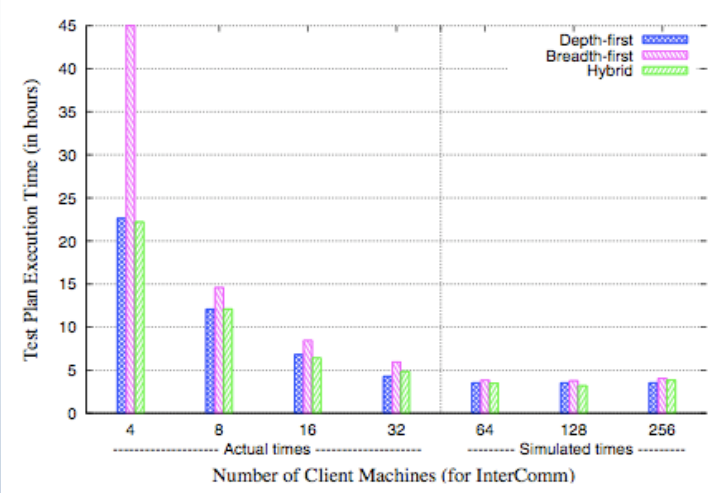
18

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

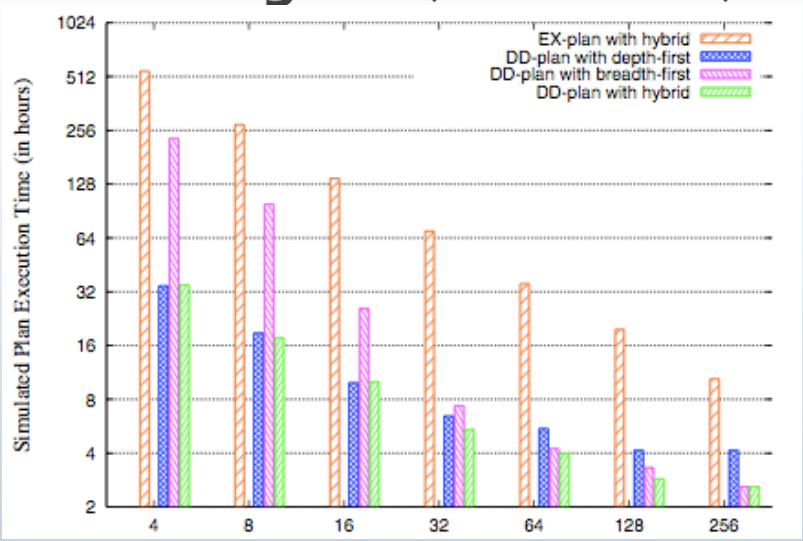
19

Comparison of Test Plan Strategies



20

Rachet behavior as successful builds grow (simulated)



21

Related Work

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

22

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

23

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

24