Quality Assurance for Large-Scale Systems

- Modern systems increasingly complex
  - Run on numerous platform, compiler & library combinations
  - Have 10’s, 100’s, even 1000’s of configuration options
  - Are evolved incrementally by geographically-distributed teams
  - Composed with other frequently changing systems
  - Have multi-faceted quality objectives
- How do you QA systems like this?
State of the Practice

• Continuous Build, Integration & Test (CBIT)
  – BuildBot, DART, GridUnit, CruiseControl
• Works well over a very small part of the full system space
• But large portions of system space go unexplored
  – Can’t fully predict effect of changes
  – Hard to interpret from-the-field failure reports & feedback

Distributed Continuous Quality Assurance

• DCQA processes conducted around-the-world, around-the-clock on powerful, virtual computing grids
  – Grids can be made up of end-user machines, project-wide resources or dedicated computing clusters
• General Approach
  – Divide QA processes into numerous tasks
  – Intelligently distribute tasks to clients who then execute them
  – Merge and analyze incremental results to efficiently complete desired QA process
Distributed Continuous Quality Assurance

- Expected benefits
  - Massive parallelization allows more QA faster
  - Improved access to resources/environs. not readily found in-house
  - Coordination effort & sharing results enables deeper analyses

Collaborators

Doug Schmidt & Andy Gohkale

Alex Orso, Myra Cohen

NISS Murali Haran, Alan Karr, Mike Last, & Ashish Sanil

Sandro Fouché, Atimf Memon, Alan Sussman, Cemal Yilmaz (Sabanci University) & Il-Chul Yoon
Skoll DCQA Infrastructure & Approach


1. Model QA Space

Skoll DCQA Infrastructure & Approach

2. Generate Initial QA tasks

See: A. Porter, C. Yilmaz, A. Memon, A. Nagarajan, D. C. Schmidt, and B. Natarajan,
Skoll: A Process and Infrastructure for Distributed Continuous Quality Assurance. IEEE

Skoll DCQA Infrastructure & Approach

3. Distribution

See: A. Porter, C. Yilmaz, A. Memon, A. Nagarajan, D. C. Schmidt, and B. Natarajan,
Skoll: A Process and Infrastructure for Distributed Continuous Quality Assurance. IEEE
Skoll DCQA Infrastructure & Approach

4. Feedback


Skoll DCQA Infrastructure & Approach

5. Steering

Implementing DCQA Processes in Skoll

- Define how QA analysis will be subdivided by defining generic QA tasks & the QA space in which they operate
  - QA tasks: Templates parameterized by “configuration”
    - Variable information needed to run concrete QA tasks
  - QA space: the set of all “configurations” in which QA tasks run
- Define how QA tasks will be scheduled & how results will be processed by defining navigation strategies
  - Navigation strategies “visit” points in the QA space, applying QA tasks & processing incremental results

The ACE+TAO+CIAO (ATC) System

- ATC characteristics
  - 2M+ line open-source CORBA implementation
  - maintained by 40+, geographically-distributed developers
  - 20,000+ users worldwide
  - Product Line Architecture with 500+ configuration options
    - runs on dozens of OS and compiler combinations
  - Continuously evolving – 200+ CVS commits per week
  - Quality concerns include correctness, QoS, footprint, compilation time & more
## Define QA Space

<table>
<thead>
<tr>
<th>Options</th>
<th>Type</th>
<th>Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>compile-time</td>
<td>{Linux, Windows XP, …}</td>
</tr>
<tr>
<td>TAO_HAS_MINIMUM_CORBA</td>
<td>compile-time</td>
<td>{True, False}</td>
</tr>
<tr>
<td>ORBCollocation</td>
<td>runtime</td>
<td>{global, per-orb, no}</td>
</tr>
<tr>
<td>ORBConnectionPurgingStrategy</td>
<td>runtime</td>
<td>{lru, lifu, fifo, null}</td>
</tr>
<tr>
<td>ACE_version</td>
<td>component version</td>
<td>{v5.4.3, v5.4.4,…}</td>
</tr>
<tr>
<td>TAO_version</td>
<td>component version</td>
<td>{v1.4.3, v1.4.4,…}</td>
</tr>
<tr>
<td>run(ORT/run_test.pl)</td>
<td>test case</td>
<td>{True, False}</td>
</tr>
</tbody>
</table>

### Constraints

\[(TAO\_HAS\_AMI) \Rightarrow (\neg TAO\_HAS\_MINIMUM\_CORBA)\]

\[run(ORT/run\_test.pl) \Rightarrow (\neg TAO\_HAS\_MINIMUM\_CORBA)\]

## Nearest Neighbor Search
Nearest Neighbor Search
Nearest Neighbor Search

Nearest Neighbor Search
Fault Characterization

• We used machine learning techniques (classification trees) to model option & setting patterns that predict test failures

Some Novel DCQA Processes & Feasibility Studies

• Configuration-level fault characterization
• Performance-oriented regression testing
• Build testing of component-based systems
Configuration-Level Fault Characterization

**Goal**
- Help developers localize configuration-related faults

**Current Solution Approach**
- Use covering arrays to sample the cfg space to test for subspaces in which regression tests fail
- Build models that characterize the configuration options and specific settings that define the failing subspace

---

Covering Arrays (CAs)

- Derive a test schedule from $t$-way covering arrays
  - a set of configurations in which all ordered $t$-tuples of option settings appear at least once

- $2$-way covering array example:

<table>
<thead>
<tr>
<th>Configurations</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1$</td>
</tr>
<tr>
<td>$O_1$</td>
</tr>
<tr>
<td>$O_2$</td>
</tr>
<tr>
<td>$O_3$</td>
</tr>
</tbody>
</table>

---

See: C. Yilmaz, M. Cohen, A. Porter, Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces, ISSTA'04, TSE v32 (1)
Covering Array Limitations

- Must choose a CA strength, $t$, before computing it
  - No way to know, \textit{a priori}, what the right value is
  - Our experience shows failures patterns changing over time
- Choose too high:
  - Run more tests than necessary
  - Testing might not finish before next testing session starts
    - Non-uniform sample negatively affects classification performance
- Choose too low:
  - Non-uniform sample negatively affects classification techniques
  - Must repeat process at higher strength

Incremental Covering Arrays

- Start with traditional CA of low strength (usually 2)
- Execute tests & classify observed failures
- If resources allow or classification performance requires
  - Increment strength
  - Build new covering array using previously run array(s) as seeds

Incremental Covering Arrays (cont.)

• Also have a variant for when you want to generate multiple CAs at each level

MySQL Case Study

• Project background
  – Widely-used, 2M+ line open-source database project
  – Cont. evolving & maintained by geographically-dist. developers
  – Dozens of cfg opts
  – Runs on dozens of OS/compiler combos

• Case study using release 5.1
  – Used 23 cfg opts with 2-5 settings each (~72M unique cfgs)
  – ~800 tests/per cfg across a grid of 50 machines
  – 3 builds: 1.2510 (2.5hrs), 1.2511 (2 days), 1.2512 (2 weeks)

• Questions
  – How big are incremental CAs?
  – How do costs & benefits of new approach compared to traditional?
Some Results

• Built 3 traditional and incremental CAs for $2 \leq t \leq 4$
  – Traditional sizes: 66, 238, 832
  – Incremental sizes: 68, 180, 586

• Tested 5994 cfgs across all 3 builds
• Both approaches exposed & classified the same failures
• Costs depend on $t$, failures patterns & time available
  – At worst size of inc. CA’s 5% greater than trad.
  – At best size of inc. CA’s 27% less than trad.
  – Incremental approach found some failures earlier than traditional

Partial Summary

• Applied process to a cfg space with over 72M cfgs
• Found many test failures from real bugs
• Incremental approach more flexible than traditional
• Appears to offer substantial savings in best case, while incurring minimal cost in worst case
Performance-Oriented Regression Testing

Goal

- Quickly determine whether a software update degrades performance

Solution Approach

- Proactively find a small set of configurations on which to estimate performance across entire configuration space
- Later, whenever software changes, benchmark this set and compare post-update & pre-update performance


Reliable Effects Screening (RES) Process

- Proactive
  - Compute a formal experimental plan (we use screening designs) for identifying "important" options
  - Execute the experimental plan across QA grid
  - Analyze the data to identify important options
  - Recalibrate frequently as system changes
- Reactive
  - When the software changes estimate the distribution of performance across entire configuration space
    - by evaluating all combinations of important options, (called the screening suite), while randomizing the rest
  - Distributional changes signal possible performance degradations
Feasibility Study

- Several recent changes to message queuing strategy.
- Has this degraded performance?
- Developers identified 14 potentially important binary options ($2^{14}=16,384$ configs.)

<table>
<thead>
<tr>
<th>Option Name</th>
<th>Option Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORBReactor-ThreadQueue</td>
<td>(FIFO, LIFO)</td>
</tr>
<tr>
<td>ORBClientConnectionHandler</td>
<td>{RW, MT}</td>
</tr>
<tr>
<td>ORBConnectionPurgingStrategy</td>
<td>(LRU, LFU)</td>
</tr>
<tr>
<td>ORBConcurrency</td>
<td>{reactive, thread-per-connection}</td>
</tr>
</tbody>
</table>

Identify Important Options

* Options B and J: clearly important, options I, C, and F: arguably important
* Screening designs mimics exhaustive testing at a fraction of the cost
Estimate Performance with Screening Suite

- Estimates closely track actuals

![Q-Q Plot for Latency](image)

Applying RES to Evolving System

- Estimated performance once a day on CVS snapshots of ATC

- Observations
  - Developers noticed the deviation on 12/14/03, but not before
  - Developers estimated a drop of 5% on 12/14/03.
  - Our process more accurately shows around 50% drop

![Latency Distribution Over Time](image)
Partial Summary

- Reliable Effects Screening is an efficient, effective, and repeatable approach for estimating the performance impact of software updates
- Was effective in detecting actual performance degradations
- Cut benchmarking work 1000x (from 2 days to 5 mins.)
  - Fast enough to be part of check-in process

Build Testing of Component-Based Systems

Goal

- Given a component-based system, determine which cfgs (components & their specific versions) do/don’t build

Solution Approach

- Model the configuration space, test a sample of the space & identify subspaces in which build process fails

The InterComm (IC) Framework

- Middleware for creating coupled scientific simulations
  - Built from up to 14 other components
  - Each comp can have several actively maintained versions
  - There are complex configuration constraints

- Developers need help to
  - Identify working/broken configurations
  - Broaden working set (to increase potential user base)
  - Rationally manage support activities

Annotated Component Dependency Graph

- ACDG = (CDG, Ann)
  - CDG: DAG capturing inter-comp deps
  - Ann: comp. versions & constraints

- Constraints for each cfg, e.g.,
  - \( \text{ver}(gf) = x \rightarrow \text{ver}(gcr) = x \)

- Can generate cfgs from ACDG
  - 3552 total cfgs. Takes up to \( \sim 10,700 \) CPU hrs to build all
Improving Test Execution

- Cfgs often share common build subsequences. Build effort should be reusable across cfgs
- Combine all cfgs into a data structure called a prefix tree
- Execute implied test plan across grid by (1) assigning subpaths (partial cfgs) to clients, (2) building each partial cfg in a VM & caching the VMs to enable reuse
- If all cfgs build, with 8 machines caching up to 8 VMs each, exhaustive testing would take ~355 hours

Direct-Dependency (DD) Coverage

- A is directly dependent on all components, {B}, such that there is path from A to B containing no component nodes
- Sampling approach
  - Identify all DDs for every component
  - Identify all valid instantiations of the DDs (version combos that violate no constraints)
  - Select a (small) set of cfgs that cover all valid instantiations of the DDs
Executing the DD Coverage Test Suite

- DD test suite much smaller than exhaustive
  - 211 cfgs with 649 comps vs 3552 cfgs with 9919 comps
  - For IC, no loss of test eff. (same build failures exposed)
- Speedups achieved using 8 machines w/ 8 VM cache
  - Actual case: 2.54 (18 vs 43 hrs)
  - Best case: 14.69 (52 vs 355 hrs)

Partial Summary

- Infrastructure in place & working
  - Complete client/server implementation using VMware
  - Simulator for exploratory analysis
- Initial results promising, but lots of work remains
- Ongoing activities
  - Alternative algorithms & test execution policies
  - More theoretical study of sampling & test exec approaches
  - Apply to more & different kinds of software systems
Future Work

- Continue improving Skoll system
- Looking for more subject systems & applications
- Incremental build testing
- Using source code analysis to further reduce state spaces

The End