How to Actually DO High-Volume Automated Testing

Cem Kaner
Carol Oliver
Mark Fioravanti
Florida Institute of Technology

Presentation Abstract

In high volume automated testing (HiVAT), the test tool generates the test, runs it, evaluates the results, and alerts a human to suspicious results that need further investigation.

Simple HiVAT approaches use simple oracles—run the program until it crashes or fails in some other extremely obvious way.

More powerful HiVAT approaches are more sensitive to more types of errors. They are particularly useful for testing combinations of many variables and for hunting hard-to-replicate bugs that involve timing or corruption of memory or data.

Cem Kaner, Carol Oliver, and Mark Fioravanti present a new strategy for teaching HiVAT testing. We’ve been creating open source examples of the techniques applied to real (open source) applications. These examples are written in Ruby, making the code readable and reusable by snapping in code specific to your own application. Join Cem Kaner, Carol Oliver, and Mark Fioravanti as they describe three HiVAT techniques, their associated code, and how you can customize them.

Acknowledgments

- Our thanks to JetBrains (jetbrains.com), for supporting this work through an Open Source Project License to their Ruby language IDE, RubyMine.
- These notes are partially based on research supported by NSF Grant CCLI-0717613 “Adaptation & Implementation of an Activity-Based Online or Hybrid Course in Software Testing.” Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

What is automated testing?

- There are many testing activities, such as…
  - Design a test, create the test data, run the test and fix it if it’s broken, write the test script, run the script and debug it, evaluate the test results, write the bug report, do maintenance on the test and the script
- When we automate a test, we use software to do one or more of the testing activities
  - Common descriptions of test automation emphasize test execution and first-level interpretation
  - But that automates only part of the task
  - Automation of any part is automation to some degree
- All software tests are to some degree automated and no software tests are fully automated.

What is “high volume” automated testing?

- Imagine automating so many testing activities
  - That the speed of a human is no longer a constraint on how many tests you could run
  - That the number of tests you run is determined more by how many are worth running than by how much time they take
- This is the underlying goal of HiVAT

Once upon an N+1th time…

- Imagine developing an N+1th version of a program
  - How could we test it?
- We could try several techniques
  1. **Long sequence regression testing.** Reuse regression tests from previous version. For those tests that the program can pass individually, run them in long random sequences
  2. **Program equivalence testing**.
     - Start with function equivalence testing. For each function that exists in the old version and the new one, generate random inputs, feed them to both versions and compare results. After a sufficiently extensive series, conclude the functions are equivalent
     - Combine tests of several functions that compare (old versus new) successfully individually.
  3. **Large-scale system comparison** by analyzing large sets of user data from the old version (satisfied user attests that they are correct). Check that the new version yields comparable results.
     - Before running the tests, run data qualification tests (test the plausibility of the inputs and user-supplied sample outputs). The goal is to avoid garbage-in-garbage-out troubleshooting.

Working definitions of HiVAT

- A family of test techniques that use software to generate, execute and interpret arbitrarily many tests.
- A family of test techniques that empower a tester to focus on interesting results while using software to design, implement, execute, and interpret a large suite of tests (thousands to billions of tests).

Breaking out potential benefits of HiVAT (different techniques → different benefits)

- **Find bugs we don't otherwise know how to find**
  - Diagnostics-based
  - Long sequence regression
  - Load-enhanced functional testing
  - Input fuzzing
  - Hostile data stream testing
- **Increase test coverage inexpensively**
  - Fuzzing
  - Functional equivalence
  - High-volume combination testing
  - Inverse operations
  - State-model based testing
  - High-volume parametric variation
- **Qualify large collections of input or output data**
  - Constraint checks
  - Functional equivalence testing

Examples of the techniques

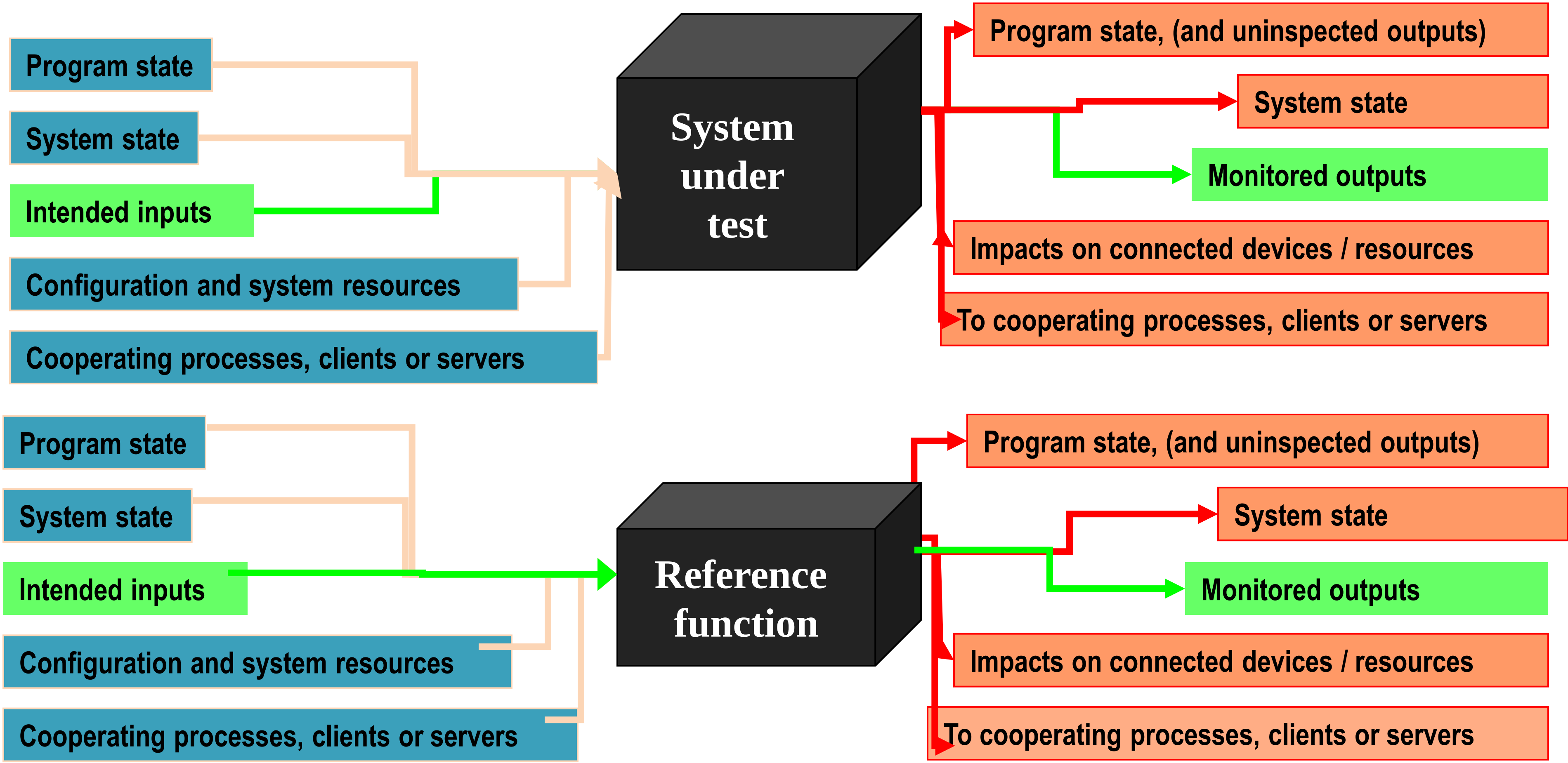
- Equivalence testing
  - Function (e.g. MASPAR)
  - Program
  - Version-to-version with shared data
- Constraint checking
  - Continuing the version-to-version comparison
- LSRT
  - Mentsville
- Fuzzing
  - Dumb monkeys, stability and security tests
- Diagnostics
  - Telenova PBX intermittent failures

Oracles Enable HiVAT

- Some specific to the testcase
  - Reference Programs: Does the SUT behave like it?
  - Regression Tests: Do the previously-expected results still happen?
- Some independent of the testcase
  - Fuzzing: Run to crash, run to stack overflow
  - OS/System Diagnostics
- Some inform on only a very narrow aspect of the testcase
  - You'd never dream that passing the oracle check means passing the test in all possible ways
  - Only that passing the oracle check means the program does not fail in this specific way
  - E.g. Constraint Checks
- The more extensive your oracle, the more valuable your high-volume test suite
  - What potential errors have we covered?
  - What potential errors have to be checked in some other way?

Reference Oracles are Useful but Partial

Based on notes from Doug Hoffman



Program state

System state

Intended inputs

Configuration and system resources

Cooperating processes, clients or servers

System under test

Program state, (and uninspected outputs)

System state

Monitored outputs

Impacts on connected devices / resources

To cooperating processes, clients or servers

Program state

System state

Intended inputs

Configuration and system resources

Cooperating processes, clients or servers

Reference function

Program state, (and uninspected outputs)

System state

Monitored outputs

Impacts on connected devices / resources

To cooperating processes, clients or servers

Using Oracles

- If you can detect a failure, you can use that oracle in any test, automated or not
- Many ways to combine oracles, as well as using them one at a time
- Each has time costs and timing costs
- Heisenbug costs mean you might use just one or two oracles at a time, rather than all the possible oracles you know, all at once.

Practical Things

MAKING HIVAT WORK FOR YOU

Basic Ingredients

- A Large Problem Space
  - The payoff should be worth the cost of setting up the computer to do detailed testing for you
- A Data Generator
  - Data generation often requires more human effort than we expect. This constrains the breadth of testing we can do, because all human effort takes time.
- A Test Runner
  - We have to be able to read the data, run the test, and feed the test to the oracle without human intervention. Probably we need a sophisticated logger so we can trace what happened when the program fails.
- An Oracle
  - This might tell us definitively that the program passed or failed the test or it might look more narrowly and say that the program didn't fail in this way. In either case, the strength of the test is driven by your ability to tell whether it failed. The narrower the oracle, the weaker the technique.

Two Examples and a Future

- Function equivalence tests (testing against a reference program)
- Long-sequence regression (repurposing already-generated data with new oracles)
- A more flexible HiVAT architecture

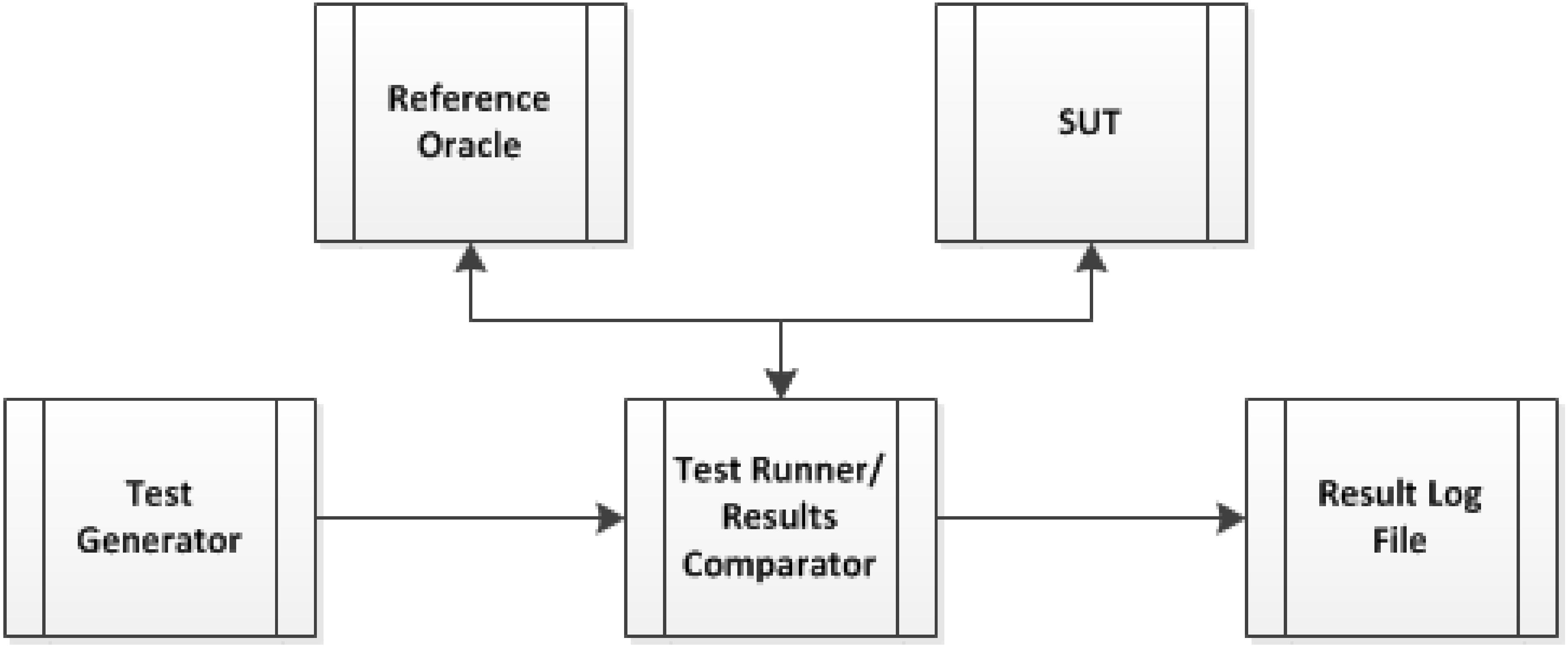
Functional Equivalence

- Oracle: A Reference Program
- Method:
  - The same testcases are sent to the SUT and to the Reference Program
  - Matched behaviors are considered a Pass
  - Discrepant behaviors are considered a Fail
- Limits:
  - The reference program might be wrong
  - We can only test comparable functions
  - Generating semantically meaningful inputs can be hard, especially as we create nontrivial ones
- Benefits: For every comparable function or combination of functions, you can establish that your SUT is at least as good as the Reference Program

Functional Equivalence Examples

- Square Root calculated one way vs. Square Root calculated another way (Hoffman)
- Open Office Calc vs. a reference implementation
  - Comparing
    - Individual functions
    - Combinations of functions
  - Reference
    - Currently reference formulas programmed in Ruby
    - Coming (maybe by STAR) compare to MS-Excel

Functional Equivalence Architecture



Functional Equivalence: Key Ideas

- You provide your Domain Knowledge
  - How to invoke meaningful operations for your system
  - Which data to use during the tests
  - How to compare the resulting data states
- Test Runner sends the same test operations to both the Reference Oracle and the System Under Test (SUT)
  - Collects both answers
  - Compares the answers
  - Logs the result

Function Equivalence: Architecture

- Component independence allows reusability for other applications and for later versions of this application.
  - We can snap in a different reference function
  - We can snap in new versions of the software under test
  - The test generator has to know about the category of application (this is a spreadsheet) but not which particular application is under test.
  - In the ideal architecture, the test running passes test specifications to the SUT and oracle without knowing anything about those applications. To the maximum extent possible, we want to be able to reuse the test runner code across different applications
  - The log interpreter probably has to know a lot about what makes a pass or a fail, which is probably domain-specific.

Functional Equivalence: Reference Code

- Work in Progress
  - http://testingeducation.org/
  - See HiVAT section of site

Long-Sequence Regression

- Oracles:
  - OS/System diagnostics
  - Checks built into each regression test
- Method:
  - Reconfigure some known-passing regression tests to ensure they can build upon one another (i.e. never reset the system to a clean state)
  - Run a subset of those regression tests randomly, interspersed with diagnostics
  - When a test fails on the N+1st time that we run it, after passing N times during the run, analyze the diagnostics to figure out what happened and when in the sequence things first went wrong

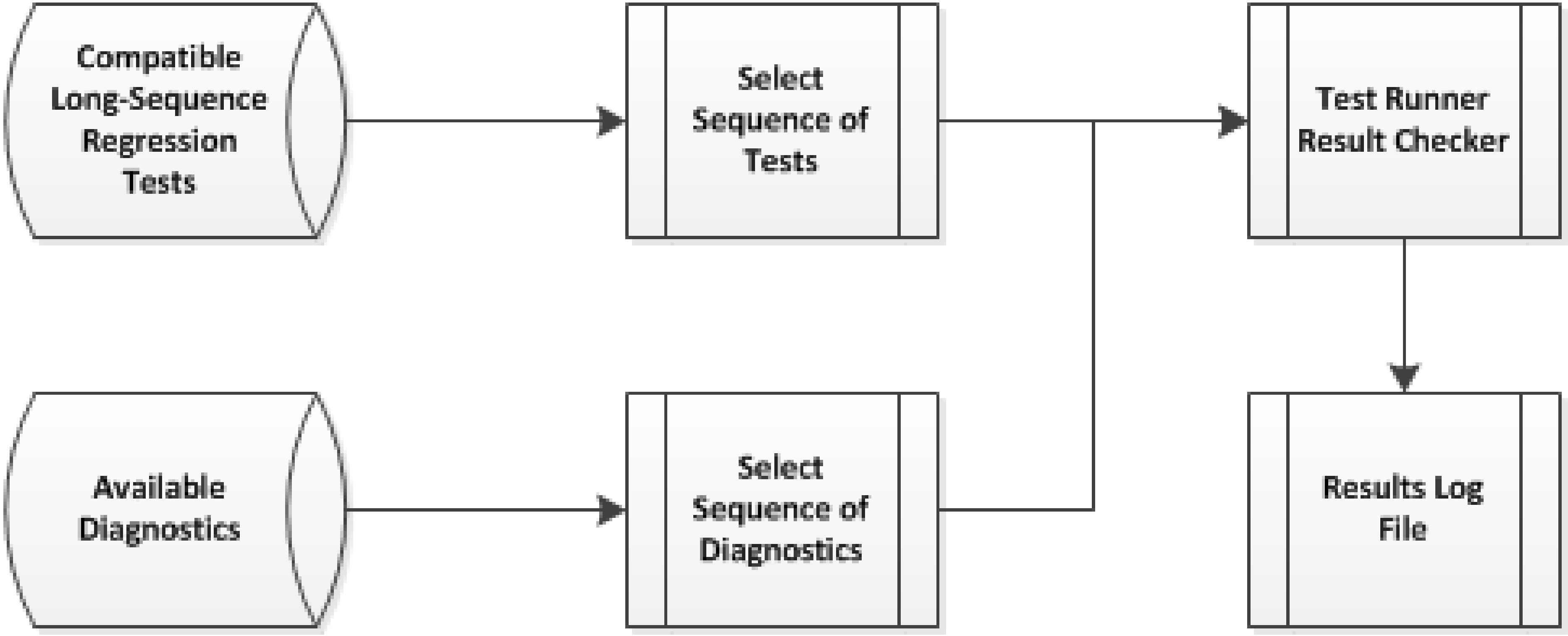
Long-Sequence Regression

- Limits:
  - Failures will be indirect pointers to problems
  - You will have to analyze the logs of diagnostic results
  - You may need to rerun the sequence with different diagnostics
  - Heisenbug problem limits the number of diagnostics that you can run between tests
- Benefits:
  - This approach is proven good for finding timing errors and memory misuse

Long-Sequence Regression Examples

- Mentsville
- Application to OpenOffice

Long-Sequence Regression Architecture


Compatible Long-Sequence Regression Tests
Select Sequence of Tests
Test Runner Result Checker
Available Diagnostics
Select Sequence of Diagnostics
Results Log File

Test Runner/Result Checker Operation

- Diagnostics
- A Regression Test
- Diagnostics
- A Regression Test
- Diagnostics
- A Regression Test
- Diagnostics
- ……..
- Final Diagnostics

Long-Sequence Regression: Key Ideas

- Pool of Long-Sequence-Compatible Regression Tests
  - Each of these is Known to Pass, individually, on this build
  - Each of these is Known to NOT reset system state (therefore it contributes to simulating real use of the system in the field)
  - Each of these is Known to Pass when run 100 times in a row, with just itself altering the system (if not, you've isolated a bug)
- Pool of Diagnostics
  - Each of these collects something interesting about the system or program: memory state, CPU usage, thread count, etc.

Long-Sequence Regression: Key Ideas

- Selection of Regression Tests
  - Need to run a small enough set of regression tests that they will repeat frequently during the elapsed time of the Long Sequence
  - Need to run a large enough set of regression tests that there's real opportunity for interactions between functions of the system
- Selection of Diagnostics
  - Need to run a fixed set of diagnostics per Long Sequence, so that you collect data to compare Regression Test failures against
  - Need to run a small enough set of diagnostics that the act of running the diagnostics doesn't significantly change the system state

Long-Sequence Regression: Reference Code

- Work in Progress (not currently available)
  - http://testingeducation.org/
  - See HiVAT section of site

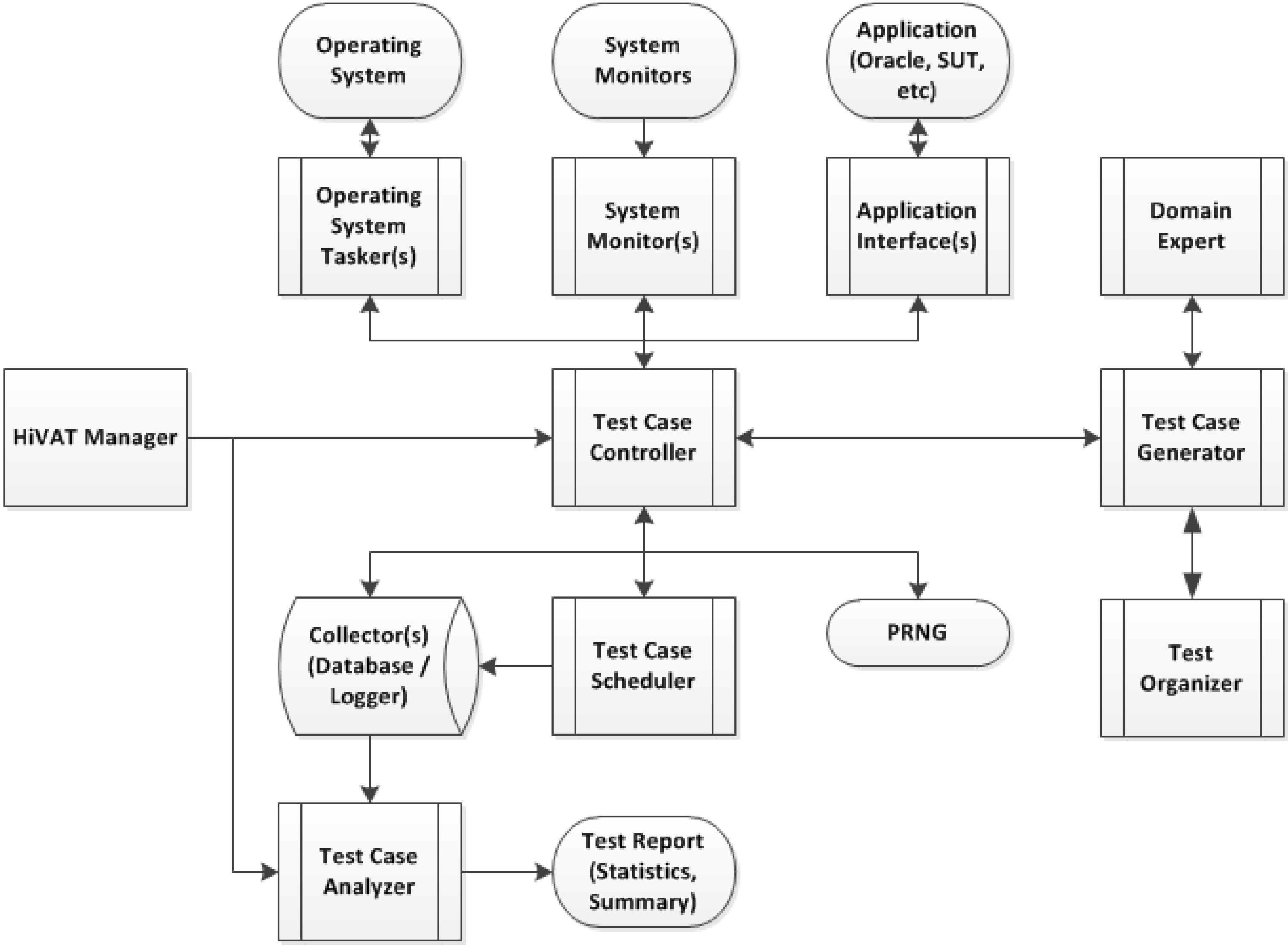
Future Power

A FLEXIBLE ARCHITECTURE FOR HIVAT

Maadi HiVAT Architecture

- Maadi is a generalized HiVAT architecture
  - Open Source and written in Ruby
  - Developed to support multiple HiVAT techniques
  - Developed to support multiple applications and workflows
  - Based on demonstrations from previous implementations
- Architecture is divided into Core and Custom components
  - Custom components are where the specific implementations reside
- Maadi, one of the earliest Egyptian mines
  - Implemented in Ruby, used RubyMine IDE, etc.

Maadi HiVAT Architecture Overview



Maadi Core Components

- Application – Ruby Interface to SUT
- Collector – Logging Interface
- Monitor – Diagnostic Utility Interface
- Tasker – Resource Consumption Interface
- Organizer – Implements the Test Technique
- Expert – Domain Expert which builds “skeleton” Procedures
- Generator – Mediates interaction between Organizer and Expert to assemble a series of Tests
- Scheduler – Collects and orders the completed Procedures
- Controller – Mediates interaction between Scheduler and the Applications, Monitors, and Taskers to conduct tests
- Analyzer – Performs analysis of collected Results
- Manager – Overall Test Conductor

HiVAT Test Process

- Configure Environment
  - CLI enables scripted runs via a profiles *
- Prepare Components
  - Validate component compatibility
  - Generate Test Cases, Schedule Test Cases, Log Test Cases
  - Log Configuration
- Execute Test Plan
  - Record results
- Generate Test Report

* Entire process can be scripted via profiles; start to finish

Language Restrictions

- Applications, Organizer and Expert must all understand the same “procedural” language in order to communicate the selection, construction, and execution of a test case
  - Using a SQL Domain Expert to generate Test Cases for Microsoft Solitaire is not useful
  - Controller will validate that all Applications and the Organizer can accept the Domain Expert
- Applications and Experts must also understand a second “result” language in order to capture the results

Test Plan Complexities

- Functional Equivalence exposed a challenge:
    - How to flexibly specify test details that required random choice on-the-fly
- Early Attempt: Try to Pre-Specify all the details you want to have vary:
    - Plan 1: log (RNG_1)
        - CONSTRAINT RNG_1: Real, >0
    - Plan 2: Sum (0...RNG_1) of CellValue (RNG_2)
        - CONSTRAINT RNG_1: Integer, >0
        - CONSTRAINT RNG_2: Real
    - Plan 3: Sum (0...RNG_1) of CellValue (log (RNG_2))
        - CONSTRAINT RNG_1: Integer, >0
        - CONSTRAINT RNG_2: Real, >0 Delta= 0

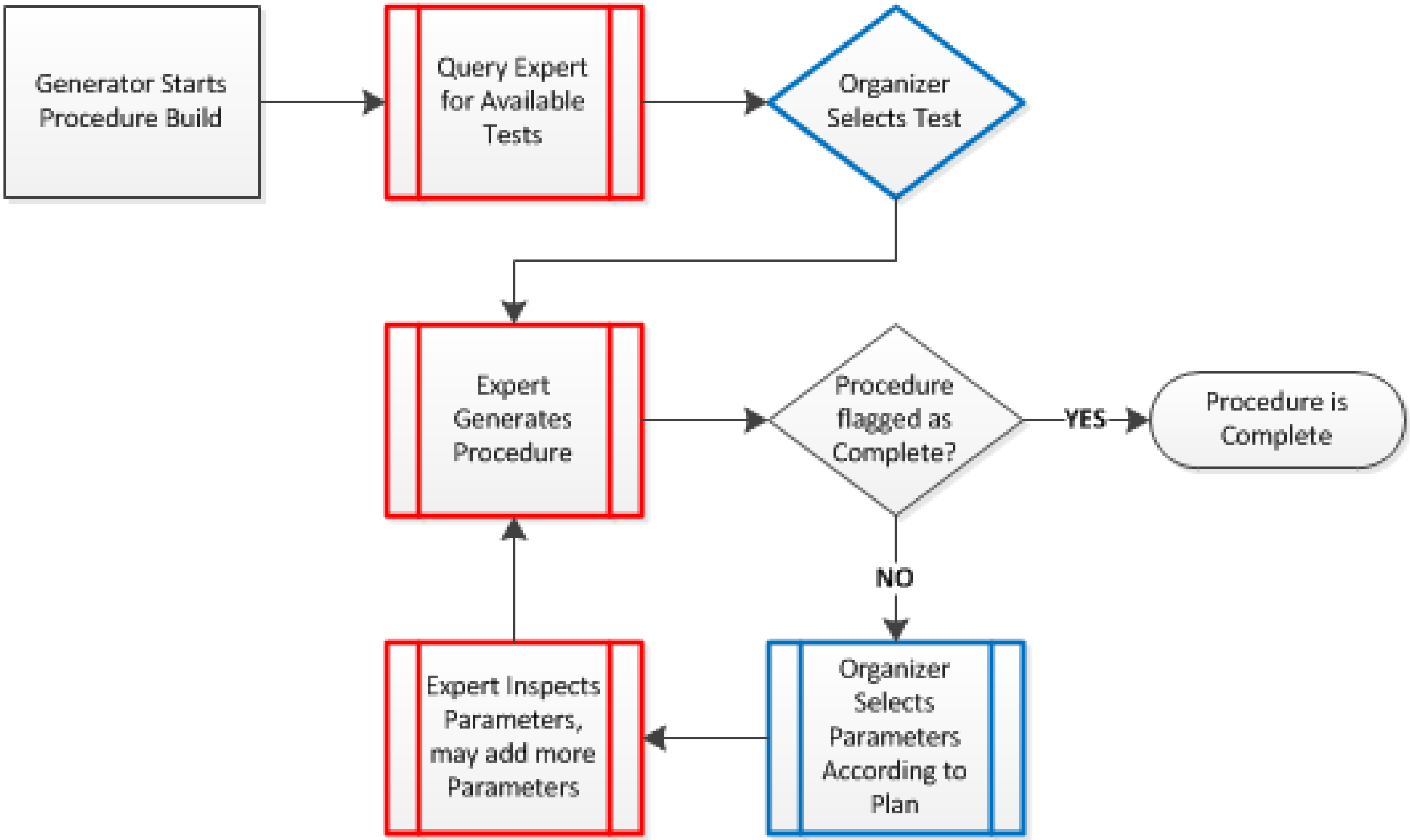
Test Plan Complexities (2)

- This gets awkward quickly:
  - Plan 4: (Sum (0...RNG_1) of CellValue (log(RNG_2))) /RNG_1
    - CONSTRAINT RNG_1: Integer, >0
    - CONSTRAINT RNG_2: Real, >0
- Really want to just specify:
  - The individual elements with their individual constraints
  - The rules for combining the elements together
- Then dynamically assemble the pieces in sensible but infinitely variable ways at test generation time
- New Approach: Mediate a **Conversation** with the Expert during test generation time

Generator Mediation

- Generator is responsible for generating a test case
  - Interaction between the Organizer and the Expert produces a test procedure
  - Generator supervises and terminates as necessary
- Test Case construction process overview:
  1. Organizer selects a Test to build
  2. Expert builds a skeleton, adds configurable parameters
  3. Organizer populates parameters
  4. Expert inspects parameter selection, expert may
     a) add additional parameters (return to step 2)
     b) flag the procedure as complete or failed

Generator Mediation (2)


Generator Starts Procedure Build
Query Expert for Available Tests
Organizer Selects Test
Expert Generates Procedure
Procedure flagged as Complete?
YES
Procedure is Complete
NO
Organizer Selects Parameters According to Plan
Expert Inspects Parameters, may add more Parameters

Organizer: One Example Test Plan

- Organizer is responsible for constructing a set of test procedures according to the Test Plan
  - The Test Plan is really the resultant set of Test Procedures
  - Organizers could be as simple as a Fuzzer
    - Test Plan could be satisfied by randomly select a value for any configurable parameter
  - Organizers could be as complex as desired and could be as specific as required
    - Test Plan could only be satisfied by successfully constructing a specific database

Organizer: One Example Test Plan (2)

- Example of a Test Plan
  - CREATE a TABLE tblStudents in dbUniversity with 5 columns
    - s_ID as a PRIMARY KEY, AUTO INCREMENT, INT
    - s_FamilyName as VARCHAR(255), NOT NULL
    - s_Program as VARCHAR(32), NOT NULL
    - s_Major as INTEGER, FOREIGN KEY(tblMajors, m_ID)
    - s_Enrolled as DATETIME
  - CREATE a TABLE tblStudentClasses in dbUniversity with 3 columns
  - …
- Following example will illustrate the process of building a procedure to according to the Test Plan

Example Procedure Construction

- Participants
  - Expert: Structured Query Language (SQL) Domain Expert
  - Organizer: Construct University Database
- Beginning Test Selection
- Expert provides a list of possible tests
  - the SQL Expert provides:
    - `CREATE, SELECT, INSERT, UPDATE, DELETE, DROP`
- Organizer selects test, requests procedure
  - the Organizer would choose:
    - `CREATE`
- After the Test has been chosen, the Organizer can request that the Expert provide a Test Procedure

Example Procedure Construction (2)

- Expert’s Initial response to choosing CREATE

```
id: CREATE-NEW-NEXT
step: CREATE [TYPE]
*  parameter: [CREATE-TYPE] =
  constraint: PICK-LIST (CHOOSE ONE: TABLE DATABASE)
```

- Expert returns procedure with 1 parameter: [CREATE-TYPE]
  - Organizer has 2 options; choose TABLE or DATABASE
- Each parameter has a constraint which the Organizer should follow
  - Many different types of constraints possible
  - Expert may or may not enforce the constraint; if not followed the procedure may be flagged as invalid
- Organizer returns procedure with option selected

Example Procedure Construction (3)

- Organizer’s Test Plan requires that the Procedure for a CREATE TABLE be built
  - Organizer selects TABLE for the [CREATE-TYPE] parameter
- Organizer returns the following procedure to the Expert

```
id: CREATE-NEW-NEXT
step: CREATE [TYPE]
*  parameter: [CREATE-TYPE] = TABLE
  constraint: PICK-LIST (CHOOSE ONE: TABLE DATABASE)
```

- Use of Step and Parameters is part of the agreed upon Procedural language
  - In the case of the SQL example, the parameters will be substituted inline to construct the desired SQL command

Example Procedure Construction (4)

- Expert’s response to [CREATE-TYPE] selection

```
id: CREATE-TABLE-PICK-DATABASE
step: CREATE [TYPE]
*  parameter: [CREATE-TYPE] =  TABLE
  constraint: PICK-LIST (CHOOSE ONE: TABLE DATABASE)
```

- Expert returns a procedure with no new parameters
  - Procedure ID has changed
    - Expert utilizes Procedure for determining next actions
    - Organizer has no selections to make but the procedure is not flagged as being complete, therefore it should return the procedure to the Expert for the next update

Example Procedure Construction (5)

- Expert's response to returned procedure

```
id: CREATE-TABLE-NEW
step: CREATE [TYPE]
*  parameter: [CREATE-TYPE] =  TABLE
  constraint: PICK-LIST (CHOOSE ONE: TABLE DATABASE)
*  parameter: [DATABASE-NAME] =
  constraint: PICK-LIST (CHOOSE ONE: HiVAT, dbUniversity)
```

- Expert returns procedure, which has 1 NEW parameter; [DATABASE-NAME]
  - [DATABASE-NAME] has a constraint in which the only two valid values are either HiVAT or dbUniversity
  - Procedure ID has changed (again)
- Expert has not modified or removed existing [CREATE-TYPE] parameter

Example Procedure Construction (6)

- Organizer selects value according to Test Plan
  - Organizer selects HiVAT for the [DATABASE-NAME] parameter
- Organizer returns the following procedure to the Expert

```
id: CREATE-TABLE-NEW
step: CREATE [TYPE]
*  parameter: [CREATE-TYPE] =  TABLE
  constraint: PICK-LIST (CHOOSE ONE: TABLE DATABASE)
*  parameter: [DATABASE-NAME] = dbUniversity
  constraint: PICK-LIST (CHOOSE ONE: HiVAT, dbUniversity)
```

- Organizer has only selected a parameter which was previously empty
  - If a previously selected parameter is changed, it may cause the Expert to flag the Procedure as Failed (e.g. setting [CREATE-TYPE] to DATABASE

Example Procedure Construction (7)

- Expert updates procedure and returns it to the Organizer

```
id: CREATE-TABLE-PARAMS
step: CREATE TABLE [TABLE-NAME]
*  parameter: [CREATE-TYPE] =  TABLE
  constraint: PICK-LIST (CHOOSE ONE: TABLE DATABASE)
*  parameter: [DATABASE-NAME] =  dbUniversity
  constraint: PICK-LIST (CHOOSE ONE: HiVAT, dbUniversity)
*  parameter: [TABLE-NAME] =
  constraint: ALPHANUMERIC WORD (LENGTH MIN: 5, MAX: 25)
*  parameter: [TABLE-COLUMNS] =
  constraint: RANGED-INTEGER (MIN: 1, MAX: 25)
```

- The procedure has 2 NEW parameters
  - [TABLE-NAME] has an ALPHANUMERIC constraint
  - [TABLE-COLUMNS] has a RANGED INTEGER constraint
  - Procedure ID and Step name have also changed

Example Procedure Construction (8)

- Organizer selects values according to Test Plan
  - Organizer selects
    - tblStudents for the [TABLE-NAME] parameter
    - 5 for the [TABLE-COLUMNS] parameter
- Organizer returns the following procedure to the Expert

```
id: CREATE-TABLE-PARAMS
step: CREATE TABLE [TABLE-NAME]
*  parameter: [CREATE-TYPE] =  TABLE
  constraint: PICK-LIST (CHOOSE ONE: TABLE DATABASE)
*  parameter: [DATABASE-NAME] =  dbUniversity
  constraint: PICK-LIST (CHOOSE ONE: HiVAT, dbUniversity)
*  parameter: [TABLE-NAME] = tblStudents
  constraint: ALPHANUMERIC WORD (LENGTH MIN: 5, MAX: 25)
*  parameter: [TABLE-COLUMNS] = 5
  constraint: RANGED-INTEGER (MIN: 1, MAX: 25)
```

Example Procedure Construction (9)

- Exchange between the Expert and Organizer continues until
  - Expert flags the Procedure as Complete
  - Expert flags the Procedure as Failed
  - Generator determines that the maximum number of exchanges have occurred *
    - Organizer and Expert may not be able to agree on acceptable test
- All procedures (including Failed procedures) are returned to the Controller
  - Controller is configured to discard failed procedures
  - If too many failed procedures are detected the user is notified and test is terminated *

* User configurable values

Maadi: Key Ideas

- Exploit the commonalities across each HiVAT technique
- Enable a conversation with the Knowledge Experts to empower the tests to be more dynamic, yet still sensible

Maadi: Reference Code

- Work in Progress
  - http://testingeducation.org/
  - See HiVAT section of site

Is this the future of testing?

- Maybe, but within limits
- Expensive way to find simple bugs
  - Significant troubleshooting costs: shoot a fly with an elephant gun and discover you have to spend enormous time wading through the debris to find the fly
- Ineffective way to hunt for design bugs
- Emphasizes the families of tests that we know how to code rather than the families of bugs that we need to hunt

As with all techniques

These are useful under some circumstances

Probably invaluable under some circumstances

But ineffective for some bugs and inefficient for others