USING STATISTICAL DISTRIBUTIONS TO GENERATE RANDOM TEST DATA

Jamie Zimmerman
Thesis Defense
Sometime Spring, 2018

Terms
SUT – Software Under Test: the program to be tested
Production rule – a recursive describing what the left-hand symbol could possibly be. For example, an A could be a B, denoted by A -> B
Test vector – A symbolic description of a test case, lacking exact data points but having an English language descriptor
Fault – misbehavior in a piece of software, also known as a bug
CSV file format – a comma-separated value file that is viewable in Microsoft Excel

Abstract

Many open-source software programs lack rigorous, wide-reaching testing. This is primarily because the testing process is deeply dependent on human invention and manual writing. Therefore, writing system tests is often avoided because of its financial and time expense on the software development lifecycle. However, the data for some system tests can be generated with special statistical distributions, which can exhibit certain trends or patterns and therefore tease out specific behaviors in the program that warrant being checked. This project aims to automate the creation of those data points to streamline the process of writing large test suites.

Terms
SUT – Software Under Test: the program to be tested
Production rule – a recursive describing what the left-hand symbol could possibly be. For example, an A could be a B, denoted by A -> B
Test vector – A symbolic description of a test case, lacking exact data points but having an English language descriptor
Fault – misbehavior in a piece of software, also known as a bug
CSV file format – a comma-separated value file that is viewable in Microsoft Excel

Introduction

Software testing is a crucial part of systems that run our lives. Industries that rely on predicting consumer habit trends or dispatching taxi cars are deeply affected when their software fails to provide accurate results. The gravity of this trust is even greater in safety-critical programs controlling gas-leak shut off valves or anesthetic delivery machines. Consequently, before these pieces of software are deployed, they must be checked and tested as rigorously as necessary, but no more. The software must eventually reach production (used in the hands of the customer or used in industrial practice “for real”), and cannot spend too much of its lifetime under test. Therefore, cost and convenience of testing is an important consideration in determining the rigor of testing.

A key component of getting correct results is knowing what that correctness looks like. Developers and testers must be careful not to confuse code that compiles and runs without errors with code that garners accurate results. To find accurate results, testers need an oracle, which is a way of determining how close the actual result of a program is to the expected result.

Unsurprisingly, attaining and checking this oracle is difficult. Given a particular input to a program, a tester must know what the expected result even is. Then, she must know how to read or understand the actual result and be able to compare it to the expected. Consequently, many companies often build the role of tester into the role of developer, since the developer has the most intimate knowledge of the design and implementation of the software, and therefore has a better sense of what the output of the program garners.

The breadth and completeness of testing is directly proportional to how much damage could occur if the software malfunctioned. However, there are some basic oracles and trends that can be tested that can offer high confidence that the program functions as anticipated. For example, it might be difficult to know what a piece of software should do, but significantly easier to know what the software should *not* do. Online banking software should not deposits millions of dollars into the account of a robber.

It is significantly easier to describe the trend of a certain test than it is to create the hundreds (if not more) data points that fit that description. Therefore, the goal of this paper is to aid the tester in designing and creating test suites so as to provide automation in the testing process. The reality of the software development environment means that solutions must be adopted to ensure accurate software while still maintaining a practical timeline and budget.

Another major challenge in software testing is the need to know all the possible situations you may want to test for. In the automobile simulation, a tester does not want to just test for braking and accelerating, they might want to test turn-signaling and beeping and brake-lighting, and also combinations of those tests to make sure that the beeping sensor does not accidentally disable the brake pedal capability. But knowing and describing every single situation may be unknown to the tester; they may not even think that that is a situation they have to test for. This is **Problem One**.

The next problem is writing the actual test data itself. A symbolic description of the test must be turned into an actual concrete input to the software. A test vector checking that beeping does not disable braking must be turned into "beeping=5s&&braking=true" or whatever format the SUT requires. This is not a step that should be done manually. VisIt, a graphical visualization tool, is capable of handling several gigabyte files, and no tester wants to or could hand write 3 gigabytes of data points just to test something. This is **Problem Two**.

Literature Survey

The essence of the oracle problem is best captured in the definition of non-testable programs, provided by Weyuker:

> "A program should be considered non-testable [if] (1) there does not exist an oracle; (2) it is theoretically possible, but practically too difficult to determine the correct output"[1]

There are a variety of reasons why the oracle does not exist, or can be exercised in a practical capacity. Weyuker describes that some software systems are like magic calculators – they are meant to inform us of the answer. She additionally mentions that some programs produce output that is impossible to read, either because of its volume or complexity. Here lies the core of software testing challenges. It is impossible to solve for every oracle or be able to describe every detail of a complex oracle – every system is entirely different and no generic template could create a standardized oracle. However, it is possible to avoid oracles altogether, generate them partially, or give hints about what details are most important to check.

One method of avoiding oracles altogether is metamorphic testing, which exploits the relationships of different executions of different input data[2, 3, 4]. For example, upon white box inspection of a system, testers can see that outputs should relate directly to their inputs. They may not know much else about the system, or what its output means, but they know what differences they should see upon two different executions[5]. Upon a certain execution they get f(5) = 20, and since they know the input-output relates directly, then they should predict that f(6) > f(5). Though exceptional in avoiding oracles, ascertaining metamorphic relationships is as challenging as the oracle problem.

Other researchers have proposed methods that do not attempt to provide or calculate an oracle, but rather aid the tester, informed of the structure of the system, exactly what details she should watch and constrain about the oracle. This takes the form of determining, through a series of tests, which variables in the system are most effective in revealing faults or bad behavior, and then providing them to the tester to define the "correct" value. One research group used mutant generation to find which variables killed the most mutants and, therefore, found the most faults[6]. An alternative solution identified chains of dependent variables using probabilistic substitution graphs to find the most important variables[7] (still unknown to the author how). These works make the best advances in the reducing the human effort needed to define and fulfill a complicated oracle.

Proposed Argument

I believe that I have designed a possible approach to these problems and have designed a simplistic solution that automates the steps described in the introduction. It is hypothesized that the Problem One is solved by the advent of combinatorial testing, which identifies most, if not all, of the test cases necessary for confidence in working software. I thank Michal for his construction of that tool that exists as the primary first step in this pipeline of testing automation. My tool, Parmgen, has solved the second problem, translating the English description of a test case into a concrete format that can be used as input into the program. I ask for the user for smaller, more simplistic descriptions of their input data, which creates less work than writing every data point by hand. Then I can use Python's random statistical distributions to generate those points, and write them to a convenient location for the tester's use. I call this pipeline GenSequence.

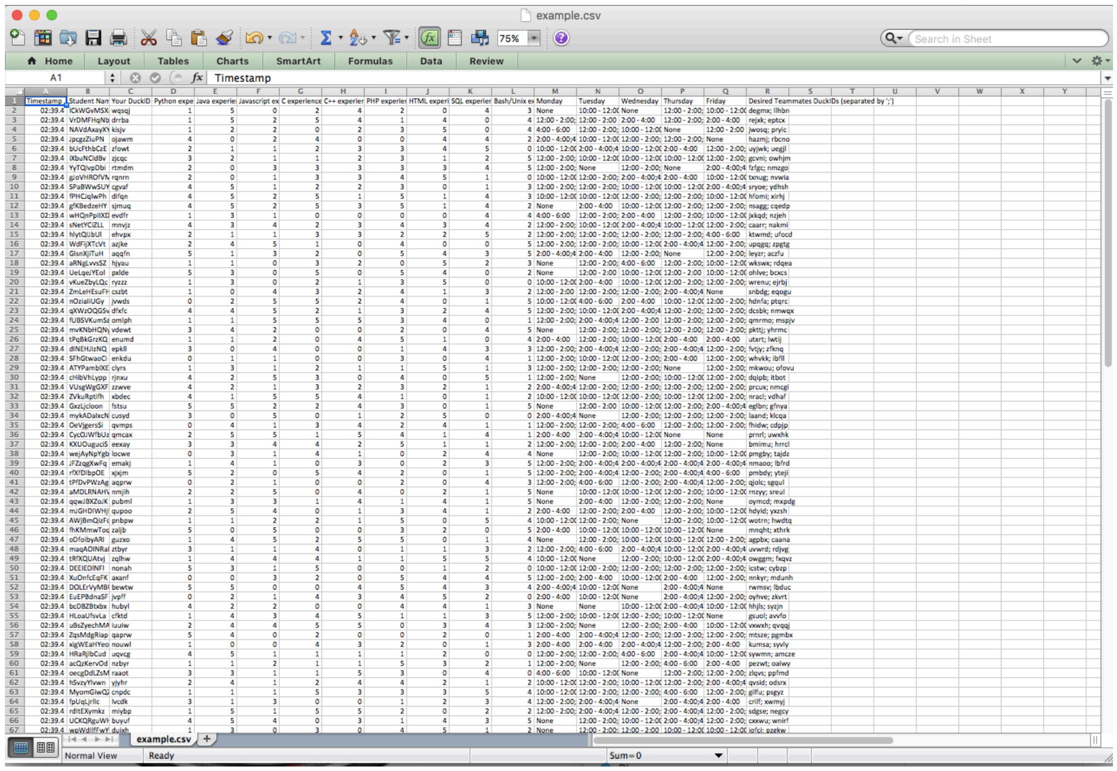
Background – Design Decisions

1. Parameters

The "input" to a program is a set of parameters, named variables that represent specific data points. How those parameters are generated, and what literal form they take, affects the outcome of the program. Large programs with larger inputs may benefit from having their input parameter data sets generated by statistical distributions.

I was inspired by this example by testing a project I helped build in my software engineering class, which took in the records of each student in a class of 40 students, describing things like their skills in certain technologies and their available free times during the week. This project made a fairly sophisticated decision about how to form optimal groups of students to work in teams, ensuring that each group has complementary skills and overlapping free times. During testing, it is important to consider bizarre scenarios, like what should happen when no student has any free time at all, or when half the class can only meet early in the week and the other half can only meet late in the week.

These parameters can follow certain trends that can be mimicked by repeated trials of statistical distributions. The input to this program is merely an Excel csv file of a record table, such that every row is a student's entry and every column is all the entries for certain parameters that really effect the outcome of the program.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	Timestamp	Student Nam	Your DuckID	Python expe	Java experie	Javascript ex	C experience	C++ experier	PHP experier	HTML experi	SQL experier	Bash/Unix ex	Monday	Tuesday	Wednesday	Thursday	Friday	Desired Teammates DuckIDs (separated by ';')
2	02:39.4	lCkWGvMSX	wqsqj	1	5	0	2	4	2	0	4	3	None	10:00 - 12:0(	None	12:00 - 2:00;	10:00 - 12:0(	degmx; llhbn
3	02:39.4	VrDMFHqNb	drrba	1	5	2	5	4	1	4	0	4	12:00 - 2:00;	12:00 - 2:00	2:00 - 4:00	12:00 - 2:00;	2:00 - 4:00	rejxk; eptcx
4	02:39.4	NAVdAxayXY	kisjv	1	2	2	0	2	3	5	0	4	4:00 - 6:00	12:00 - 2:00;	10:00 - 12:0(	None	12:00 - 2:00	jwosq; pryic
5	02:39.4	JpcgzZiuPN	ojawm	4	0	2	4	0	0	4	4	2	2:00 - 4:00;4	10:00 - 12:0(	12:00 - 2:00;	12:00 - 2:00;	None	hazmj; rbcno
6	02:39.4	bUcFthbCzE	zfowt	2	1	1	2	3	3	4	5	0	10:00 - 12:0(	2:00 - 4:00;4	10:00 - 12:0(	2:00 - 4:00	12:00 - 2:00;	uyjwk; uegjl
7	02:39.4	iXbuNCldBv	zjcqc	3	2	1	1	2	3	1	2	5	12:00 - 2:00;	10:00 - 12:0(	10:00 - 12:0(	10:00 - 12:0(	12:00 - 2:00;	gcvni; owhjm
8	02:39.4	YyTQJvpDbi	rtmdm	2	0	3	3	3	3	3	4	5	12:00 - 2:00;	None	12:00 - 2:00;	None	2:00 - 4:00;4	fzfgc; nmzgo
9	02:39.4	gJoVHROfVN	rqnrn	2	0	1	3	3	4	5	1	0	10:00 - 12:0(	12:00 - 2:00;	2:00 - 4:00;4	2:00 - 4:00	10:00 - 12:0(	txnug; nvwia
10	02:39.4	SPaBWwSUY	cgvaf	4	5	1	2	2	3	0	1	3	12:00 - 2:00;	12:00 - 2:00;	10:00 - 12:0(	10:00 - 12:0(	2:00 - 4:00;4	sryoe; ydhsh
11	02:39.4	fPHCJqlwPh	difqn	4	5	2	5	1	5	1	4	3	10:00 - 12:0(	10:00 - 12:0(	12:00 - 2:00;	12:00 - 2:00;	10:00 - 12:0(	hfomi; xirhj
12	02:39.4	gfKBedzeHY	sjmuq	4	5	2	5	3	5	1	4	2	None	2:00 - 4:00	10:00 - 12:0(	12:00 - 2:00;	12:00 - 2:00;	nsagg; cqedp
13	02:39.4	wHQnPpllXD	evdfr	1	3	1	0	0	0	0	4	4	4:00 - 6:00	12:00 - 2:00;	2:00 - 4:00	12:00 - 2:00;	10:00 - 12:0(	jxkqd; nzjeh
14	02:39.4	sNetYCiZLL	mnvjz	4	3	4	2	3	4	3	4	2	12:00 - 2:00;	10:00 - 12:0(	2:00 - 4:00;4	10:00 - 12:0(	12:00 - 2:00;	caarr; nakmi
15	02:39.4	hlytQJJbUI	ehvpx	2	1	1	3	3	2	2	5	2	12:00 - 2:00;	12:00 - 2:00;	12:00 - 2:00;	12:00 - 2:00;	4:00 - 6:00	ktwmd; ufocd
16	02:39.4	WdFijXTcVt	azjke	2	4	5	1	0	4	0	0	5	12:00 - 2:00;	12:00 - 2:00;	10:00 - 12:0(	2:00 - 4:00;4	12:00 - 2:00;	upqgq; zpgtg
17	02:39.4	GlsnXjiTuH	aqqfn	5	1	3	2	0	5	4	3	5	2:00 - 4:00;4	2:00 - 4:00	12:00 - 2:00;	None	12:00 - 2:00;	leyzr; aczfu
18	02:39.4	aRNgLvvsSZ	hjyau	1	1	0	3	2	0	5	2	3	None	12:00 - 2:00;	4:00 - 6:00	12:00 - 2:00;	10:00 - 12:0(	wkswx; rdqea
19	02:39.4	UeLqeJYEol	pxlde	5	3	0	5	0	5	4	0	2	None	12:00 - 2:00	10:00 - 12:0(	12:00 - 2:00	10:00 - 12:0(	ohlve; bcxcs
20	02:39.4	vKueZbyLQc	ryzzz	1	3	0	2	1	3	5	0	0	10:00 - 12:0(	2:00 - 4:00	10:00 - 12:0(	12:00 - 2:00;	12:00 - 2:00;	wrenu; ejrbj
21	02:39.4	ZmLeHEsuFH	cszbt	1	0	4	3	2	4	1	3	2	12:00 - 2:00	12:00 - 2:00;	12:00 - 2:00;	2:00 - 4:00;4	None	snbdg; eqogu
22	02:39.4	nOzialiUGy	jvwds	0	2	5	5	2	4	0	1	5	10:00 - 12:0(	4:00 - 6:00	2:00 - 4:00	10:00 - 12:0(	12:00 - 2:00;	hdnfa; ptqrc
23	02:39.4	qXWzOQGSv	dfxfc	4	4	5	2	1	3	2	4	5	12:00 - 2:00;	10:00 - 12:0(	2:00 - 4:00;4	12:00 - 2:00;	12:00 - 2:00;	dcsbk; nmwqx
24	02:39.4	fUBSVKumSz	omlph	1	1	5	5	5	3	4	0	1	12:00 - 2:00;	2:00 - 4:00;4	12:00 - 2:00	12:00 - 2:00;	12:00 - 2:00;	qmrmo; mspjv
25	02:39.4	mvKNbHQNy	vdewt	3	4	2	0	0	2	0	4	5	None	12:00 - 2:00;	12:00 - 2:00;	12:00 - 2:00;	12:00 - 2:00;	pkttj; yhrmc
26	02:39.4	tPqBkGrzKQ	enumd	1	1	2	0	4	5	1	0	4	2:00 - 4:00	12:00 - 2:00;	10:00 - 12:0(	2:00 - 4:00	2:00 - 4:00	utxrt; lwtij
27	02:39.4	dINEHJlzNQ	epkll	3	0	4	0	0	1	4	3	3	12:00 - 2:00;	2:00 - 4:00;4	12:00 - 2:00;	2:00 - 4:00;4	12:00 - 2:00;	fvtjy; zfknq
28	02:39.4	SFhGtwaoCi	enkdu	0	1	1	0	0	3	0	4	1	12:00 - 2:00;	10:00 - 12:0(	12:00 - 2:00;	2:00 - 4:00	12:00 - 2:00;	whvkk; ibfll
29	02:39.4	ATYPamblXE	clyrs	1	3	1	2	1	1	5	1	3	12:00 - 2:00;	12:00 - 2:00;	12:00 - 2:00;	None	12:00 - 2:00;	mkwou; ofovu
30	02:39.4	cHibVhLypp	rjnxu	4	2	5	3	0	4	0	5	1	12:00 - 2:00;	None	12:00 - 2:00;	10:00 - 12:0(	12:00 - 2:00;	dqipb; itbot
31	02:39.4	VUsgWgGXF	zzwve	4	2	1	3	2	3	2	1	2	2:00 - 4:00;4	12:00 - 2:00;	12:00 - 2:00;	12:00 - 2:00;	12:00 - 2:00;	prcux; nmcgi
32	02:39.4	ZVkuRptifh	xbdec	4	1	5	5	4	1	0	1	2	10:00 - 12:0(	10:00 - 12:0(	12:00 - 2:00;	10:00 - 12:0(	12:00 - 2:00;	nracl; vdhaf
33	02:39.4	GxzLjcloon	fstsu	5	5	2	2	4	3	0	1	5	None	12:00 - 2:00	10:00 - 12:0(	12:00 - 2:00;	2:00 - 4:00;4	eglbn; gfnya
34	02:39.4	mykADalxcN	cusyd	3	0	5	0	1	2	5	0	0	2:00 - 4:00;4	None	12:00 - 2:00;	12:00 - 2:00;	12:00 - 2:00;	laand; klcqa
35	02:39.4	OeVjgersSi	qvmps	0	4	1	3	4	2	4	1	1	12:00 - 2:00;	12:00 - 2:00;	4:00 - 6:00	12:00 - 2:00;	12:00 - 2:00;	fhidw; cdpjp
36	02:39.4	CycOJWfbUz	qmcax	2	5	5	1	5	4	1	4	1	2:00 - 4:00	2:00 - 4:00;4	10:00 - 12:0(	None	None	prnrl; uwxhk
37	02:39.4	KXUOuguciS	eexay	3	3	4	4	2	5	1	1	2	12:00 - 2:00;	12:00 - 2:00;	2:00 - 4:00	12:00 - 2:00;	None	bmimu; hrrcl
38	02:39.4	wejAyNpYgb	locwe	0	3	1	4	1	0	2	4	4	None	12:00 - 2:00;	10:00 - 12:0(	12:00 - 2:00;	10:00 - 12:0(	pmgby; tajdz
39	02:39.4	JFZzqgXwFq	emakj	1	4	1	0	3	0	2	3	5	12:00 - 2:00;	2:00 - 4:00;4	2:00 - 4:00;4	2:00 - 4:00;4	2:00 - 4:00;4	nmaoo; lbfrd
40	02:39.4	rfXfDIbpOE	xjxjm	5	2	0	5	4	2	0	1	5	12:00 - 2:00;	2:00 - 4:00;4	12:00 - 2:00;	2:00 - 4:00;4	4:00 - 6:00	pmbdy; yteji
41	02:39.4	tPfDvPWzAg	aqprw	0	2	1	0	0	2	0	4	3	12:00 - 2:00;	4:00 - 6:00	12:00 - 2:00;	2:00 - 4:00;4	12:00 - 2:00;	qjolc; sgqul
42	02:39.4	aMDLRNAHV	nmjih	2	2	5	0	4	0	2	1	5	None	10:00 - 12:0(	10:00 - 12:0(	12:00 - 2:00;	10:00 - 12:0(	rnzyy; sreul
43	02:39.4	qqwJBXZoJK	pubml	1	3	3	1	4	1	4	1	5	None	2:00 - 4:00	12:00 - 2:00;	12:00 - 2:00;	None	oymcd; mxpdg
44	02:39.4	mJGHDIWHj	qupoo	2	5	4	0	1	3	4	1	2	2:00 - 4:00	12:00 - 2:00;	2:00 - 4:00	12:00 - 2:00;	10:00 - 12:0(	hdyid; yxzsh
45	02:39.4	AWjBmQizFc	pnbpw	1	1	2	2	1	5	0	5	4	10:00 - 12:0(	12:00 - 2:00;	None	12:00 - 2:00;	10:00 - 12:0(	wotrn; hwdtq
46	02:39.4	fhKMmwToc	zaljb	5	0	5	0	0	3	2	0	5	2:00 - 4:00	10:00 - 12:0(	10:00 - 12:0(	10:00 - 12:0(	None	mnqht; xthrk
47	02:39.4	oDfoibyARI	guzxo	1	4	5	2	5	5	0	1	4	None	12:00 - 2:00;	10:00 - 12:0(	10:00 - 12:0(	12:00 - 2:00;	agpbx; caana
48	02:39.4	maqAOINRal	ztbyr	3	1	1	4	0	1	1	3	2	12:00 - 2:00;	4:00 - 6:00	2:00 - 4:00;4	10:00 - 12:0(	2:00 - 4:00;4	uvwrd; rdjvg
49	02:39.4	tRfXQUAtvj	zqlhw	1	4	4	4	1	1	5	5	4	10:00 - 12:0(	None	12:00 - 2:00;	10:00 - 12:0(	2:00 - 4:00;4	owggm; fxqvz
50	02:39.4	DEEIEDINFI	nonah	5	3	1	5	0	0	1	2	0	10:00 - 12:0(	12:00 - 2:00;	12:00 - 2:00;	12:00 - 2:00;	12:00 - 2:00;	icstw; cybzp
51	02:39.4	XuOnfcEqFK	axanf	0	0	3	2	0	5	4	4	5	12:00 - 2:00;	2:00 - 4:00	10:00 - 12:0(	2:00 - 4:00	12:00 - 2:00;	nnkyr; mdunh
52	02:39.4	DOLErVyMBI	bewtw	5	5	0	0	4	5	4	3	4	2:00 - 4:00;4	10:00 - 12:0(	None	2:00 - 4:00;4	None	rwmsv; lbduc
53	02:39.4	EuEPBdnaSF	jvpff	0	2	1	4	3	4	5	2	0	2:00 - 4:00	10:00 - 12:0(	None	2:00 - 4:00;4	12:00 - 2:00;	oyhve; zkvrt
54	02:39.4	bcDBZBtxbx	hubyl	4	2	2	0	0	4	4	1	3	None	None	10:00 - 12:0(	2:00 - 4:00;4	10:00 - 12:0(	hhjls; syzjn
55	02:39.4	HLoaUfsvLa	cfktd	1	4	3	4	5	1	1	3	5	12:00 - 2:00;	10:00 - 12:0(	12:00 - 2:00;	10:00 - 12:0(	None	gsuol; avvfo
56	02:39.4	uBsZyechMA	iuuiw	2	4	4	5	5	0	3	4	3	12:00 - 2:00;	None	12:00 - 2:00;	2:00 - 4:00	10:00 - 12:0(	vxwxh; qvqqj
57	02:39.4	ZqsMdgRiap	qaprw	5	4	0	2	0	0	2	0	1	2:00 - 4:00	2:00 - 4:00;4	12:00 - 2:00;	12:00 - 2:00;	12:00 - 2:00;	mtsze; pgmbx
58	02:39.4	xigWEaHYeo	nouwl	1	0	0	4	3	2	0	1	3	2:00 - 4:00	2:00 - 4:00	2:00 - 4:00;4	12:00 - 2:00;	2:00 - 4:00	kumsa; syvly
59	02:39.4	HRaRjlbCud	uqvcg	4	5	1	1	1	1	2	0	0	12:00 - 2:00;	12:00 - 2:00;	4:00 - 6:00	2:00 - 4:00;4	10:00 - 12:0(	sywmn; amcze
60	02:39.4	acQzKervOd	nzbyr	1	1	2	1	1	5	3	2	1	12:00 - 2:00;	None	12:00 - 2:00;	4:00 - 6:00	2:00 - 4:00	pezwt; oaiwy
61	02:39.4	oecgDdLZsM	raaot	3	3	1	1	5	3	0	4	0	4:00 - 6:00	10:00 - 12:0(	None	12:00 - 2:00;	12:00 - 2:00;	zlqvs; ppfmd
62	02:39.4	hSvzyYlvwn	yjyhr	2	4	1	2	4	4	2	1	2	10:00 - 12:0(	12:00 - 2:00;	10:00 - 12:0(	12:00 - 2:00;	2:00 - 4:00;4	qvsid; odsrx
63	02:39.4	MyomGiwQz	cnpdc	1	1	1	5	3	3	3	5	4	10:00 - 12:0(	12:00 - 2:00;	12:00 - 2:00;	4:00 - 6:00	12:00 - 2:00;	gilfu; psgyz
64	02:39.4	fpUqLjrllc	lvcdk	3	1	3	0	0	1	2	3	4	12:00 - 2:00;	2:00 - 4:00;4	None	2:00 - 4:00;4	2:00 - 4:00	crilf; xwmyj
65	02:39.4	rdItEXymkz	miybp	1	5	1	5	5	2	0	2	2	12:00 - 2:00;	2:00 - 4:00;4	12:00 - 2:00;	2:00 - 4:00;4	12:00 - 2:00;	sdgse; negcy
66	02:39.4	UCKQRguWH	buyuf	4	5	4	0	3	1	4	3	5	None	12:00 - 2:00;	10:00 - 12:0(	2:00 - 4:00;4	12:00 - 2:00;	cxxwu; wnirf
67	02:39.4	wpWdllfFwY	duixh	1	3	0	3	0	4	5	1	2	None	12:00 - 2:00;	12:00 - 2:00	10:00 - 12:0(	10:00 - 12:0(	iofci; pzekw

Figure 1 shows the CSV file of the literal input into the Team Building program. The software takes in the filename, opens the file, and parses the data. Each row is a student, and each column is a parameter, such as free time availability on Monday or self-rated HTML experience level.

2. Pairwise Testing

Using a pairwise testing tool developed by the author's advisor[8], Michal Young, we can create symbolic test vectors that describe what each test case should generally look like, using English adjectives to describe what the input should be. Pairwise testing rests on the hypothesis that a majority of faults in a program are a result of combinations of parameters that work with one another to create or exacerbate a problem. Creating the minimum number of test cases that satisfactorily describe all combinations of possible parameters creates the maximum code coverage. The success of finding all faults increase with greater numbers of combinations of parameters; that is to say that all combinations of four parameters will find more faults than all combinations of three. However, pairwise testing finds all combinations of two parameters and is often sufficient for maximum code coverage while still reducing the time, and therefore cost, of creating or running all test cases. This is even more likely when the total number of parameters is also somewhat low. Combining only two parameters is a matter of convenience and time, since more test cases require more time to create.

While translating a symbolic test vector to concrete data may present a challenge, it is extremely valuable in reducing the uncertainty of knowing what sort of trend the input test data follows. Say, for example, a tester needs to generate concrete point for this test vector:

Item Purchased	**Price**	**Delivery Method**
Large Item	Expensive	Ultrafast

Table 1: Single Symbolic Test Vector

She knows that some test case will send a large item by ultrafast means. In our generation of concrete data (described in the next section) in this test vector, she might end up with

Motorboat	18700.00	Jet Airplane

Table 2: Single Concrete Test Case

The interaction of these two parameters, a very large object sent by ultrafast means, ought to trigger a bug in the system, because perhaps her company does not have that capability. If this situation is not handled correctly in the software, the user of the software, the customer, will be misled about the quality of their shipping network. This example is very trivial, but it exemplifies the problem well, and shows how pairwise testing methods can bring this awareness to the tester and developer and help them improve their software. Now imagine a much longer and complicated test vector in a very advanced version of the company's ecommerce website; maybe now it has millions of possible item categories and hundreds of shipping methods and can also handle coupon codes and student discounts and special offerings: many more parameters to consider. It is helpful to know that the test vector was created from a description including Large Item - Expensive - Ultrafast. The symbolic test vector describes the input well enough for the tester to understand and use common sense to predict what the output ought to be. Then if she tests that test case and finds that the customer receives a message that sending a boat through lightning speed mail is impossible, she gains more confidence that her program works correctly.

The pairwise testing tool is designed such that the tester describes all the possibilities for what each parameter *could possibly be*, and then the tool generates all the test vectors that describe what the parameter ends up becoming. When using this tool, testers should list every kind of statistical distribution that they would be interested in seeing, or what makes sense in the context of their program.

2. Context-Free Grammars

The aforementioned skipped step uses context-free grammars implemented in the Python templating tool Mako to generate concrete test data. This tool, Makogram, has been developed by the author's advisor[9]. Say a test case has the following form:

Case --> Item + Price + DeliveryMethod

What constitutes an item? Item might have a production rule

Item --> "Ship" or *"Book"* or *"Loaf"*

The grammar tool used is constructed such that a non-terminal can be the result of a function, and that result can be a randomly generated data point. For example,

Price --> lambda x: random.choice([i for i in range(x)])

Meaning that the data point for the Price parameter would be some number chosen at equal randomness from any number between 1 and x.

A terminal symbol can be written to be the returned result of a much more sophisticated function - one perhaps that returns a Gaussian distribution of data points. For example, consider an earthquake modeling program, which takes a set of data points for its magnitudes parameter, and it can have all sort of descriptions attached to it: Gaussian, uniform, cardioid, etc. All of these descriptions will occur somewhere in a test vector, so each is mapped to the corresponding function. When the magnitudes parameter is set to be generated by a bell-curve distribution, the bell-curve random generator will generate that set of points.

Generating good test data for actual magnitudes of an earthquake must be based on the knowledge that a magnitude cannot be negative or greater than 10.0. This is the limit of the program. It cannot make logical guesses about the natural language of the parameter. But the tool is written to minimize the number of times the tester must identify this constraint – just once, in the beginning.

3. Law of Large Numbers

The Law of Large Numbers states that the actual outcomes will approach the expected outcomes as the sample size increases to infinity[10]. One canonical example is a series of coin flips, with an equal probability of flipping heads as tails. It is expected that exactly 50% of the samples will be heads and 50% tails. In a sample size of only one hundred flips, the percentage of heads to tails may only be 46.0% - 54.0%, but a sample size of ten thousand would come far, far closer to an even 50-50 split, possibly 50.37% - 49.63%.

This principle applies to all kinds of probability distribution types, not just coin flip probability. Students' grades tend to figure towards a bell-curve distribution, but a graph of 50 students' grades will look more misshapen than a graph of thousands of students' grades. Moreover, the principle applies to many programming languages' utilities that support randomness. The Python language contains a built in module called random which provides a variety of generators for all flavors of probability distribution. Individual points are generated according to a particular distribution, and the generation of a large enough sample size is guaranteed to follow the expected trend.

How large is large? Bernoulli proved that the actual outcome will approach the expected outcome as the sample size grows to infinity. So in general, the greater the sample size the more accurate the results.

I expect my use case to arrive on the order of hundreds of data points, so I believe a default sampling magnitude on the order of ten thousand randomly generated data points should suffice. However, I have provided options for the user to specify how big the initial sample set is, in which case it is up to them to choose an appropriate magnitude. In either case, the initially generated set of points is far too many, so I have devised a scheme to reduce the sample set to the appropriate size while still encapsulating the distributions guaranteed by the Law of Large Numbers. For each parameter, I generate the too-large sample size, sort it in increasing order, and selectively choose every nth point to include in the reduced set of points, such that the size of the reduced set is the desired number – how every many the test case needs.

This selection scheme is yet to be proven rigorously, but I believe that it suffices for now, and does indeed capture the essence of the law of large numbers. Pictorial representations show that this scheme is not perfect, but is certainly far better than choosing exactly the desired number of points

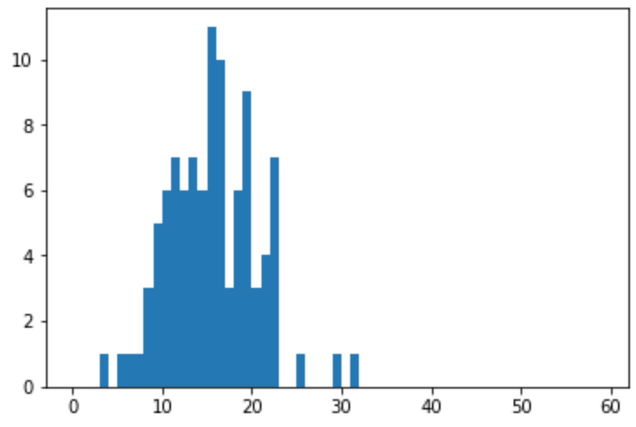


Figure 2 shows the graph of points generated by normal distribution with a sample size of 100

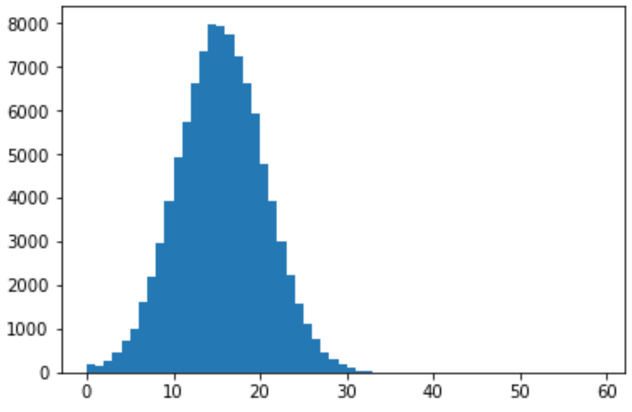


Figure 3 shows the graph of points generated by normal distribution with a sample size of 10,000

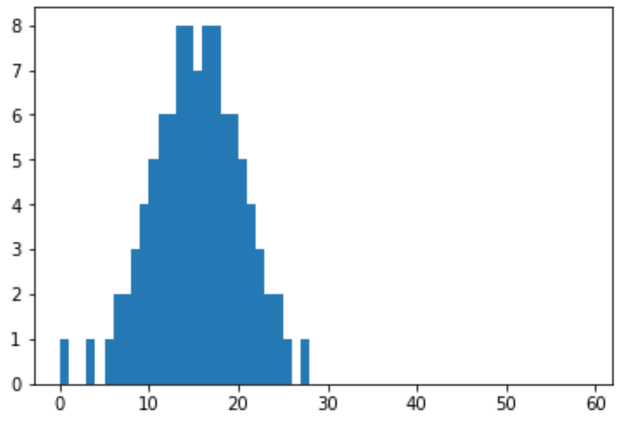


Figure 4 shows the graph of points generated by normal distribution and reduced down to a sample size of 300.

4. Cardioids

In some testing situations, it may be useful to consider the relationship between two parameters, a relationship that mimics real life. For example, a physics-modeling simulation might have a loose property that larger objects move slower than smaller objects. These two parameters, size and velocity are interdependent.

A cardioid object acts as a two-column parameter. The end user specifies the relationship between the two sub-parameters by providing descriptions of what data point range pairings are preferred (called favorites) and what data point range pairings should only occur infrequently (called outliers or non-favorites). The data generation step will look at these descriptions and generate the data point pairings such that 90% of the samples are favorites and 10% are outliers. For example, the tester will specify favorites for the cardioid relationship between size and velocity: large and slow, medium and medium, and small and fast. Non-favorites are outlying particles: small and slow, and large and fast.

The cardioid relationship spanning two parameters occurs only if the tester specifies it as a possibility in the test vector creation step. There are certainly situations where it does not make semantic sense to use cardioid relationships. But, it does allow testers to generate tests that maintain realistic sense. It also allows the user to include their own test data generation constraints, if statistical randomness does not suffice.

Methods

One key part of this project is deciding if it accomplished its goal and answered its research question. Is using random statistical distributions a valid approach to writing test data? The answer to this question must be supported by observations on the performance and quality of GenSequence.

Testers might use this tool at any stage during the software development cycle. Perhaps their application is highly advanced, has been neatly designed, and has fixed bug reports found through other means of testing or from end user reports. Therefore, that piece of software simply may not have that many bugs. That is highly unlikely, but it is possible. So measuring the usefulness of GenSequence on fault finding ability alone would depend on the state of the software it is testing, giving the indication that GenSequence is inadequate when really the software is just really well done.

The intention of this project was to streamline the testing process, and provided automated methods of data creation. The usefulness of this program is measured by the ease of test suite creation. Ultimately it is a tester's decision as to how useful this project is, but these goals are nearly a self-fulfilling prophecy. More features added to the project only increases the ease of use, since a machine does more of the "heavy-lifting" than a human brain otherwise would. Nevertheless, there is further discussion later.

Finally, it is important to determine how valid the random statistical distribution approach is. A few case studies will be used to determine if knowing how a parameter's data was generated helps clarify the expected result. First I will test my testing tool against a planetary orbits simulation. Then I will identify open-source projects that have limited testing framework but are legitimate enough to benefit from test data creation, and generate test data for them using my tool. I will also consider these properties: the length of the script generating data, how much code a tester would have to write, and how quickly the data is generated (more data takes more time).

Results

Still to be determined

Concluding Thoughts

Still to be determined

References

[1] Weyuker, Elaine J. "On Testing Non-Testable Programs". *The Computer Journal* Volume 25, Issue 4 (1 November 1982): 465-470, accessed October 22, 2017
https://doi.org/10.1093/comjnl/25.4.465

[2] Mikael Lindvall, Adam Porter, Gudjon Magnusson, and Christoph Schulze. "Metamorphic model-based testing of autonomous systems". *Proceedings of the 2nd International Workshop on Metamorphic Testing* (MET '17). IEEE Press, Piscataway, NJ, 35-41, (2017).

[3] Sergio Segura, Amador Durán, Javier Troya, and Antonio Ruiz Cortés. "A template-based approach to describing metamorphic relations". *Proceedings of the 2nd International Workshop on Metamorphic Testing* (MET '17). IEEE Press, Piscataway, NJ, 3-9, (2017).

[4] Tsong Yueh Chen. "Metamorphic testing: a simple method for alleviating the test oracle problem". *Proceedings of the 10th International Workshop on Automation of Software Test* (AST '15). IEEE Press, Piscataway, NJ, 53-54 (2015).

[5] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz and S. Yoo, “The Oracle Problem in Software Testing: A Survey”. *IEEE Transactions on Software Engineering* Volume 41, No. 5 (May 2015): 507-525.

[6] Matt Staats, Gregory Gay, and Mats P. E. Heimdahl. “Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing”. *Proceedings of the 34th International Conference on Software Engineering* (ICSE '12). IEEE Press, Piscataway, NJ, 870-880, (2012).

[7] Junjie Chen, Yanwei Bai, Dan Hao, Lingming Zhang, Lu Zhang, Bing Xie, and Hong Mei. “Supporting oracle construction via static analysis”. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (ASE 2016). ACM, New York, NY, 178-189, (2016).

[8] https://github.com/TestCreator/GenPairs

[9] https://github.com/TestCreator/GenSequence

[10] Guttag, John. *Introduction to Computation and Programming Using Python: With Application to Understanding Data*. Cambridge, MA: MIT Press, 2017, pp. 156-157.

Annotated Bibliography

Barr, E.T, and M. Harman, P. McMinn, M. Shahbaz and S. Yoo, "The Oracle Problem in Software Testing: A Survey," in *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507-525, May 1 2015. DOI: 10.1109/TSE.2014.2372785
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6963470&isnumber=7106034
This paper describes different testing techniques and methods of parsing an oracle.

Chen, Junjie, and Yanwei Bai, Dan Hao, Lingming Zhang, Lu Zhang, Bing Xie, and Hong Mei. 2016. Supporting oracle construction via static analysis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (ASE 2016). ACM, New York, NY, USA, 178-189. DOI: https://doi.org/10.1145/2970276.2970366
Static collection of fault-finding variables is a variant of the method that Staats, Gay, and Heimdahl proposed.

Chen, Tsong Yuen. 2015. Metamorphic testing: a simple method for alleviating the test oracle problem. In *Proceedings of the 10th International Workshop on Automation of Software Test* (AST '15). IEEE Press, Piscataway, NJ, USA, 53-54.
A discussion on metamorphic testing practices and applications.

Guttag, John. *Introduction to Computation and Programming Using Python: With Application to Understanding Data*. Cambridge, MA: MIT Press, 2017, pp. 156-157.

Bernoulli proved that the Law of Large Numbers guarantees certain statistical predictions.

Jahangirova, Gunel. 2017. Oracle problem in software testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (ISSTA 2017). ACM, New York, NY, USA, 444-447. DOI: https://doi.org/10.1145/3092703.3098235

A discussion on the benefits of oracle location: inside or outside the source code.

Lindvall, Mikael, and Adam Porter, Gudjon Magnusson, and Christoph Schulze. 2017. Metamorphic model-based testing of autonomous systems. In *Proceedings of the 2nd International Workshop on Metamorphic Testing* (MET '17). IEEE Press, Piscataway, NJ, USA, 35-41. DOI: https://doi.org/10.1109/MET.2017..6

Segura, Sergio, and Amador Durán, Javier Troya, and Antonio Ruiz Cortés. 2017. A template-based approach to describing metamorphic relations. In *Proceedings of the 2nd International Workshop on Metamorphic Testing* (MET '17). IEEE Press, Piscataway, NJ, USA, 3-9. DOI: https://doi.org/10.1109/MET.2017..3

Since metamorphism is thriving, this group attempts to standardize the practice with a template so that practitioners who begin using the technique have a baseline approach that makes the transition easier.

Staats, Matt, and Gregory Gay, and Mats P. E. Heimdahl. 2012. Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing. In *Proceedings of the 34th International Conference on Software Engineering* (ICSE '12). IEEE Press, Piscataway, NJ, USA, 870-880.

By seeding faults in mutant versions of a program, an existing test suite can find the variables most likely to cause faults in a program. By narrowing the size of the expected oracle values, the tester has significantly less work.

Weyuker, Elaine J. On Testing Non-Testable Programs, *The Computer Journal*, Volume 25, Issue 4, 1 November 1982, Pages 465–470, https://doi.org/10.1093/comjnl/25.4.465

This paper defines what it means to be non-testable, and clearly describes the oracle problem and why it is so difficult to provide every expected value in an oracle.