SWEETPEA: A LANGUAGE FOR EXPERIMENTAL DESIGN

by

Anastasia Cherkaev

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

Department of Computer Science

The University of Utah

October 2018

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Anastasia Cherkaev**
has been approved by the following supervisory committee members:

Vivek Srikumar ,	Chair(s)	**22 Oct 2018** Date Approved
Matthew Flatt ,	Member	**22 Oct 2018** Date Approved
Jonathan Cohen ,	Member	**22 Oct 2018** Date Approved
none ,	Member	**22 Oct 2018** Date Approved
none ,	Member	**22 Oct 2018** Date Approved

by **Ross Whitaker** , Chair/Dean of
the Department/College/School of **Computer Science**
and by **David Kieda** , Dean of The Graduate School.

ABSTRACT

The replicability crisis in experimental science is fueled by a lack of transparent and explicit discussion of experimental design in published work. An experimental design is a description of experimental factors and how to map those factors onto a sequence of trials such that researchers can draw statistically valid conclusions. This thesis introduces SweetPea, a SAT-sampler aided language that facilitates creating understandable, reproducible and statistically robust experiemental designs. SweetPea consists of (1) a high-level language to declaratively describe an experimental design, and (2) a low-level runtime to generate unbiased sequences of trials given satisfiable constraints. The high-level language provides primitives that closely match natural descriptions of experimental designs. To ensure statistically significant results, every possible sequence of trials that satisfies the design must have an equal likelihood of being chosen for the experiment. The low-level runtime samples sequences of trials by compiling experimental designs into Boolean logic, which are then passed to a SAT-sampler. The SAT-sampler provides guarantees that the solutions it finds are statistically robust.

For my parents.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

NOTATION AND SYMBOLS

α	fine-structure (dimensionless) constant, approximately $1/137$
α	radiation of doubly-ionized helium ions, He++
β	radiation of electrons
γ	radiation of very high frequency, beyond that of X rays
γ	Euler's constant, approximately $0.577\,215\ldots$
δ	stepsize in numerical integration
$\delta(x)$	Dirac's famous function
ϵ	a tiny number, usually in the context of a limit to zero
$\zeta(x)$	the famous Riemann zeta function
$\ldots$	$\ldots$
$\psi(x)$	logarithmic derivative of the gamma function
ω	frequency

CHAPTER 1

MOTIVATION

A scientific conclusion is only as trustworthy as the experimental design it is based on. Incorrectly designed or biased experiments lead to possibly invalid conclusions; therefore creating correct, statistically unbiased, reproducible experimental designs is paramount to performing meaningful experiments.

1.1 Reliable Experimental Design and Reproducibility Crisis

The replicability crisis in experimental science is fueled by a lack of transparent and explicit discussion of experimental design in published work. While there are many software tools for modeling and running experiments [1] [2] [4], there are, to the best of our knowledge, none for designing them. Currently, scientists design experiments by writing complex scripts which manually balance the experimental factors of interests. There are two major issues with this approach: the first is that it may not (and often does not) produce unbiased sequence of trials. In practice, researchers construct these sequences without any statistical guarantees because the brute force solution for constructing unbiased sequences by enumerating all options is intractable; for a typical experiment the size of the search space is 10^{100}. The second issue is that this approach is brittle. It is easy to introduce bugs that go unnoticed but which may have large consequences, and it is difficult to verify and reproduce another researcher's implementation.

SweetPea can be viewed as a domain-specific interface to SAT-sampling, and while there are other languages that rely on SAT-solvers [5], none that we know of leverage the guarantees provided by SAT-samplers. To ensure statistically significant results, every possible trial sequence that satisfies the constraints must have an equal likelihood of being chosen for the experiment. This guarantees that the method for generating trial sequences is not introducing bias. In practice, however, researchers construct these trial sequences without statistical guarantees. The number of valid sequences is both intractably large and sparse in the space of all sequences, so it is not possible to find a valid sequence by randomly sampling all sequences or by enumerating all valid sequences.

The runtime to generates unbiased sequences of trials given satisfiable constraints. At the heart of the bias problem is the need to sample from constrained combinatorial spaces with statistical guarantees; SweetPea samples sequences of trials by compiling experimental designs into Boolean logic, which are then passed to a SAT-sampler. The SAT-sampler Unigen provides statistical guarantees that the solutions it finds are approximately uniformly probable in the space of all valid solutions. This means that while producing sequences of trials that are perfectly unbiased is intractable, we do the next best thing– produce sequences that are *approximately* unbiased.

1.2 Declarative Programming: Science without the Engineering Burden

Virtually all fields of scientific research increasingly rely on computational tools. Computational tools open the door to analyses that are otherwise intractable, time consuming and error-prone. Moreover, writing programs contributes to making research reproducible because it creates digital artifacts like files, which allow one scientist to share, analyze and run a program written by another. The drawback, however, is that writing correct, maintainable, complex programs takes significant engineering effort. Requiring scientists to be engineers in addition to being highly trained domain specialists needlessly impedes the progress of science. Declarative languages allow their users to describe the result they want, as contrasted with imperative languages which require their users to describe the how to construct the result.

1.3 Requirements Statement

There is a need for a software system that allows domain scientist to design unbiased, replicable experiments. Moreover, this system needs to provide an easy-to-use, declarative interface so that scientists who are not necessarily trained as software engineers can create and reason about complicated experimental designs, and transparently share their experimental setups and design choices. SweetPea is just such a system; it is a language which provides semantics for describing experiments, a runtime for synthesizing experimental sequences from specifications, and a set of tools for debugging over-constrained designs.

While the need for a system to automate experimental design is general to many types of science, we have built a prototype targeted for psychology and neuroscience, where issues of reproducibility and complexity of design have become a focus of attention.

CHAPTER 2

SWEETPEA OVERVIEW

The previous chapter motivated the need for a software system that facilitates creating replicable, statistically robust experimental designs; this chapter provides a high-level overview of how SweetPea addresses this need.

SweetPea consists of a high-level language and a low-level runtime. The language provides primitives that closely match the terms that scientists use to describe their experimental designs. The runtime synthesizes an experimental sequence which is guaranteed to not prefer any valid solution over any other. The runtime provides this guarantee by representing the experiment as a boolean formula and interfacing with a SAT sampler; this constrains the language to be amenable to being translated into a boolean formula.

In this chapter we'll start by looking at a simple version of a classic psychology experiment, the Stroop test, to identify the fundamental components of an experimental design. Then we'll see how these components are represented in the language and the runtime.

2.1 Running Example: the Stroop Experiment

The Stroop experiment is a well-known psychology experiment, originally published by John Stroop in 1935. A subject is shown a stimulus, and asked to perform a task based on some property of the stimulus; the researcher measures how the subject's reaction time varies depending on the stimulus. One version of this experiemnt involves showing subjects a word printed on a slide and asking them to say the color of the word. All of the words are the names of colors, such as the text 'red' and 'blue'. Some of the stimuli are congruent, meaning that the color of the ink matches the text, such as the word red printed in red ink. Other stimuli are incongruent, such as the word red printed in blue ink (figure **Figure 2.1** on page 6). The Stroop effect is the observation that subjects have a longer reaction time when the stimulus is incongruent.

Let's consider the smallest version of the Stroop experiment, where the stimuli consist of two colors. Each stimulus is specified by independent and control variables, called *factors*: ink color and text. Each factor has two *levels*, red and blue. Figure **Figure 2.2** on the next page shows the *full crossing* of these factors for this simple case, for a total of 4 possible stimuli. For reference, real experiments have on the order of 5 to 8 factors with 2 to 4 levels, leading to tens to hundreds of possible stimuli. The *design* of the experiment is the list of all factors that describe each stimulus. The design of the experiment may contain factors that are not present in the full-crossing.

Each subject is shown an ordering of the possible stimuli. The researchers may want to place additional *constraints* on the ordering, such as first familiarizing the subject with the task by showing some number of congruent stimuli before showing them a mix of congruent and incongruent stimuli. For our small example, let's consider the constraint that there should be no repetitions of stimuli whose text is the same. The ordering in **Figure 2.2** on the following page is a valid ordering which satisifies these constraints, but swapping the order of the first and second trial would produce an invalid ordering under those constraints.

To prove that we do not introduce bias because of the way we construct experimental sequences, we would like to ensure that each valid sequence is equally likely. In this example, we have four stimuli so there are 4 factorial = 24 possible orderings. Of those 24 orderings, 8 satisfy the constraints as shown in **Figure 2.3** on page 7, and to provide this guarantee we want each of those 8 to be equally likely.

2.2 A Language for Experimental Design

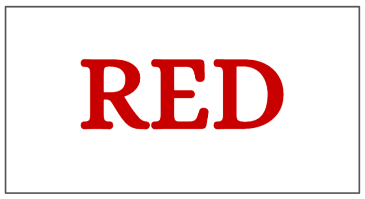
Let's see how the simple Stroop experiment can be represented in SweetPea, and then at how that representation can be translated to a boolean formula to generate an experimental sequence.

The version of the SweetPea language we'll discuss is embedded in Python, so uses Python syntax.

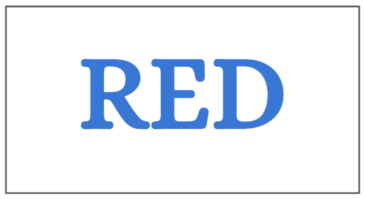
First, we represent the factors directly as a list of levels:

```
ink_color = ("ink_color", ["red", "blue"])
text      = ("text",      ["red", "blue"])
```

Next, let's represent the constraint that there should be no repetitions of stimuli whose

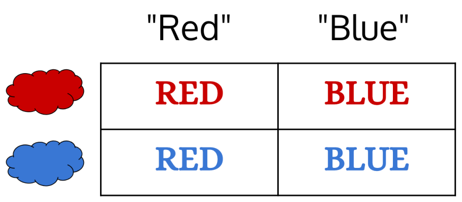


(a) A congruent Stroop stimulus.

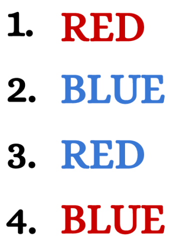


(b) An incongruent Stroop stimulus.

Figure 2.1: Example Stroop stimuli.



(a) The full crossing of possible stimuli.



(b) A possible ordering of stimuli.

Figure 2.2: All stimuli and a possible ordering.

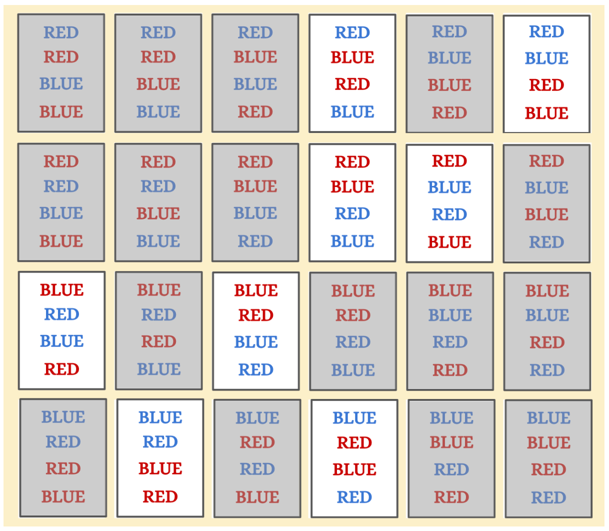


Figure 2.3: Of the 24 possible orderings, the highlighted 8 satisfy the constraints.

text is the same:

```
def no_text_reps(text0, text1):
    return text0 != text1

constraints = enforce(Transition(no_text_reps, [text, text]))
```

TODO mention ENFORCE– or specify a factor with only one level??

does this work? who knows? what's the API for specifying counting contraints?

Having specified the factors and the constraints over those factors, we can now specify the rest of the experimental design:

```
design = [ink_color, text]
crossing = design

experiment = full_crossing(design, crossing, constraints)

synthesize_trials(experiment)
```

In this example, the design and the full crossing are the same, though that doesn't need to always be the case. We specify the experiment to be the full crossing, but more generally we can construct experiments out of multiple ”blocks” of crossings. The call to synthesize_trials translates the experiment to a boolean formula representation and calls the SAT sampler.

2.3 A Runtime for Uniform Sampling

A *boolean formula* consists of *boolean variables* combined using boolean operators, such as AND, OR and NOT. A *satisfying assignment* is a specification of True or False to each boolean variable which causes the entire formula to evaluate to True. Some formulas are unsatisfiable.

An example of an unsatisfiable formula is (A AND (NOT A)). Regardless of whether A is True of False, this formula will always evaluate to False. In contrast, the formula (A AND B) is satisfied by the assignment A is True and B is True, because (True AND True) evaluates to True.

How do we represent the program above as a boolean formula?

First, we represent each level of each trial as a boolean variable, which corresponds to whether or not that level is chosen for the produced experimental sequence. For our Stroop example, there are 4 trials, each of which have 4 boolean variables corresponding to each of the levels for a total of 16 boolean variables. A real experiment will have on the order of hundreds to thousands of boolean variables. These level variables are bound by additional constraints such as that only one level per factor is true, as well as constraints that say how many instances of a level should exist in a given experimental block.

For our Stroop example, this means that for each trial we create boolean variables to represent ink_color=red, ink_color=blue, text=red, text=blue. We create boolean formulas for each trial which say that exactly one of ink_color=red or ink_color=blue must be true, and that there must be two ink_color=red's in the fully crossed block.

We have multiple strategies for encoding constraints, but for now let's say we have a strategy for encoding constraints of the form "no more than K in a row". We can use this to encode our constraint that there should be no repetitions of stimuli whose text is the same.

Once we've created this boolean formula which represents all the relationships which the levels must fulfill, we can pass this formula to the SAT sampler. If there exists an assignment of the boolean variables which satisfies all the constraints, the sampler will return one such assignment. If there are no satisfying assignments the sampler will state that the formula is unsatisfiable.

The solution that the sampler finds is guaranteed to be approximately uniformly probable in the space of all possible solutions. We currently use the SAT sampler Unigen, but providing this guarantee is the goal of all uniform sampling. The sampler provides this goal by using a special family of hash functions called universal hash functions. For our Stroop example, this means that each of the 8 valid orderings in **Figure 2.3** on page 7 should be approximately equally likely.

If the sampler finds a satisfying assignment to the variables, we can then translate those variables back to the levels they represent. For our Stroop example, this means we will get assignments to the 16 boolean variables, 4 of them for each trial. These will determine the values of each of the trials, and because we have satisfied all the necessary constraints, results in a valid ordering of the trials.

CHAPTER 3

RELATED WORK

- there are three related areas: work related to the pyscology computation and experimental design, work related to domain specific and solver-aided languages, and work related to sampling combinatorial spaces.

3.1 Psychology Toolboxes

- psyScope [1]

- psychoPy [2]

- OpenSesame [4]

3.1.1 Reproducibility Crisis

- TODO: surely something goes here

3.2 Domain Specific Languages

- probably don't need to cite anything here?

3.2.1 Solver Aided Languages

- rosette [5]

- sketch: "Domain-Specific Symbolic Compilation"

- dafny maybe

- hyperkernel: co-designing a language and the verification

3.3 Combinatorial Search Spaces

- finding solutions in a large search space– no really, very large

- how large?

- so large

- what is the nature of our search constraints? things like 60 red words, 60 blue words (in Stroop, see chapter 2).

3.3.1 Sampling Methods

- sampling is the problem of finding solutions

- could try solutions at random: turns out they are sparse (most examples don't have 60 red, 60 blue)

- could try to generate all solutions: turns out there are too many (lots of possible arrangements)

- could use MCMC, but doesn't provide guarantees

- this project is *really* about providing this guarantee that we're not introducing bias because this is a huge deal

3.3.2 Boolean Satisfiability

- SAT is a classic problem, NP-complete

- SAT solvers are really efficient solvers

- To specify something in SAT, you use variables and specify invariants

- the SAT solver finds an assignment that satisfies the invariants

- we use a SAT sampler which finds multiple assignments, and with guarnatees

- an alternative is using SMT constraints

- we compile to SAT because unigen is available; to use a different tool we could pretty easily swap out the backend

3.3.3 Uniform Sampling

[3]

- what problem is it solving? want uniform for coverage

- who else cares about this problem

- how is it solved: universal hash functions

- what alternatives exist

CHAPTER 4

SWEETPEA LANGUAGE

The goal of the SweetPea language is to have semantics that match the terms researchers use to describe their experiments, while also being amenable to being translated efficiently to SAT.

- summary here

TODO: everytime I saw ”sometimes” come up with an example.

4.1 Components of an Experiment

To motivate our choice of primitives, we will first look at the components of an experiment.

4.1.1 Descriptions of Stimuli

- Usually factor with discrete levels

- experiments typically have 2-7 factors with 2-4 levels each: important because it means a large search space

- sometimes they might be nested (light color, dark color)

- sometimes want to sample a continous distribution

- sometimes don't fully cross all of them because can't keep the person there that long, but really want to try all combinations

4.1.2 Ordering Constraints

- need to specify the ordering to run Experiments, ie if you're testing the effect of the presence of A, you better be able to run it with and without A etc

- really these are about relationships (presence or absense) of levels or factors, or arbitrary nestings. (1) congruence and (2) transitions as examples of derived levels

- sometimes about relationships of other relationships! like, you can imagine balancing transitions

- usually the complexity is limited by experimental limits, ie people actually have to complete these

4.1.3 Experimental Design and Balancing

- often want a full-crossing

- sometimes experiment is too large for full crossing, so while experiment has over attributes (factors) they might not appear in all examples

- sometimes its impossible to fully-counterweight and it'd be awesome if the tool could tell you if you were trying to do something impossible

- sometimes can fix impossible by weighted crossing

- sometimes can fix by "near balancing" or toss-away block, ie balancing transitions

- ultimate declarative would be just say "these are what I want to analyze, balance for me"

4.1.4 Experimental Structure

- Experimental structure (ie multiple blocks)

- sometimes want prologue / epilogue blocks

- sometimes need these to balance transitions

- sometimes have an experiment that consists of multiple experiments

- sometimes want to reason between subjects because space is so large

4.2 SweetPea Primitives

- for all: how does it map onto the pysch and why is it amenable to SAT

- details about SAT encodings in the next section

4.2.1 Factors and Levels

- possibly nested lists; that is to say trees

- an experiment has one option for a level on at a time

- we can represent whether a level is on or not as a boolean variable, then have a boolean constraint that says that only one can be on at a time

- a nesting isn't represented in the boolean logic, it's just a reference to arbitraty groupings of levels that can be used in the relationships; can build a factor directly or from these groups. Q: what if overlapping level in multiple groupings? probably need to define it as a level then construct multiple references.

- don't currently support sampling factors that are cont distributions because it's challenging to translate that to discrete boolean SAT

4.2.2 Derived Levels

- how do we represent these relationships?

- because they all have to do with ordering, let's consider a "window". Windows have a width and a stride.

- example: transitions

- example: congruence

- windows implicitly "select" and group levels.

- we can then define functions. they are allowed to depend on the state (on or off) of levels, and use this state to say whether or not *they* are "on". Example, congruence. In this way, they're defining new levels (since a level is just a thing that knows whether it's on or off) – which is why they're derived factors.

- these should allow us to define things that skip elements, like the "bait" example

- why is this amenable to SAT? we've got discrete boolean elements whose state depends only upon the state of other boolean elements.

- an advantage is that Levels can have any [printable] type (can be strings, numbers, etc) and then you can build your derivation functions however you like based on those properties. maybe list an example with numbers.

- two options: you can either define a factor with a partition (levels are "where the func is true" and "where the func is false") or by defining and combining levels

4.2.3 Experimental Design, Balacing and Experimental Structure

- really straightforward. An experimental sequence is a list of experimental blocks. An experimental block is at the high level these derivations and at the low level linearized into a list of SAT that the runtime can execute. A design is literally which factors are visible and for balancing we currently only support full-crossings.

- we'll discuss how these are represented in SAT in the next section

CHAPTER 5

SWEETPEA RUNTIME

- chapter summary here

5.1 Conjunctive Normal Form (CNF)

A boolean formula consists of boolean variables combined using boolean operators, such as AND, OR and NOT. There are many canonical forms for boolean formulas; the one that SAT solvers commonly use is Conjunctive Normal Form (CNF). CNF is an ”and of ors”. This means that CNF is built out of OR clauses, which are clauses in which the boolean OR operator is applied to a list of variables. A or B or (not C) or D is an example of an OR clause. CNF is then the AND operator applied to a list of OR clauses. (A or B) and (C or not D) is an example of a boolean formula in CNF.

SAT solvers typically process boolean formulas in CNF. This is partially because they are amenable to common search strategies that solvers use in search of satisfying assignments, and partially because there is an efficient translation from a boolean formula in any form to the CNF canonical form.

More specifically, solvers typically process boolean formulas in the DIMACS CNF form. In the DIMACS convention, boolean variables are denoted by their index. This means that a formula like (A or B) and (C or not D) is represented as (1 or 2) and (3 or -4). This index based renaming is very convinient as a convention for generating fresh variable names– you just need to increment the variable counter.

- variables are index based (important for examples)

- keep track of : 1) variables which need assignemnts, 2) boolean formulas using those vars

- CNF is standard because it makes search easy ; what is it

- can efficently (how efficeintly) translate into CNF

5.2 Representing SweetPea Primitives in CNF

- section summary here:

- mention counting constraints in two flavors

5.2.1 Representing Levels and Factors

- as mentioned in the last chapter, levels correspond naturally to boolean values

- need additional constraint; one hot encoding, exactly one true at a time. literally represent as (a and not b) or (not a and b) – this would be a bad choice if there were many levels but given the experimental size this is the best decision.

- walk through an example

- also need ”sum” constraints. These also count up, but we expect these to be much bigger numbers, ie in the experiment w/ 7 factors w/ 2 levels, there are 64 of each level. So these we compile to ”counting constraints”. Basically these are linear inequality. We compile them to SAT efficeintly through the Tsietin transform (see next chapter for details).

- walk through an example

5.2.2 Representing Derived Levels and Derivation Functions

Internally, we pick out the levels, create new levels whose value depends on a boolean relationship of those levels. The relationship is then defined by a literal truth table: - generate a truth table for the defined function by: - each input is a factor - then take all boolean combos and literally run it - that generates a truth table - encode that truth table in the logic by translating to CNF

- Work an example, for instance congruence.

5.3 Communicating with the SAT-Sampler

- block diagram of all runtime

- variables marked as ”important” vs aux

- translate output to be human readable; easy to integrate with other environments

5.4 Correctness Guarantees

- cite guarantee from unigen

- postulate why this is preserved: TODO

CHAPTER 6

IMPLEMENTING SWEETPEA

- chapter summary

6.1 Language Implementation: Decisions and Alternatives

- section summary

6.1.1 Embedding Language

- was in haskell

- now is in python

- in the future, perhaps in racket for macro options

- why is this important / consequences to these choices

6.1.2 Internal Representations

- pros / cons of using one-hot vs binary : trade-off between more variables and more clauses

- todo: more examples

6.2 Runtime Implementation: Tsietin Transform

- necessary to efficeintly encode on the scale of 60 of these 120 boolean vars need to be true

- how efficient is it?

- works by mimicing popcount circuitry

- generates a bunch of ”junk variables”

6.2.1 Binding Variables: Iff

- new variable ”equals” if their values are in lockstep

6.2.2 Adders

- half adder example

- full adder example

6.2.3 Ripple Carry Adders

- multiple options here, the one I went with

- trade-offs of options

6.2.4 Pop Count Circuit

- pop count circuit example

6.2.5 Exhaustive Testing

- example of input / output: see how it's prone to error

- tested for a given size all assignments

CHAPTER 7

COMPILING STROOP IN EXCRUTIATING DETAIL

- chapter summary

- maybe delete this chapter?

- maybe walk through small Stroop top-to-bottom with all the details

- and then look at the distribution

7.1 Stroop in SweetPea

- example code listing

- it'd be nice to have a simple derivation function

- show how these get translated on the high level (ie, a list: color one hot constraints, text one hot constraints, color sum constraints)

7.2 Compiling Stroop to CNF

- show how the high level constraints get translated to low level constraints

- explain one of them

- show full DIMACS file

7.3 Synthesizing an Experimental Sequence

- show input to unigen and output

- show how that output is translated back to human-readable

7.4 Verifying the Uniformity Guarantee

- run it a bunch and histogram results

CHAPTER 8

FUTURE WORK

- chapter summary

8.1 Beyond Psychology

- other science domains and what would need to change

8.2 Future Language

- section summary: discussed wishes in chapter 2 section 1, these are the ones that are currently unimplemented

8.2.1 Weighted Crossings

- discussed weighted crossings as a necessary component of experiments in chapter 2; not yet implemented

8.2.2 Sampling Continuous Factors

- this is challenging because how does this get translated to SAT?

8.2.3 Automated Experimental Design

- ANOVA experimental design

- need to make sure that this is correct in the domain in many different flavors of experiment

8.2.4 Syntactic Sugar

- don't have to write the name of the factor when the level name is unique

-

8.3 Future Runtime

- section summary

8.3.1 Verified Core

- the motivation for SweetPea is that it will make it easy to write correct experiments; that only holds if sweetpea doesn't itself have bugs.

- the tsietin transform is ripe for bugs because it's doing a non-human-readable transformation

- would be cool to formally verify that the transformations are correct

8.3.2 Debugging unSAT experiments

- why is this a problem: usability: if it's unSAT it just say unSAT but not why. Not very useful.

- can get "minimally unsat core" from SAT solver, maybe can translate that back to user-defined levels to guess what the problem is

8.3.3 Iterative experimental design and Partial Satisfiability

- really want to know if experiment is over-constrained

- maybe could try specify subsets to try to iterately find the most constraints that can be simultaneously satisfied – solvers let you push / pop clauses

8.3.4 Optimizations

- xor constraints

- truth table simplification (QuineMcCluskey)

- choice of SAT encodings and variable v. clauses

CHAPTER 9

CONCLUSION

- Problem we tried to solve
- Why it’s important
- What we did
- What are the consequences + forward looking

REFERENCES

[1] J. Cohen, B. MacWhinney, M. Flatt, and J. Provost, *Psyscope: An interactive graphic system for designing and controlling experiments in the psychology laboratory using macintosh computers*, Behavior Research Methods, Instruments, & Computers, 25 (1993), pp. 257–271.

[2] S. Mathôt, D. Schreij, and J. Theeuwes, *Opensesame: An open-source, graphical experiment builder for the social sciences*, Behavior research methods, 44 (2012), pp. 314–324.

[3] K. S. Meel, M. Y. Vardi, S. Chakraborty, D. J. Fremont, S. A. Seshia, D. Fried, A. Ivrii, and S. Malik, *Constrained sampling and counting: Universal hashing meets sat solving.*, 2016.

[4] J. W. Peirce, *Generating stimuli for neuroscience using psychopy*, Frontiers in neuroinformatics, 2 (2009), p. 10.

[5] E. Torlak and R. Bodik, *A lightweight symbolic virtual machine for solver-aided host languages*, in ACM SIGPLAN Notices, vol. 49, ACM, 2014, pp. 530–541.