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

November 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)	**26 Nov 2018** Date Approved
Matthew Flatt ,	Member	**26 Nov 2018** Date Approved
Jonathan Cohen ,	Member	**26 Nov 2018** Date Approved
none ,	Member	**26 Nov 2018** Date Approved
none ,	Member	**26 Nov 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 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 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; I'm sorry this thesis doesn't contain a single theorem.

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.

This thesis describes SweetPea, a system for describing experimental designs and generating experimental sequences that satisfy the design in a statistically rigerous way. SweetPea's high-level language allows scientists to declaratively describe the experiment they want to conduct rather than forcing them to mechanically describe how to construct their experimental sequences. This allows scientists to write concise, correct programs which describe their experiments, and produce unbiased experimental sequences; these programs can then be published and shared to document the experiment and facilitate replicability.

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.

SweetPea 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 [3] 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.

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.

SweetPea is a vision of what this needed system could be; it is still a work in progress. This thesis documents the motivation, goals, and current state and future vision for SweetPea.

CHAPTER 2

SWEETPEA OVERVIEW

SweetPea is a software system for creating replicable, statistically robust experimental designs. SweetPea consists of a high-level language and a 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.

The ultimate vision for SweetPea is for the researcher to declaratively describe the analysis they wish to perform, and for the system to automatically derive the constraints required to produce an experiment for the desired analysis. Further, the sytems would ideally interactively iterate with the user to let them explore exactly what they wanted to analyze, and how choices they make effect the feasibility of the experimental design.

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

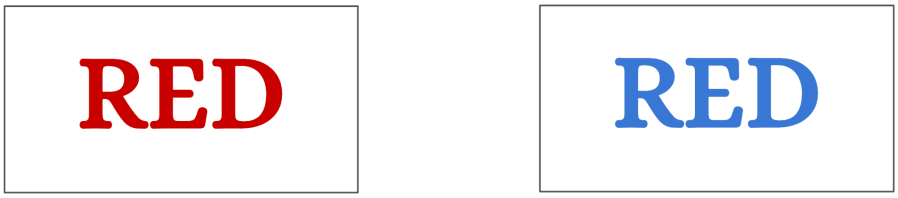
To prove that we do not introduce bias because of the way we construct experimental sequences, we would like to ensure that each valid sequence is equally likely. In this example, we have four stimuli so there are 4 factorial = 24 possible orderings. Of those 24 orderings, 8 satisfy the constraints as 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.

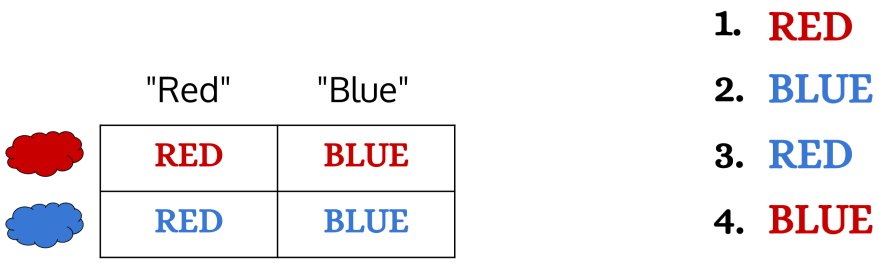
The version of the SweetPea language we'll discuss is embedded in Python, so uses Python syntax.

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



(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.

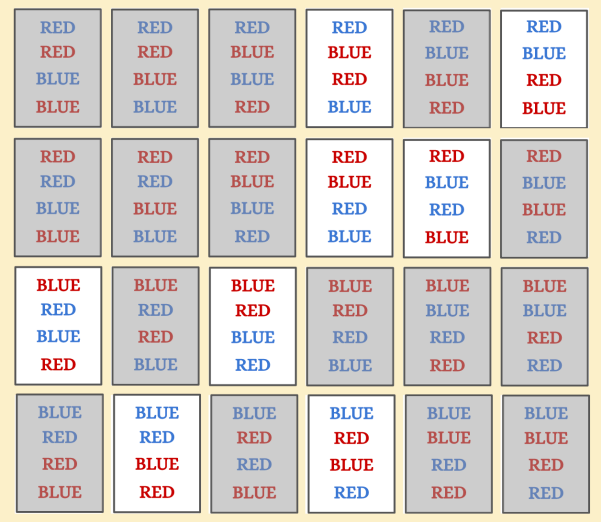


Figure 2.3: Of the 24 possible orderings, the highlighted 8 satisfy the constraints.

```
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 text is the same:

```
k = 2
constraints = [NoMoreThanKInARow(k, ("text", "red")), NoMoreThanKInARow(k, ("tex
```

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.

We'll revisit this example in chapter 7 to see an end-to-end encoding of the Stroop experiment, as well as the sequences produced by SweetPea.

CHAPTER 3

SWEETPEA LANGUAGE

SweetPea achieves its goal of creating unbiased experimental sequences from high-level experimental designs by relying on a SAT sampler. Writing experimental designs in the language of a SAT sampler– that is, as a boolean formula– is highly uncomfortable. SweetPea, therefore, conviniently bridges the gap between scientists and SAT Samplers by creating a wormhole: scientists describe their experimental designs in the terms they usually use in their domain, and SweetPea translates that high-level specification into an enormous low-level boolean formula. The SweetPea runtime then passes the SAT Sampler UNIGEN that formula, and translates the low-level assignments into a high-level experimental sequence which satisfies the design. That experimental sequence is guaranteed to be approximately as likely as any other sequences which fulfills the design, reassuring scientists that they did not accidentally introduce bias because of the way the sequence was constructed.

In this chapter, we'll document the features of the SweetPea language. They largely follow the components we saw in the Overview chapter– experimental designs consist of trials described in terms of factors and levels, and constraints and relationships over those trials, described here in terms of windows and derivation functions, and counting and balancing constraints. Finally, the full experiment is described in terms of a design, a crossing and experimental blocks. Here we'll look at a variety of experiments which motivate these various features, and the high-level language we provide to describe those experiments. In the next chapter we'll take a close look at how these features are efficiently encoded into a boolean formula.

3.1 Components of an Experiment

We briefly saw the parts of an experiment with the Stroop test in Chapter 2. Now let's look at the range of experiments we wish to model, and how our representations of these

experiments are amenable to SAT.

3.1.1 Descriptions of Stimuli

TODO diagrams

Each component of a stimuli is represented as a *factor* which can take on a disrete number of *levels*. In the Stroop experiment this was represented as the ink color being red of blue, and the text color being red or blue.

Experiments typically have 2 to 7 factors with 2 to 4 levels each, resulting in roughly 50 to 200 stimuli. Often the ideal goal is to *fully cross* all of the factors to produce all possible combinations of stimuli, but sometimes factors are left out of the full crossing. For example, a subject can only reliably be tested for so long before they are tired, so sometimes factors which are deemed to be irrelevant to the control variables and variables of interest are excluded from the full crossing.

Levels may be nested into sub-levels. One reason to want to do this is to apply different constraints to each of the sublevels. For instance, a color factor may have nested sublevels of light colors (like yellow and pink) and dark colors (like navy and maroon).

Sometimes researchers may want to define factors with levels sampled from a continuous distribution, for example to model a spectrum of light or sound. Sweetpea currently does not support this other than by the research manually discretizing the continoum into several discrete values. This case is particularly challenging for sweetpea because it is difficult to phrase sampling a continious distribution as a boolean formula; for more thought on how we could better support this see Future Work.

3.1.2 Ordering Constraints

Researchers must have some control over the ordering of stimuli to be able to test their hypothesis. For example, if you are testing whether the presence of a certain level effects the subject's reaction time, you need control over when that level occurs and how often it occurs in relation to other levels. More generally, you need to be able to constrain the presence or absense of relationships which you define over the levels. Since levels can be grouped into arbitrary sub-levels, you should more generally be able to specify constraints on relationships over the sub-levels.

TODO: about diagrams

One example of such a relationship is a *transition* constraint, which specifies whether each level is followed by the same level (a repetition) or a different level (a switch). A common type of transition constraint is the specification that a certain level shouldn't appear more than k-times in a row.

Another example is a *congruence* constraint, which specifies whether the levels of different factors are, according to a user-specified metric, complementary or conflicting. A congruence constraint for the Stroop is whether the color of the ink and the text are the same (the word "red" in red ink) or not.

Another common relationship is a *balancing* constraint. A pair of levels is balanced if each instance shows up the same number of times; a full crossing of all levels is a fully balanced experiment.

Constraints can also be defined in terms of other constraints. For instance, in the Stroop test one may want to balance the number of congruent and incongruent levels, or to balance the transitions between congruent and incongruent levels.

More generally, new experiments may need to define relationships between levels that other existing experiments don't have. SweetPea must have enough flexibility to allow researchers to define these new relationships.

3.1.3 Experimental Design and Balancing

Experiments which include every possible combination of levels are *fully crossed* designs. Fully crossings are desirable because they allow the researcher to test the full space of possible stimuli.

Sometimes, however, an experiment is either too large or too small. An experiment can be too large if it has too many elements in the full crossing because there are limits on how many tasks a subject can perform in a single session reliably. If an experiment is too large it can either be restated as a full crossing of only a subset of the factors (with the non-crossed factors being assigned at random, with uniform probability), or it can be divided across subjects or across experimental sessions. An experiment can be too small if the full crossing has too few stimuli to reliably draw conclusions. In this case, the experiment can either be multiple full crossings back-to-back, or the contents of multiple full crossings combined into a larger experimental block.

Fully-crossing a design is a form of balancing– it is the specification that each level occurs as frequently as each other level and that each level occurs once.

Sometimes the constraint of using a full-crossing makes other types of balancing impossible because the levels are over-constrained. Consider, for instance, the Stroop test with 3 different colors instead of two: then it isn't possible to both fully-cross the design and to balance the congruent and incongruent levels. (This is because there are 3 congruent levels, and 6 incongruent levels). One possible desired outcome in this case is for SweetPea to report that an experiment is over-constrained.

Another alternative is that if balancing the congruent and incongruent levels is more important than a fully-crossed design, then we can instead create a *weighted crossing* instead. In a weighted crossing we can specify that we wish to under-sample certain levels (the incongruent ones in our example) or over-sample other levels (the congruent ones). Weighted crossings also allow us to express the experiments mentioned above, where the full crossing has either too many or too few stimuli to be practical.

3.1.4 Experimental Structure

An experiment is composed of one or more experimental blocks; an experimental block is a sequence of stimuli. Experiments may be organized into blocks for several reasons: there may be prologue or epilogue blocks which set up some condition for the subject, the experiment may be designed to be run in several related sessions, or consist of several sub-experiments.

It is impossible to balance some commonly desired constraints with a single block; for instance, it is not possible to have as many "repeat" transitions as "switch" transitions in the fully-balanced Stroop experiment because there are an even number of stimuli, resulting in an odd number of transitions. This means it's not possible to have as many of one transition as the other.

There are two approaches to resolving this issue. The first is nearly-balancing the levels: instead of requiring that there are exactly the same number of each level, require that there are the same number plus-or-minus one. The other solution is to create a "throw-away" block with a single stimuli in it which is used to satisfy the transition balancing constraint without being included in the full-crossing constraint.

As mentioned in the previous section, it is also sometimes useful to have multiple blocks to repeat stimuli for an experiment with too few stimuli or to break up an experiment with too many stimuli across several sessions or subjects.

3.2 SweetPea Primitives

Now that we've seen a range of experiments that we wish to represent, let's see how we can represent them in SweetPea. Here we'll look at the high-level encodings we use, and why those encodings are amenable to being represented in SAT. In the next chapter we'll look at the SAT encodings in more detail.

3.2.1 Factors and Levels

Factors have names, and a possibly nested list of levels, each of which also has a name. Names are used to specify the experimental sequence that the runtime eventually returns, but they do not need to be strings. They can be any "printable" data-type; this can be useful because names are referenced in user-defined constraints as we'll see in the next subsection.

```
example_factor = ("example_name", ["example_level"])
boolean_factor = ("booleans", [False, True])
text       = ("text",       ["red", "blue"])
light_colors = ("light_colors", ["pink", "aqua"])
dark_colors = ("dark_colors", ["navy", "maroon"])
ink_color = ("ink_color", [light_colors, dark_colors])
```

In the resulting experimental sequence, exactly one choice of level is selected per trial. Internally, the experimental sequences is represented as [TODO DIAGRAM] a list of trials, where each trial is represented as a list all possible levels. There are multiple possible encodings, but the one we chose here is a "one-hot" encoding, where each level is a boolean variable which indicates whether or not it is chosen. All the boolean variables corresponding to levels of a single factor can then have an additional constraint applied that exactly one must be true. This has the effect of "chosing" one level for every factor in the trial, thereby specifying the trial. Because we do this for every trial, a satisfying assignment to these constraints represents a valid experimental sequence. See [diagram] for an example, and see Chater 6 for a more detailed discussion on the trade-offs of the choice of this, and other, encodings.

Nested levels are handled in exactly the same way- the nesting is just a way to reference a grouping in the high-level representation. This is useful because it can be used to talk about relationships between groups, ie, in constraints. At the boolean representation level, however, these groupings are irrelevant and the structure is flattened.

3.2.2 Deriving Levels with Windows

TODO COUNTING CONSTRAINTS

How do we represent relationships between levels and enforce constraints over them? All of the constraints we are considering relate to order, so let's consider a symbolic window which slides over the list of trials. A window has a width and a stride. The width specifies how many trials to consider at once; the stride specifies how many trials to move forward by when the window moves. When combined with user-defined functions, windows allow the user to specify new "derived" levels and factors based on the values of the levels within the window. See the diagram [DIAGRAM].

Consider the transition relationship, which for a given factor specifies whether each following trial has the same value for that factor (a repetition) or a different value (a switch). We will represent this relationship as a new, derived, factor. This factor has two levels: one level which indicates that the following stimuli was a repetition, the other that it was a switch. What are the parameters for the window for this relationship? Because we consider two trials at a time, and want to consider all pairs of consecutive trials, the width is two, and the stride is one.

Given this conceptual window, the user can then provide a function for each level of the derived factor which specifies, based on the values of the levels which are "in scope" in the window, whether the derived level is selected or not. For the transition example, this may look like:

```
def repetition(ink_colors):
  if ink_colors[0] == ink_colors[1]:
    return True
  return False
```

The repetition function takes a list of ink colors. Because we're considering all consecutive pairs of trials, the window size is two, so the length of the ink colors list is also two. Conceptually, this function will be applied to all consecutive pairs of trials and report

whether those trials are repeat or not.

We can use this repetition function to define a new pair of levels. Recall that levels act as indicators; when a level of a trial is true, that means that trial possesses the quality that level represents. The level derived using a transition-wide window and the repetition function defined above is true when the trial is has the same ink color property as the trial before it. Because levels can be defined to be any [printable] type, derivation functions can use any property of the level name as part of their logic.

Putting it all together, these derived levels are represented in SweetPea as:

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

def repetition(ink_colors):
  if ink_colors[0] == ink_colors[1]:
    return True
  return False

repeat_level  = DerivedLevel("rep", Transition(repetition, [ink_color, ink_color
switch_level  = DerivedLevel("swi", Transition(repetition, [ink_color, ink_color
transitionFactor = Factor("congruent?", [repeat_level, switch_level])
```

Let's consider another use-case. In the Stroop experiment, it is useful to ask whether or not a trial is *congruent*, meaning whether the ink color and the text specify the same color or not. We can define this relationship by considering a WithinTrial window, in which the width is one (since we consider a single trial at a time), and the stride is also one. The derivation function for congruence is similiar to the one for repetitions; here however we always compare the first (and only) item in the ink colors and texts lists.

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

def congruent(ink_colors, texts):
  if ink_colors[0] == text[0]:
    return True
  return False

con_level  = DerivedLevel("rep", WithinTrial(congruent, [ink_color, ink_color]))
inc_level  = DerivedLevel("swi", WithinTrial(congruent, [ink_color, ink_color],
congruenceFactor = Factor("congruent?", [con_level, inc_level])
```

Windows allow us to represent, more generally than just transitions and within-trial constraints, relationships across a sequence of trials. Some experiments may wish to de-

fine relationships which skip trials in the middle of the sequences. An example is an experiment with a ”bait” trial: subjects are asked to respond based on TODO clarify. Additionally, changing the stride allows for things like specifying a constraint over the first half, and then second half of the experiment.

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

- 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

3.2.3 Experimental Design, Balancing and Experimental Structure

Once we've described the trials in terms of levels and factors, we need to specify the set of trials that make up the experiment. The *design* of the experiment is the specification of factors represent a trial. To specify the set of trials in the experiment we need to specify the *crossing* of the design.

A common crossing which SweetPea supports is a *full crossing*. A full crossing consists of exactly one instance of every possible trial. For the small Stroop experiment this means four trials, with one instance each of: (red ink, red text), (red ink, blue text), (blue ink, red text), (blue ink, blue text). Another supported crossing is a variation of a full crossing which is each instance appears exactly k times instead of exactly once. A reason to exclude a factor from a design is to exclude it from the full crossing.

In a full crossing, each of the levels is *balanced* with respect to all of the other levels; they all appear equally frequently. An experiment may additionally specify other levels which must be balanced, including derived levels. For example, we may wish to balance the congruent and incongruent levels.

The final aspect we need to specify is the structure of the experimental sequence. An experimental sequence is a list of one or more experimental blocks. Each experiemental block consists of a crossing of the experimental design. Some experiments may wish to specify multiple experimental blocks to represent prologue or epilogue blocks, or to split an experiment across multiple sessions.

Having discussed and motivated all of the aspects of the SweetPea language, in the next chapter we'll see how these elements are encoded as a boolean formula.

CHAPTER 4

SAT ENCODINGS

In this chapter we'll examine how the language forms we saw in the previous chapter are efficiently encoded as boolean formulas. A program in SweetPea is translated to a boolean formula in Conjunctive Normal Form (CNF), which is a canonicalization that is frequently used by SAT solvers and samplers. We use two techniques for efficient SAT encodings: we encode large "counting" constraints using a Tsietin transform, and user's derivation constraints more directly. After examining how a program is encoded, we'll also discuss the runtime communication with the sampler, and why the correctness guarantees are preserved.

4.1 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 the search strategies that solvers commonly employ while searching for 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, and negation is denoted by a minus sign. 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

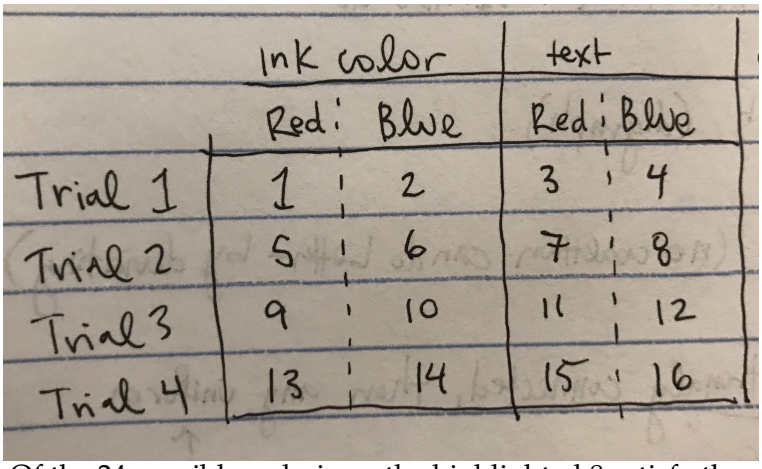


Figure 4.1: Of the 24 possible orderings, the highlighted 8 satisfy the constraints.

convention for generating fresh variable names– you just need to increment the variable counter to get a fresh variable. Each or-clause is written on an individual line, and every term within the or-clause is written as an element of a list; each list is terminated with the number 0. Finally, each formula also has a header which clues the consumer to the number of variables and clauses in formula. The full DIMACS CNF specification of the example in this paragraph is: p cnf 4 2 1, 2, 0 3, -4, 0

This specification is the langauge required to leverage the awesome power of SAT samplers and solvers, but it is clearly not amemable to being manually specified by human hands for anything but the most toy examples. That is why SweetPea provides a high-level domain-specific interface which compiles to these low-level specifications. An interesting aspect of the translation is ensuring that the encoding is efficeint: that is, polynomial in the number of number of variables and clauses with respect to the constraint size.

4.2 Representing SweetPea Primitives in CNF

Recall that experimental designs describe trials in terms of factors and levels, and the relationships between those trials in terms of windows, derivation functions and counting constraints. Here, let's see how all of those components are encoded into a boolean formula; in the next chapter we'll discuss trade-offs of some encoding decisions and more details.

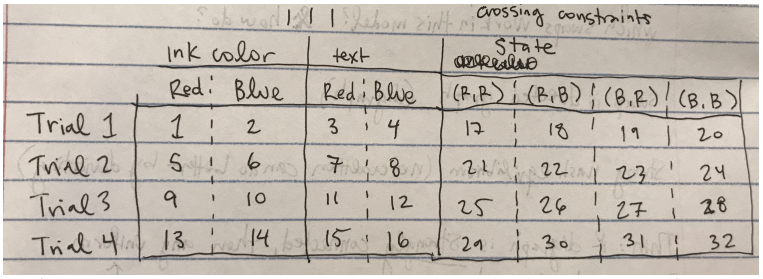


Figure 4.2: Of the 24 possible orderings, the highlighted 8 satisfy the constraints.

4.2.1 Representing Levels and Factors

An experimental sequence consists of a specification for every trial in the experiment. Each trial is parameterized by factors, and the specification indicates which level of each factor is selected. Levels correspond naturally to boolean variables; a factor has one level at a time selected and each level can exist in either a selected state or a non-selected state.

The encoding we use allocates one boolean variable for each level of each factor of each trial. See figure **Figure 4.1** on the previous page for a visualization of the variable allocation for the running Stroop example. In the running example, there are 4 trials, each of which is described by two factors with two levels. The encoding we use allocates one variable for every factor for every trial. As mentioned in the previous subsection, variables in the DIMACS CNF format are index-based, which means that the number 1 refers to the first boolean variable and so on. This means that the assignments to variables 1 through 4 represents the specification of the first trial. For instance, the assignment ”1 -2 -3 4” means the ink color is red (and not blue), and the text is blue (and not red). For any experiment we allocate (total number of levels) * (number of trials) boolean variables which directly encode a specification of an experimental sequence.

In addition to any user-defined constraints, we always have two kinds of constraints over the level variables to correctly model the experiment. The first are *consistency constraints* which state that only one level of each factor is selected at once. In the running example constraints of the form ”(1 and not 2) or (not 1 and 2)” for each pair of levels would ensure that only one level is true at a time. The second kind of constraint is a *crossing constraint* which ensures that the correct number of each kind of possible trial occurs in the experimental sequences.

Representing crossing constraints is more complicated. In the running example, the

logic for the crossing constraints should ensure that there is one (red ink, red text), one (red ink, blue text), one (blue ink, red text) and one (blue ink, blue text). To do encode this, we allocate 4 more variables for each trial, each of which literally encodes which of those 4 states that trial represents. See figure **Figure 4.2** on the preceding page for a visualization of the additional variables. Let's look at the constraints we need to add for the first trial– first we'll add a constraint that specifies that trial is in the (red ink, red text) state if-and-only-if the red ink level is selected, and the red text level is selected (17 iff (1 and 3)). We add constraints of this form for the other 3 states as well, ie (18 iff (1 and 4)), (19 iff (2 and 3)), (20 iff (2 and 4)). Next, we create constraints to ensure that exactly one of each of the state variables across all the trials is true, ie, (exactly one of 17, 21, 25, 29 is true), (exactly one of 18, 22, 26, 30 is true), and so on.

A realistic experiment may have on the order of 100 trials. This means that we need to be careful with this last constraint, that exactly one of ¡list of length 100¿, is true. In other types of crossings (see chapter 4, section 4.2.3) that constraint may more generally be exactly k of ¡list of length 100¿ is true. The naive encoding (these k, or those k, or those k, and so on) is exponential, therefore not efficient, and in practice too large for our problem size. Instead, we get an efficient encoding using a Tsietin transform to encode these *counting constraints*. For a detailed description of this algorithm, see the next chapter.

In summary, an experimental sequence is represented as a list of boolean variables for each level of each trial. Assignments to these variables uniquely specify the experimental sequence. Irregardless of other user constraints, these variables are bound by constraints to ensure exactly one level is selected at a time for each factor, and that each state in the crossing is represented the correct number of times. The next chapter will examine the trade-offs from our encoding decisions. Next we'll see how these variables which represent levels can be used to encode derived levels and used with window constraints.

4.2.2 Representing Derived Levels and Derivation Functions

Derived levels are represented using the same technique as we used for the crossing constraints. We allocate a new boolean variables for each derived level, and bind it (in an if-and-only-if clause) to the original experimental sequence variables using the user-provided derivation function. To describe this process, we'll see how to add congruency

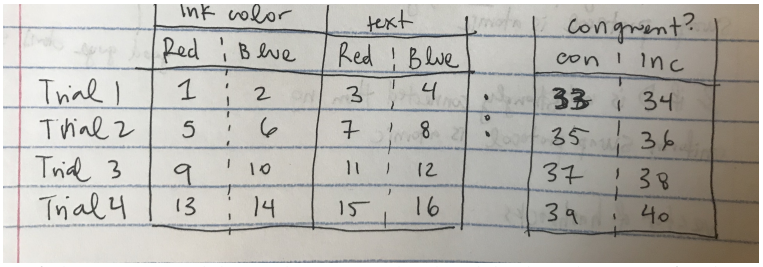


Figure 4.3: Of the 24 possible orderings, the highlighted 8 satisfy the constraints.

constraints to the Stroop example; see figure **Figure 4.3** on the current page for a visualization of these new variables for the derived congruence levels.

The current sweetpea implementation special-cases the constraint encoding for congruency because this is a common constraint in the experiments we wish to model. We encode the congruency TODO

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.

4.3 Correctness Guarantees

- cite guarantee from unigen

- postulate why this is preserved: TODO

CHAPTER 5

IMPLEMENTING SWEETPEA

This chapter examines the implementation and encoding decisions.

5.1 Language Implementation: Decisions and Alternatives

SweetPea is an embedded domain specific language, meaning that instead of providing its own syntax and the compiler infrastructure to match, it is instead implemented as a library in an embedding language. Here we look at the choice of embedding language, as well as some choices about boolean encodings.

5.1.1 Embedding Language

The current implementation of SweetPea is embedded in Python; the syntax presented in the example code snippets in chapter 4 are valid python program snippets. The subsystem for handling Tsietin transforms is currently implemented in Haskell; an earlier implementation of the user-facing SweetPea syntax was also implemented in Haskell. We chose Haskell as an implementation language because its strong static types and functional purity lent it to verifying the correctness of the system. However, many scientists are familiar with Python, and not with Haskell syntax, so we decided to port the user-facing interface to Python to make it easier for them to use and interface with their existing workflows.

5.2 Communicating with the SAT-Sampler

Figure 5.1 on the following page illustrates SweetPea's current architecture. It is more complicated than a single executable. This is partially because part of the constraints are generated by a Python program and part by the Haskell subsystem, and partially because the SAT sampler that SweetPea calls out to, Unigen, only runs on POSIX compliant systems (Linux, but not MacOS). Because of both of these facts, SweetPea is a python program

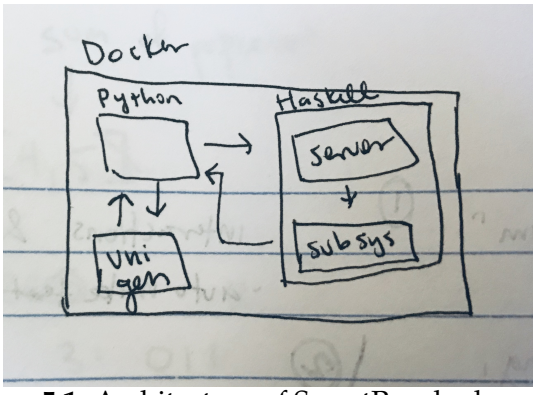


Figure 5.1: Architecture of SweetPea deployment.

which runs, and calls out to, a docker container. This docker container consists of the Haskell subsystem which can efficiently encode counting constraints, and unigen, which takes the constraints that the program compiled to and produce a satisfying assignment, should one exist.

Unigen provides two features in addition to the standard DIMACS format: the ability to denote which variables are ”important” for a final assignment (in our case, those that directly encode the state of the experimental sequence), and the ability to provide xors rather than or clauses. The current version of SweetPea uses the first feature (the important variables are called the *supports*), and does not use the second, though it is an optimization for future work.

After SweetPea, the python program which called out to the docker container, gets a response from unigen, it translates it back to a human-readable format. This means translating the assignments to the experimental sequence they represent (see **Figure ??** on page ?? for an example). The resulting sequence can either be printed as a list of strings, or can be exported to a list of python dictionaries to be directly integrated into other psychology computational tools such as psyNeuLink.

5.2.1 Internal Representations

As mentioned in the previous chapter, we encode levels by allocating one boolean variable per level to indicate the whether that level is in a selected or unselected state. This isn’t the only possible

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

- todo: more examples

5.3 Encoding Counting Constraints: 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”

5.3.1 Binding Variables: Iff

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

5.3.2 Adders

- half adder example

- full adder example

5.3.3 Ripple Carry Adders

- multiple options here, the one I went with

- trade-offs of options

5.3.4 Pop Count Circuit

- pop count circuit example

5.3.5 Exhaustive Testing

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

- tested for a given size all assignments

CHAPTER 6

COMPILING STROOP IN EXCRUTIATING DETAIL

This chapter introduces no new material, but instead collects the parts of the running example described in chapters 4 through 6 to provide an end-to-end view of running a simple experiment in SweetPea. We will then also look at a distribution of solutions provided by SweetPea.

6.1 Stroop in SweetPea

- example code listing of small w/ congruence in full crossing

6.2 Compiling Stroop to CNF

- show how the high level constraints get translated to low level constraints (ie, a list: color one hot constraints, text one hot constraints, derivations, state constraints: show a diagram for the popcount circuit which is generated)

- show full DIMACS file

6.3 Synthesizing an Experimental Sequence

- show input to unigen and output

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

6.4 Verifying the Uniformity Guarantee

- run it a bunch and histogram results

TODO MOVE AFTER IMPLEMENTATION DETAILS

CHAPTER 7

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.

7.1 Psychology Toolboxes

- psyScope [1]

- psychoPy [2]

- OpenSesame [4]

7.1.1 Reproducibility Crisis

- TODO: surely something goes here

7.2 Domain Specific Languages

- probably don't need to cite anything here?

7.2.1 Solver Aided Languages

- rosette [5]

- sketch: "Domain-Specific Symbolic Compilation"

- dafny maybe

- hyperkernel: co-designing a language and the verification

7.3 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).

7.3.1 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

7.3.2 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

[6]

7.3.3 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 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.

[6] G. S. Tseitin, *On the complexity of derivation in propositional calculus*, in Automation of reasoning, Springer, 1983, pp. 466–483.