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

December 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)	**03 December 2018** Date Approved
Matthew Flatt ,	Member	**03 December 2018** Date Approved
Jonathan Cohen ,	Member	**03 December 2018** Date Approved
none ,	Member	**03 December 2018** Date Approved
none ,	Member	**03 December 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 designs 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 experimental 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 which satisfies the design must have an equal likelihood of being chosen for the experiment. This requirement is computationally intractable; SweetPea gains traction by approximately sampling the solutions using the SAT-sampler Unigen. This allows scientists to easily create sharable experimental designs in terms familiar to their domain which produce unbiased sequences of trials.

For my parents; I'm sorry this thesis contains neither theorems nor jokes.

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

ACKNOWLEDGEMENTS

A huge thank you to my advisor, Vivek Srikumar, for graciously being willing to advise me despite the fact that I read the floating point standard for fun. I couldn't have had a better advisor; Vivek is a master peddler of ideas and an unrelenting optimist– he consistently convinced me that the research we were working on was interesting and important whenever I lost faith. I really appreciate both being given space to explore my research interests as well as direction on projects, even when they involved working out bit-wise arithmetic. In addition to learning at least a little about how research is done (it's at the twelfth hour), I hope I've learned enough from Vivek about story telling, patience and confidence to apply these practices in the future.

Thank you to everyone involved with SweetPea: Matthew Flatt, Jonathan Cohen, Sebastian Musslick and Ben Draut. Thanks to Jon & Sebastian for so enthusiastically providing the motivation for this project; it's much more fun to work on something that someone really actually wants to use. Thanks to Matt for his guidance and his patience as I discovered details he saw all along, and to Ben for taking the helm to get SweetPea into the wild.

Thanks to the incredibly patient crew who proof-read everything I've written in the past few years- Tobin Yehle, Dasha Pruss, Nic Bertagnolli & Zach Price. Thank you to Will Byrd for making me upset with the state of computing and to Michael Adams for telling me how all my academic interests were related.

Thanks also to my friends from the Recurse Center for many delightful computer science distractions, and to my friends in SLC for distractions of other flavors. Thank you to Tobin, Nic and Laura for always being on the lookout for the next scheme, wild idea, and adventure, and for inviting me along– and sometimes carrying my pack.

And, of course, thank you to my parents, and to my sister.

Life is rolling on, and it's all very exciting.

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

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 [3] [5] [12], 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 [18], 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 [7] 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

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, then, 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. 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. 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. SweetPea provides this guarantee by representing the experiment as a boolean formula and interfacing with a SAT sampler; this constrains the high-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 [17]. 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.

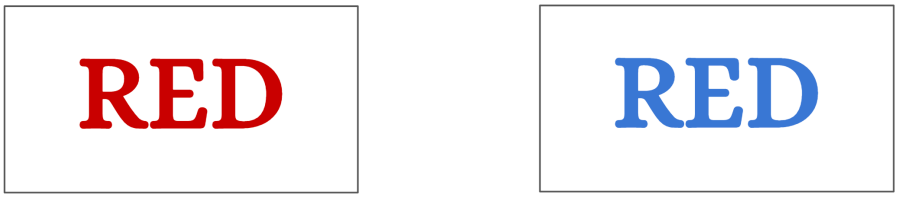
To prove that we do not introduce bias because of the way we construct experimental sequences, we would like to ensure that each valid sequence is equally likely. In this example, we have four stimuli so there are 4 factorial = 24 possible orderings. Of those 24 orderings, 8 satisfy the constraints as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

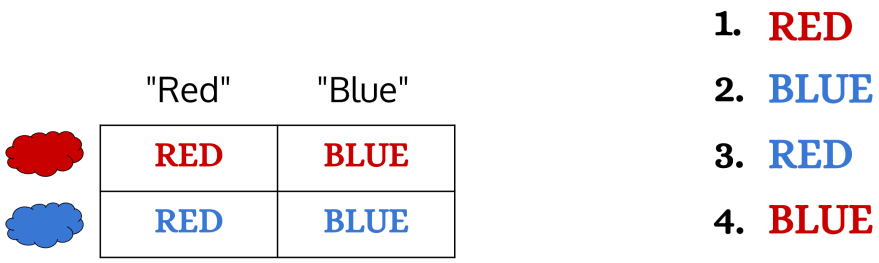
The version of the SweetPea language we'll discuss is embedded in Python, so uses Python syntax.

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.

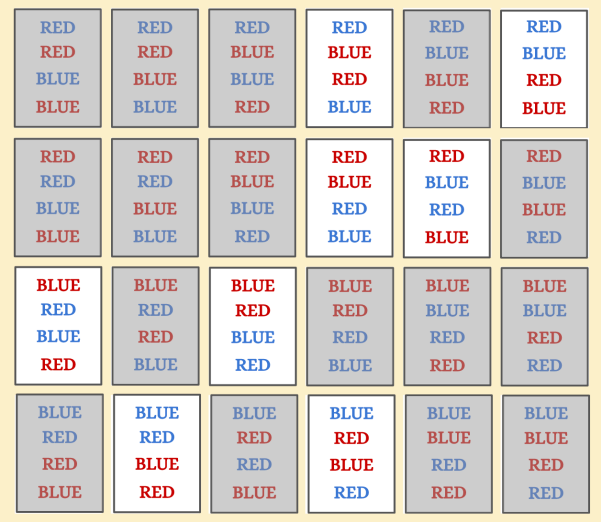


Figure 2.3: Of the 24 possible orderings, the highlighted 8 satisfy the constraints.

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

Next, let's represent the constraint that there should be no repetitions of stimuli whose text is the same. `NoMoreThanKInARow` is a constraint constructor function in SweetPea, which allows us to declaratively specify the constraint:

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

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 = FullyCrossBlock(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 or 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 which uses universal hash functions to find solutions in a principled way, but providing this guarantee is the goal of all uniform sampling. Unigen guarantees that it will return a satisfying assignment with approximately uniform probability of choosing any satisfying assignment to a given boolean formula.

How does this guarantee, which is provided at the level of boolean variables, translate to the guarantee we want, at the level of the experimental design? As we'll discuss further in Chapter 4, so long as every boolean formula uniquely describes a single experimental design, we know that the satisfying experiemental sequence is approximately uniformly probable among all satisfying experimental sequences. 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

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: scientists describe their experimental designs in the terms they usually use in their domain, and SweetPea handles the rest.

In this chapter, we'll document the features of the SweetPea language. They largely follow the components we saw in the Overview chapter. Experimental design specifications consist of:

- trials described in terms of factors and levels,
- constraints and relationships over those trials,
- counting and balancing constraints,
- and a block specification consisting of a crossing and a design.

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 Experimental Design Wishlist

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.

3.1.1 Descriptions of Stimuli

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.

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 aqua) 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 researcher 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 we are testing whether the presence of a certain level effects the subject's reaction time, we need control over when that level occurs and how often it occurs in relation to other levels. More generally, we need to be able to constrain the presence or absense of relationships which we define over the levels. Since levels can be grouped into arbitrary sub-levels, we should more generally be able to specify constraints on relationships over the sub-levels.

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). This is an important condition to control for in task-switching experiments because a subject's reaction time may depend on whether they just performed a similar or dissimilar task. An experimenter may wish to specify that a subject experience as many repetition or tasks as task switches.

Another example, which we saw in Chapter 2, 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.

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, experiments expressed in SweetPea may need to define user-specified relationships between levels.

3.1.3 Experimental Design and Balancing

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

Sometimes, however, factors either need to be excluded or duplicated to make an experiment feasible. 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. 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 (as in **Figure 7.1** on page 39): 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).

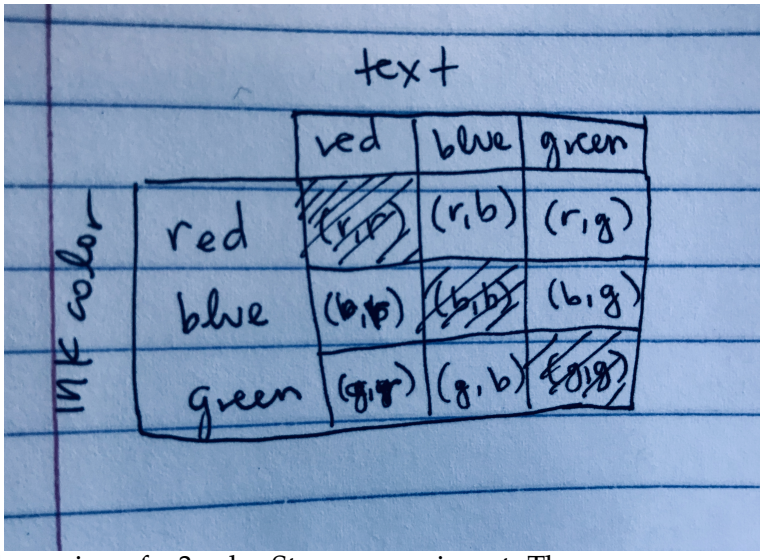


Figure 3.1: Full crossing of a 3 color Stroop experiment. There are more congruent stimuli (along the diagonal) than incongruent stimuli.

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. This desirable, more general crossing specification, is not currently implemented in SweetPea and is further discussed in the Future Work Chapter.

3.1.4 Structure of an Experiment

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 not always possible 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 exactly 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 Language Reference

Now that we have the context of the types of experiments we would like to model, let's discuss the SweetPea grammar. A specification of an `Experiment` is a valid SweetPea program. The full schema is provided below; each part will be discussed in detail in the following subsections:

```
Experiment         = [Block, ...]
Block              = Design Crossing [Constraint, ...]
Design             = [Factor, ...]
Crossing           = [Factor, ...] CrossingType
CrossingType       = FullyCrossed
                   | MultipleFullyCrossed
Constraint         = AtMostKInARow  K (FactorName, LevelName)
                   | ExactlyKInARow K (FactorName, LevelName)
                   | AtLeastKInARow K (FactorName, LevelName)
Factor             = FactorName [Level, ...]
                   | FactorName [DerivedLevel, ...]
DerivedLevel       = LevelName Window
Window             = Stride Width DerivationFunction [FactorName, ...]
DerivationFunction = ( [FactorName, ...]... ) -> Boolean
Level              = <any printable type> | [Level, ...]
FactorName         = <any printable type>
```

Some of these terms are specified directly, like FactorName and Level, while others are created via constructors. A full list of the constructors in SweetPea is:

```
Factor constructor:
  Factor(<factor name>, <list of levels>)

Window constructors:
  Transition(<derivation function>, <list of factors>)
  WithInTrial(<derivation function>, <list of factors>)
  Window(<stride>, <width>, <derivation function>, <list of factors>)
```

```
DerivedLevel constructor:
  DerivedLevel(<level name>, <window>)

Counting Constraint constructors:
  AtMostKInARow (<k>, (<factor name>, <level name>))
  ExactlyKInARow(<k>, (<factor name>, <level name>))
  AtLeastKInARow(<k>, (<factor name>, <level name>))

Block constructors:
  FullyCrossBlock(<design>, <crossing_factors>, <weights>)
  MultipleFullyCrossBlock(<design>, <crossing_factors>, <weights>, <numReps>)
```

Once the user has constructed a complete SweetPea specification by creating an `Experiment`, they can call a function to synthesize the experimental sequence:

```
synthesize_trials(<experiment>, <number of experiment sequences>)
```

The remaining subsections will investigate and document each component of the grammar.

3.2.1 Factors and Levels

```
  Factor             = FactorName [Level, ...]
                     | FactorName [DerivedLevel, ...]
  Level              = <any printable type> | [Level, ...]
  FactorName         = <any printable type>
```

Factors are constructed as named tuples, specified by factor 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 = Factor("example_name", ["example_level"])
boolean_factor = Factor("booleans",     [False, True])
numeric_factor = Factor(1234,           [1, 2, 3, 4])
text           = Factor("text",         ["red", "blue"])
light_colors   = Factor("light_colors", ["yellow", "aqua"])
dark_colors    = Factor("dark_colors",  ["navy", "maroon"])
ink_color      = Factor("ink_color",    [light_colors, dark_colors])
```

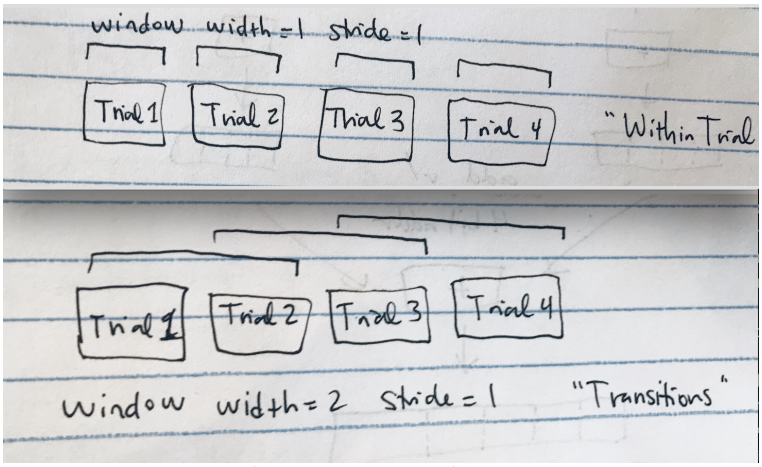


Figure 3.2: Windows define the scope of relationships between trials.

In the resulting experimental sequence, exactly one choice of level is selected per trial; if a factor has nested levels then the nesting is flattened in the final experimental sequence. Nesting is only useful at the high-level to describe groupings of levels, for instance to be the subject of a constraint.

3.2.2 Deriving Levels with Windows

```
DerivedLevel       = LevelName Window
Window             = Stride Width DerivationFunction [FactorName, ...]
DerivationFunction = ( [FactorName, ...]... ) -> Boolean
```

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 derivation functions, windows allow the user to specify new "derived" levels and factors based on the values of the levels within the window.

SweetPea provides constructors for two common window sizes: `WithinTrial` windows, which are used to define new properties based on properties of each individual trial have width of 1, stride of 1. `Transition` windows, which are used to define properties based on each consecutive pair of trials, have width of 2, stride of 1.

```
  Window Constructors:
Transition(<derivation function>, <list of factors>)
WithInTrial(<derivation function>, <list of factors>)
Window(<stride>, <width>, <derivation function>, <list of factors>)
```

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 derived 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, in the running example, 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

def switch(ink_colors):
  return not repetition(ink_colors)

repeat_level = DerivedLevel("repeat",
                 Transition(repetition, [ink_color]))
switch_level = DerivedLevel("switch",
                 Transition(switch, [ink_color]))
transition_factor = Factor("transition?", [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] == texts[0]:
    return True
  return False

def incongruent(ink_colors, texts):
  return not congruent(ink_colors, texts)

con_level  = DerivedLevel("con",
               WithinTrial(congruent, [ink_color, text]))
inc_level  = DerivedLevel("inc",
               WithinTrial(incongruent, [ink_color, text])
congruence_factor = 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. See **Figure 3.2** on page 16 for an illustration. Some experiments may wish to define 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 a property of a trial some number of trials previous. Additionally, changing the stride allows for things like specifying a constraint over the first half, and then second half of the experiment.

SweetPea currently supports transition and congruency constraints, but does not support the most general form of derivation function. See future work for a discussion on how to handle fully arbitary derivation functions.

3.2.3 Counting Constraints

SweetPea also provides three counting constraint constructors which let the user constrain levels to appear some number of times. These constructors are also named tuples, which take the number to count to, k, and the level to which apply this constraint. Since individual levels are not directly referenced by variables, the user must specify a tuple of factor name and level name.

```
Counting Constraint constructors:
  AtMostKInARow (<k>, (<factor name>, <level name>))
  ExactlyKInARow(<k>, (<factor name>, <level name>))
  AtLeastKInARow(<k>, (<factor name>, <level name>))
```

The `AtMostKInARow` constraint is useful as a ”sanity check” constraint, to ensure that independent variables do not show up an unreasonable number of times.

3.2.4 Experimental Design, Balancing and Experimental Structure

```
Experiment          = [Block, ...]
Block               = Design Crossing Constraints
Design              = [Factor, ...]
Crossing            = [Factor, ...] CrossingType
CrossingType        = FullyCrossed
                    | MultipleFullyCrossed
Block constructors:
  FullyCrossBlock(<design>, <crossing_factors>, <weights>)
  MultipleFullyCrossBlock(<design>, <crossing_factors>, <weights>, <numReps>)
```

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. As mentioned before, a full crossing is commonly, but not always, used.

The design is specified explicitly as a list of factors, while the block is built by a constructor which knows how to construct the block based on the crossing type. Each constructor takes two lists of factors, one which represents all the factors which represent a trial (`design`), and one which represents which factors should be included in the crossing (`crossing_factors`). Each trial in the resulting experimental sequence will have a value for factors specified in `design`.

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.

Finally, to generate the experimental sequence, the user can call `synthesize_experiment` which takes an `Experiment` and produces a list of trial specifications.

Putting together all the code snippets in this section, here is a full specification of a simple Stroop experiment in SweetPea:

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

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

def incongruent(ink_colors, texts):
  return not congruent(ink_colors, texts)

con_level  = DerivedLevel("con",
               WithinTrial(congruent, [ink_color, text]))
inc_level  = DerivedLevel("inc",
```

```
              WithinTrial(incongruent, [ink_color, text])
congruence_factor = Factor("congruent?", [con_level, inc_level])

design       = [color, text, conFactor]
crossing     = [color, text]

constraints = [NoMoreThanKInARow(1, ("congruent?", "con"))]

block        = FullyCrossBlock(design, crossing, constraints)

experiments  = synthesize_trials(block, 5)
```

This design will produce an experimental sequence which might look like:

```
color blue | text red  | congruent? inc
color blue | text blue | congruent? con
color red  | text blue | congruent? inc
color red  | text red  | congruent? con
```

In this chapter we have seen the SweetPea language features, which is the information a user would want to program in SweetPea. We have not yet discussed how these language features are implemented; in the following two chapters we will see how each of these primatives is encoded as a part of 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 convention for generating fresh variable names– we 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.

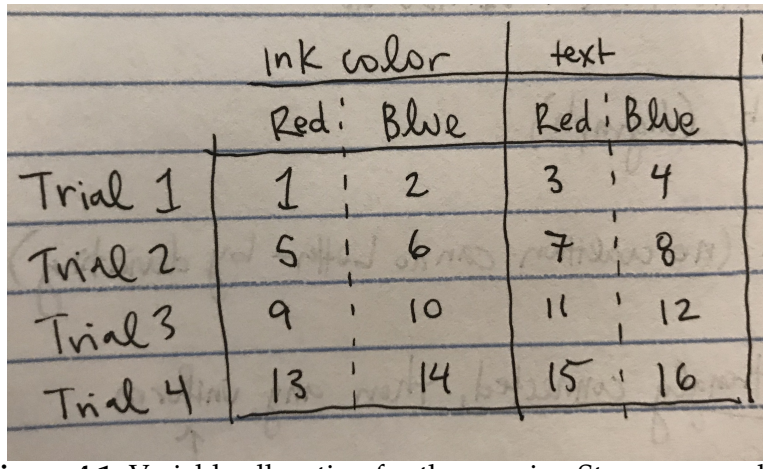
4.2 Representing SweetPea Primitives in CNF

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.

4.2.1 Representing Levels and Necessary Constraints

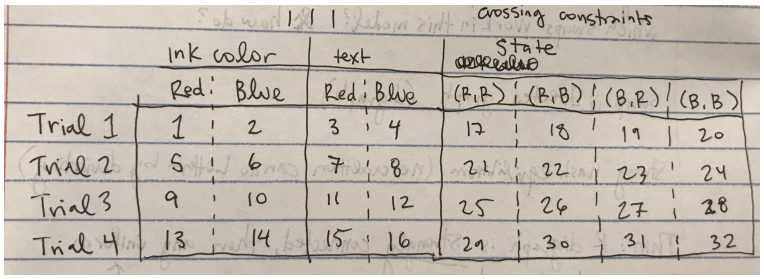
An experimental sequence consists of a specification for every trial in the experiment. Each trial is parameterized by factors, and the specification indicates which level of each factor is selected. Levels correspond naturally to boolean variables; a factor has one level at a time selected and each 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 next page for a visualization of the variable allocation for the running Stroop example. In the running example, there are 4 trials, each of which



	Ink color		text	
	Red	Blue	Red	Blue
Trial 1	1	2	3	4
Trial 2	5	6	7	8
Trial 3	9	10	11	12
Trial 4	13	14	15	16

Figure 4.1: Variable allocation for the running Stroop example.



	Ink color		text		crossing constraints State			
	Red	Blue	Red	Blue	(R,R)	(R,B)	(B,R)	(B,B)
Trial 1	1	2	3	4	17	18	19	20
Trial 2	5	6	7	8	21	22	23	24
Trial 3	9	10	11	12	25	26	27	28
Trial 4	13	14	15	16	29	30	31	32

Figure 4.2: Additional variable allocation necessary for encoding the states possible in the full crossing.

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.

There are two kinds of constraints which need to be encoded; user-defined constraints and two other constraints which are always necessary 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 efficiently encode these *counting constraints* following a technique similar to the one described in [15]. 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

The current SweetPea implementation special-cases the constraint encoding for congruency and transitions because these are common constraints in the experiments we wish to model.

Let's consider a congruency derivation over the simple Stroop example. In addition to the variables which represent the levels of each trial, we'll allocate 2 new variables for each trial each of which indicates whether or not the trial is congruent. This means the variable allocation looks like:

Trial #	color		text		congruent?	
	red	blue	red	blue	con	inc
1	1	2	3	4	5	6
2	7	8	9	10	11	12
3	13	14	15	16	17	18
4	19	20	21	22	23	24

Then, we encode constraints on when these new variables are true. When is the first trial congruent? The first trial (`5`) is congruent if-and-only-if either the trial's color is red (`1`) and the text is red (`3`), or the color is blue (`2`) and the text is blue (`2`). In boolean logic this is: `5 iff (1 and 3) or (2 and 4)`. We create constraints of this form for all the remaining variables, and translate this constraint to CNF through a general CNF conversion technique.

What if instead we considered transitions? Let's reinterpret the variable named `5` to indicate whether that trial has the same text as the following trial:

Trial #	color		text		transition	
	red	blue	red	blue	swi	rep
1	1	2	3	4	5	6
2	7	8	9	10	11	12
3	13	14	15	16	17	18
4	19	20	21	22	23	24

We can then state that the trial switches (`5`) if-and-only-if this trial's text is red (`3`) and the next trial's text is also red (`9`), or they are both blue (`4` and `10`). In boolean logic this is: `5 iff (3 and 9) or (4 and 10)`. This identical clause structure allows us to model both of these common derivations. Encoding arbitrary derivations is left as future work.

4.3 Correctness Guarantees

The goal of SweetPea is to produce experimental sequences which satisfy the experimental design with uniform probability: not favoring any valid sequence over any other. SweetPea achieves this goal by relying on the uniform samping guarantee that Unigen provides in [2].

Unigen is a near-uniform SAT sampler. Near-uniform is defined in [7] as

$$\begin{aligned}\frac{1}{\text{number of solutions} \times (1+\epsilon)} &\leq \text{Probability[Finding a given solution]} \\ &\leq (1+\epsilon)/\text{number of solutions}\end{aligned}$$

This guarantees that the number of *SAT witnesses*, or satisfying solutions, to the CNF is approximately uniformly probable. How does the number of satisfying SAT solutions relate to the number of experimental sequences which satisfy the experimental design?

When we translate from the high level specification to the boolean formula, we introduce new internal boolean variables (ie the sum and carry bits of the adders in the popcount encoding). We hypothesize that the number of satisfying SAT solutions is exactly the same as the number of experiment sequences when none of the internal boolean variables are redundant. Then, all the variables which directly encode the experiment are in lockstep with all the extra internal variables. If there are redudant variables, they could skew the number of satisfying assignments such that finding an assignment would be uniform with those variables, but not in the variables we care about. To the best of our knowledge, our encoding does not introduce redundant variables, and therefore preserves this guarantee.

CHAPTER 5

IMPLEMENTION DETAILS

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.

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 encoding the counting constraints 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.

Why SAT sampling? Why does SweetPea need to use a SAT-sampler? Using a SAT-sampler is like bringing a nuke to a knife fight; if a more straight-forward solution was tractable, it would be much simpler, faster, and more maintainable to use a simple solution.

Recall that the goal is to find an ordering of trials which satisfies the constraints, and to return that valid ordering with uniform probability relative to all the other valid orderings. This means that our search space is the space of all possible orderings of the given set of trials (which may realistically consist of 120 trials, corresponding to 120! orderings), and elements of this search space are specific orderings.

The first straight-forward solution is to try a random element in the search space,

and see if that ordering satisfies the constraints. For realistic experiments, we found this approach to be intractable– the set of solutions is sparce in the search space.

The other simpler solution is to generate all of the valid elements, and then choose uniformly among them. In practice, we also found this approach to be intractable. While the space of solutions was sparce in the search space, it was also far too large to exhaustively generate.

The simplest possible solution, and the one used in practice today by psychologists, is to generate the experimental sequence one trial at a time, backtracking when an assignment is made which violates the constraints. Unfortunantly this approach does not produce uniformly probable solutions. Similarly, sampling methods like MCMC would also be able to tractably find solutions, but wouldn't provide any guarantee about their distribution.

For the experimental designs we want to support, we found that the methods mentioned above were either intractable or favored certain satisfying sequences over others. Solving the search problem SweetPea addresses is #P hard [19], and our problem size is too large to use brute force despite the exponentially large space.

SweetPea uses the SAT-sampler Unigen because Unigen implements an approximate sampling method with statistical guarantees. Approximate sampling is a compromise: it allows us to gain traction on the otherwise untractably large space, while none-the-less maintaining strong statistically rigerous guarantees of the distribution of solutions. We use Unigen because implementing the underlying sampling method to work well in practice is complicated and an art; it seemed wiser to rely on an existing implementation rather than re-engineer it. This decision, however, to outsource searching the space to Unigen, came at the cost of encoding the domain specific constraints as a boolean formula. The decision to have SweetPea translate from the high-level domain specific semantics to SAT, as opposed to any other possible search space encoding, is an engineering decision; it comes directly from deciding it was better to rely on Unigen rather than reimplement parts of it's strategy. It is concievable that in the future, SweetPea would be able to compile to other backend representation, amenable to other search strategies.

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

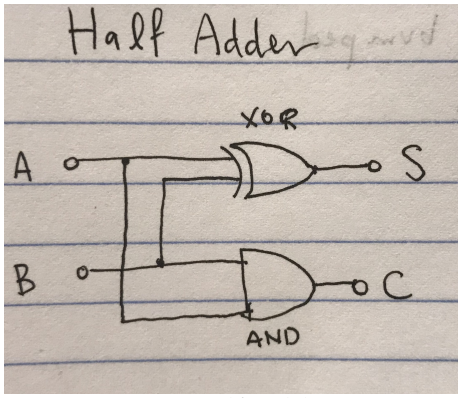


Figure 5.1: Half Adder Logic.

or unselected state. This isn't the only possible encoding; for instance, an encoding which is more compact in the number of variables would represent the index of the selected level in binary. This is a trade-off between variables and clauses, and without any aprior knowledge about the SAT solver's search strategy we chose the encoding which made it easiest to write the constraints which rely on those variables.

5.2 Encoding Counting Constraints with Popcount

A fully crossed design specifies that each possible combination of levels occurs once in the design. A realistic experiment may have 7 factors, with 2 levels each, for a total of $2^7 = 128$ trials. Each of the levels will appear in half the trials, which means we need to encode constraints of the form "exactly 64 of these 128 variables must be true". The naive encoding ("these 64, or those 64, or those 64, etc") will have 64 choose 128 clauses, which is 10^{37} clauses– far more than any SAT solver can possibly handle. Therefore, we need to encode these kinds of counting constraints in an encoding which is linear in the number of clauses and variables.

To achieve this linear encoding, we emulate the logic in a *popcount* circuit. A popcount circuit is a piece of hardware built out of logic gates which, given a bit string, reports how many bits of the bit string are set. A popcount circuit is implemented as a series of multi-bit wide adders. To explain how we emulate the logic in these circuits we will start with adders and build up to the popcount circuit.

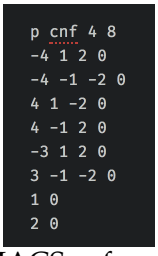


Figure 5.2: The DIMACS cnf representation of 1 + 1.

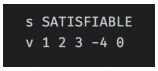


Figure 5.3: The solution to 1 + 1.

5.2.1 Adders

First let's consider a half-adder. A half-adder is a circuit which takes two single bit inputs `a` and `b` and sets two single bit outputs `s` ("sum") and `c` ("carry") according to what the sum of the values of `a` and `b` are. The value of the input bits is 1 if that variable is true, and 0 if that variable is false. The output bits are boolean functions of the input bits: `s = a xor b ; c = a and b`. See **Figure 5.1** on the previous page for an illustration.

What does it mean for a boolean variable to "equal" some relationship between other boolean variables? Really what we mean here is that their values vary in lock-step; that the variable being bound is true only when the relation is true, and false identically when the relation is false. This means that we can represent a half-adder with the boolean formula: `(s iff (a xor b)) and (c iff (a and b))`.

How does using a SAT solver allow us to emulate this circuit? Recall that the solver's job is to return an assignment to each of our variables `(a, b, c, s)` such that the entire formula.

Let's look at some examples. Lets imagine adding 1 + 1; this means that the full formula we hand to the solver is: `(s iff (a xor b))` and `(c iff (a and b))` and `(a)` and `(b)`. See **Figure 5.2** on this page for the DIMACS cnf representation of these boolean clauses. For the entire formula to be true, each of the and clauses must be true. This trivially means that the assignment for `a` is true, and `b` is true. Looking at the more interesting clauses, we know `s` is true `iff (a xor b)`. `a xor b` is false when both `a` and `b` are true, so `s` must be false for that clause to be true. We also know that `c` is true `iff (a and b)`, so `c` must also be true.

This means that for this input, the solver would give us the satisfying assignment:

```
a is true
b is true
s is false
c is true
```

See **Figure 5.3** on the previous page for the solution from the SAT solver given the query in **Figure 5.2** on the preceding page.

We can interpret this to mean that the carry bit is 1, and the sum bit is 0: meaning that 1 + 1 = 2. The rest of the constructions in this section follow this intuition: if we build up logical relationships between boolean variables which emulate circuits which perform the computation we wish perform, and constrain some of the input variables, then SAT solver will find what the values of all the other variables must be.

In practice, we want to use a full-adder rather than a half-adder; the difference is that a full-adder also considers a "carry in" bit. This makes it possible to chain full-adder together into multi-bit wide adders.

There are multiple ways to chain together multiple full-adders. SweetPea uses the simplest option, which is ripple-carry adders. N-bit wide ripple-carry adders chain together n full-adders in the natural way: the sum bits of each successive adder form the sum bits of the solution, while the carry bits chain from one adder to the next.

Alternative ways to build multi-bit adders include carry-lookahead adders and carry-save adders; these circuits were designed as physical circuits which can compute the result of the addition in fewer cycles. In some sense, we are also concerned with how long it takes to compute the result: we would like to use the encodings which are most helpful for the solver. There isn't any reason to believe that using a more complicated multi-bit adder design would result in a more favorable encoding; therefore we decided to use the multi-adder design which is simplest to implement, maintain and debug.

5.2.2 Popcount Circuit

The popcount circuit counts the number of bits which are set in a bitstring by a divide-and-conquer-like technique. We'll discuss the process below; refer to **Figure 5.4** on the next page for a diagram.

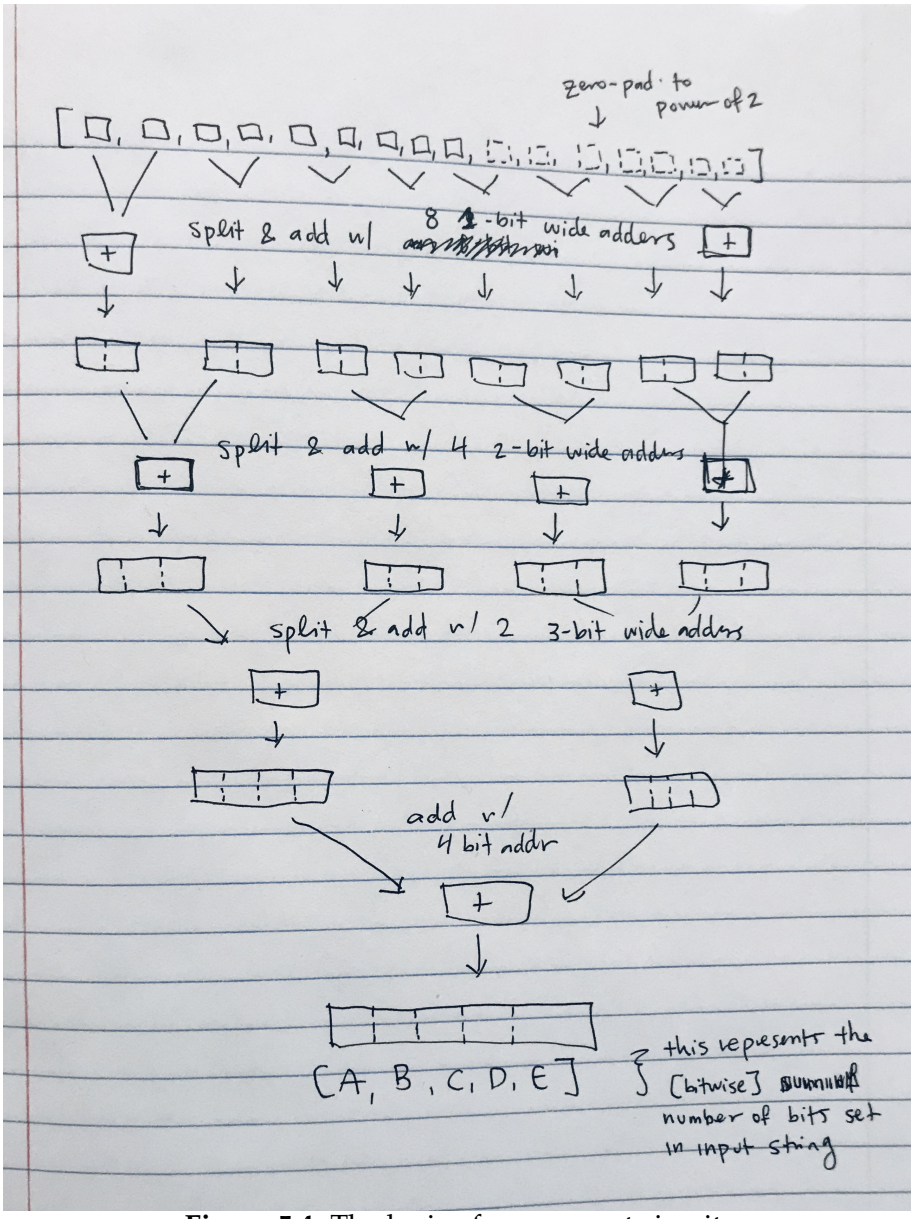


Figure 5.4: The logic of a popcount circuit.

Let's work through an example: let's count the number of set bits in a bitstring with 9 bits. The first thing we do is zero-pad the bitstring until the next power of two. This is an encoding choice which allocates extra variables to avoid having to handle edge-cases in our logic by allowing us to always be able to divide-and-conquer neatly.

Next, we'll pair up bits in the bitstring, and add those using 8 1-bit adders. These two-bit adders will produce 2-bit solutions. Each of these 8 solutions represents how many bits were set in each of their two bit inputs. We repeat this process until we are left with a single 5-bit solution.

Explicitly, this means that we will pair up each of the two-bit solutions, and add them with 4 2-bit wide adders. These will produce 3 bit solutions. We will then pair up these three-bit solutions, and add them with 2 3-bit adders. This will produce only 2 4-bit solutions. Finally, we can take these 2 4-bit answers and add them together with a 4-bit adder to produce a final 5-bit solution.

This final solution represents the number of bits set in the original bitstring in 5-bit binary. Recall that our desired interface was to specify constraints of the form "of these n variables, exactly k must be true". We can then specify this constraint by setting the bits of the final popcount solution to be exactly the same as the binary representation of k. For example, let's specify that we want exactly 3 of the 9 bit to be true our 9-bit popcount example. Let's call the bits of the solution `[A, B, C, D, E]`. A 5-bit binary representation of 3 is [0, 0, 0, 1, 1], so we can enforce the desired constraint by stating `[-A, -B, -C, D, E]`.

We also want to provide an interface for specifying constraints of the form "of these n variables, at least (or at most) k must be true". To encode these constraints, we follow a similar technique: we compute the bits which represent the result of the popcount, and encode k in binary. Then, to enforce the inequality we perform a trick: we rewrite $k < n$ as $k - n < 0$. To enforce this at the bit-level we write (-n) by representing it in twos-complement. We can then add k + (-n) using a ripple-carry adder. Then, to ensure that $k + (-n) < 0$ we will force the result of that summation to be negative by forcing the top sum bit to be zero.

CHAPTER 6

RELATED WORK

SweetPea is a computation tool for pyschologists; it is a solver-aided language which translates between the vocabulary which psychologists employ and a SAT formula. To the best of our knowledge, SweetPea is the only system to address this specific problem, however there is a large body of work related to each of these concepts. Here we present previous work on computational tools for psychology, domain specific and solver aided languages, and on working in combinatorial search spaces.

Psychology Tools, Experimental Design & Replicability There are many computational tools used by psychologists, however, they automate stimulus presentation for running experiments, not experimental design. Some popular toolboxes include PsyScope [3] (now called E-Prime), PsychoPy [5], OpenSesame [12], Presentation and PsychToolBox. PsyNeuLink is a tool for running computational models on experimental tasks. All of these tools are helpful for automating psychology experiments, however they all fill a different role than SweetPea: ideally SweetPea would easily interface with these tools to pipeline experiments from their design to their execution.

SweetPea addresses issues arising from the replicability crisis in psychology, by creating reproducible and sharable experimental sequences. The replicability crisis has been well documented, espcially in recent years– see [11], [10], [14], [13], [6], [16]. In the future, we hope that SweetPea can provide an even higher-level interface where a psychologist can specify a contrast they wish to study, and SweetPea can choose the best constraints to impose on the design. Related work has been done on optimal experiment design by [8].

Solver Aided Languages SweetPea can be thought of as a domain-specific interface to any tool which solves boolean formulas, and specifically to the SAT sampler Unigen, which provides the near-uniformity guarantee. There are other *solver aided languages* such as, Dafny [4], which is the language for interacting with the solver z3 in, and Sketch [1]

which ”Domain-Specific Symbolic Compilation”. There is also Rosette [18] which is a platform for developing solver aided DSLs. Finally, Hyperkernel [9] discusses the process of co-designing a system to be amenable to verification, analogously to how the SweetPea language is designed to be amenable to translation into SAT.

Combinatorial Search Spaces unigen: [7]

CHAPTER 7

FUTURE WORK

SweetPea is a work-in-progress. The ultimate vision for SweetPea is to be a convinient system for describing many diverse types of experimental designs, and then quickly and seamlessly integrating the resulting experimental sequences with the users' existing experiment running pipelines. Ideally, SweetPea could be a tool in dialog with the user; the user should be able to use SweetPea to iterative explore the space of possible satisfiable experimental designs. To achieve this vision, SweetPea will need more high-level language features, and more runtime support for debugging unsatisfiable experiments.

One possible future direction for SweetPea which is orthogonal to this vision is to expand SweetPea to support experiments in other domains beyond psychology. Possible candidates for fields that could benefit from an experimental design description language are biology and machine learning; in both biology and machine learning, however, the experimental designs describe the set of trials to consider and lack a notion of ordering. This means that SweetPea would likely need to support more semantics for describing subsets of the crossing space, and applying constraints to these subsets.

7.1 Language Wishlist

This section describes some language features which would facilitate supporting a wider range of experiments.

Windows on Counting Constraints Currently, counting constraints are specified using the `AtMostKInARow`, `ExactlyKInARow` or `AtLeastKInARow` constructors. These operate over the entire experimental sequence, which is limiting in the constraints they can express. If these constraints were expanded to operate over a window, in the way that derived levels do, they could be used to express a wider range of desired constraints, including more complicated balancing constraints.

```
Constraint = [AtMost][AtLeast][Exactly]KInARow K (FactorName, LevelName) Window
```

General Case Derivations Currently, SweetPea can handle transition and congruency constraints. These are common constraints in task switching experiments, but to be able to represent a wider variety of experiments, SweetPea needs to handle arbitrary derivation functions. Recall that a derivation function is a python function with the function signature:

```
DerivationFunction = ( Listof( FactorName )... ) -> Boolean
```

Derivation functions are used by windows, along with a list of factors to which this function should be symbolically applied to:

```
Window = Stride Width DerivationFunction Listof( FactorName )
```

To encode any function which takes some number of lists of factor names, and evaluates the truth value of those factors, we could use a truth table encoding. This means that we could generate a truth table which represented every possible combination of the input variables and compute what the output value of the derivation function would be for every input combination. We could then allocate a new level by specifying that that level is true iff (all the rows of the truth table which are true). We could then further simplify this truth table by the Quine-Mccluskey algorithm. This approach would be limited by the number of variables in the truth table, and another approach may be necessary to support designs which express conditions that depend on the state of many boolean variables.

Weighted Crossings It would be useful to have a mechanism to specify a way to over or under sample a full crossing. A usecase for this is a version of the Stroop experiment with 3 colors; in a full crossing there are 3 congruent stimuli, and 6 incongruent stimuli (see **Figure 7.1** on the following page). It is well-motivated to want to balance the number of congruent and incongruent stimuli. One can imagine either undersampling the incongruent stimuli (choosing 3 of the 6 with uniform likelihood), or oversampling the congruent stimuli (such that each congruent stimuli occurs twice). More generally, it will be worth spending time in the future investigating more ways of specifying parts of the space of the

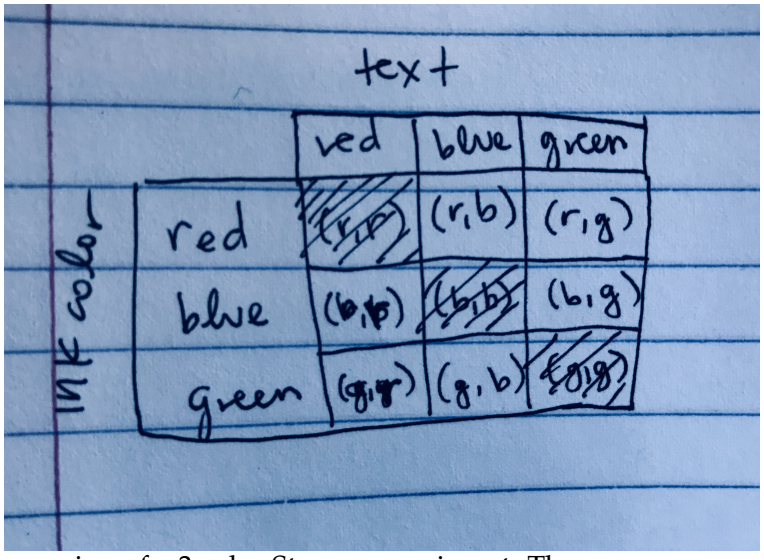


Figure 7.1: Full crossing of a 3 color Stroop experiment. There are more congruent stimuli (along the diagonal) than incongruent stimuli.

full crossing.

Sampling Continuous Factors Some experiments may contain levels which represent continuous values, such as sampling colors from the continuous color spectrum. We can currently represent these experiments by binning the continuous values into a number of discrete levels– but can we do better? This is challenging because the translation to SAT is necessarily discrete. Perhaps this could be supported as a pre- and post- processing step, or perhaps Unigen could natively support this. The user could specify the continuous value, and SweetPea could performing binning, find a solution, and add random noise to the solution to simulate sampling from a continuous distribution.

Automated Experimental Design In the spirit of declarative programming, it would be excellent to extend SweetPea to automatically derive the minimal set of counterbalancing conditions that satisfy the specified statistical analysis. Specifically, we may wish to provide an interface for the ANOVA (ANalysis Of VAriance) technique, which would allow researchers to state "contrast red with blue" instead of explicitly stating a design that does so. This higher-level interface will make it easier to specify experiments that accurately match the desired statistical analysis. Perhaps such an interface would also support iterative and interactive experiment specification. For instance, if multiple experiments match a desired statistcally analysis, perhaps the user can use SweetPea to explore the options and tradeoffs of each of those possible experiments.

7.2 Runtime Wishlist

This section describes some runtime features which would facilitate a more interactive and usable environment.

Verified Core SweetPea promises to make it easier to write correct experiments; this promise can only be true if SweetPea itself doesn't contain bugs. The boolean encoding parts of the code, especially the counting constraint encoding, is ripe for bugs because it produces a CNF formula, which is difficult for humans read and to verify at a glance. Currently, we protect against bugs through exhaustive testing. In the future it would be wonderful to do one better, and to formally verify that these transformations are correct.

Debugging unSAT experiments If a user writes down an experimental design which is overspecified, the SAT-sampler will state that the constraints are unsatisfiable. This is not very helpful– why is it overspecified? Developing debugging support would create a more pleasant user experience. One possible approach is that some SAT-solvers can provide a *minimally unsatisifiable core*, which specifies what set of variables lead to a contradicting assignment. It may be possible to use this minimally unsatisfiable core to translate back to which constraints over which levels are reported to be contradictory.

Iterative Experimental Design and Partial Satisfiability Extending the idea of adding support for debugging unsatisfiable experiments, it would be great to have SweetPea be a tool for exploring the edge of satisfiability. For instance, perhaps a researcher could attempt to specify an ambitious experimental design which tests multiple things at once: perhaps in the future SweetPea could allow them to iteratively explore which subsets of their overconstrained design are satisfiable. One technique for facilitating this is some SAT solvers allow you to push / pop individual clauses; perhaps this fine-grain precision can be used to find the most constraints which can be simultaneously satisfied.

CHAPTER 8

CONCLUSION

This thesis presented SweetPea, a system for concisely describing experimental designs and rigerously generating experimental sequences that satisfy the design. 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.

SweetPea provides high-level notions of factors, levels, derived levels and factors, and counting and balancing constraints. Psychologists can use this high-level language to specify their experimental designs, and SweetPea will encode those descriptions as a boolean formula. SweetPea guarantees that each experiemental sequences which satisfies the design is approximately uniformly likely. Guaranteeing that each sequence is exactly uniformly likely is an intractable problem because it is #P-hard for a large problem-size; we gain traction on this problem by making the acceptable compromise of guaranteeing that the sequences are approximately uniformly probably. SweetPea achieves this promise by calling out to the SAT-sampler Unigen, which in turn uses universal hash functions to get uniformity.

SweetPea can be used to model common task-switching experiments such as variants of the Stroop test. We expect that in the future SweetPea can be used to model a wide range of experiments in psychology and other experimental sciences.

REFERENCES

[1] R. Bodík, K. Chandra, P. M. Phothilimthana, and N. Yazdani, *Domain-specific symbolic compilation*, in LIPIcs-Leibniz International Proceedings in Informatics, vol. 71, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[2] S. Chakraborty, K. S. Meel, and M. Y. Vardi, *A scalable and nearly uniform generator of sat witnesses*, in International Conference on Computer Aided Verification, Springer, 2013, pp. 608–623.

[3] 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.

[4] K. R. M. Leino, *Developing verified programs with dafny*, in Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, 2013, pp. 1488–1490.

[5] 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.

[6] S. E. Maxwell, M. Y. Lau, and G. S. Howard, *Is psychology suffering from a replication crisis? what does failure to replicate really mean?*, American Psychologist, 70 (2015), p. 487.

[7] 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.

[8] J. I. Myung and M. A. Pitt, *Optimal experimental design for model discrimination.*, Psychological review, 116 (2009), p. 499.

[9] L. Nelson, H. Sigurbjarnarson, K. Zhang, D. Johnson, J. Bornholt, E. Torlak, and X. Wang, *Hyperkernel: Push-button verification of an os kernel*, in Proceedings of the 26th Symposium on Operating Systems Principles, ACM, 2017, pp. 252–269.

[10] H. Pashler and C. R. Harris, *Is the replicability crisis overblown? three arguments examined*, Perspectives on Psychological Science, 7 (2012), pp. 531–536.

[11] H. Pashler and E.-J. Wagenmakers, *Editors introduction to the special section on replicability in psychological science: A crisis of confidence?*, Perspectives on Psychological Science, 7 (2012), pp. 528–530.

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

[13] F. L. Schmidt and I.-S. Oh, *The crisis of confidence in research findings in psychology: Is lack of replication the real problem? or is it something else?*, Archives of Scientific Psychology, 4 (2016), p. 32.

[14] D. J. Simons, *The value of direct replication*, Perspectives on Psychological Science, 9 (2014), pp. 76–80.

[15] C. Sinz, *Towards an optimal cnf encoding of boolean cardinality constraints*, in International conference on principles and practice of constraint programming, Springer, 2005, pp. 827–831.

[16] W. Stroebe, *Are most published social psychological findings false?*, Journal of Experimental Social Psychology, 66 (2016), pp. 134–144.

[17] J. R. Stroop, *Studies of interference in serial verbal reactions.*, Journal of experimental psychology, 18 (1935), p. 643.

[18] 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.

[19] L. G. Valiant, *The complexity of enumeration and reliability problems*, SIAM Journal on Computing, 8 (1979), pp. 410–421.