Context Aware Data Generation Through Domain Specific Language

Caleb Druckemiller

Follow this and additional works at: https://digitalcommons.harrisburgu.edu/csms_student-coursework

Recommended Citation

This Dissertation is brought to you for free and open access by the Computer and Information Sciences, Graduate (CSMS) at Digital Commons at Harrisburg University. It has been accepted for inclusion in Other Student Works by an authorized administrator of Digital Commons at Harrisburg University. For more information, please contact library@harrisburgu.edu.
Context Aware Data Generation Through Domain Specific Language

by

Caleb Druckemiller

Thesis submitted to the Faculty of the Graduate School of the

HARRISBURG UNIVERSITY
OF SCIENCE AND TECHNOLOGY

in fulfillment of the requirements for the degree of the
Master of Science in Computer and Information Sciences

Supervised by: Abrar Qureshi, Ph.D.

2020
Abstract

Software engineers frequently find themselves relying on context aware data which is driving the need for the ability to generate said data on a mass scale with complex schema validation. Generating this data is quite expensive, which interns causes development cost to rise. This also increases the level of entry for the software tester to because they need to create more and more complex data structures. A domain specific language tailored to this task would accelerate the process of software testing without the need for massive training on a new platform or tool. Domain specific languages are the epitome of modularity and flexibility while maximizing proficiently. The language proposed will supplement and replace the need for privacy invading web scraping for organic data from live specimens and the need to hire expensive data scientists or data engineers to produce this data with complicated algorithms to handle the various schemas required. The requirements for the language will be a definition of the schema, validation rules, and desired output. Provided these minimal inputs a dataset can be generated to fit the needs of software testers and software engineers writing unit tests for their applications. Data driven applications require millions of records worth of data to be completely tested and the generation of this data comes with high costs. A tool that is designed to leverage modern computing techniques, modern computer hardware infrastructure, and new approaches in data generation would greatly reduce the associate costs of this data generation while enabling software engineer, testing engineers, and data scientists to produce large quantities of data at a high degree of flexibility and complexity in design.
Keywords: domain specific language, context aware data, software testing, unit testing, data generation, data science, data, smart data, recommender systems, schema validation, lexical analysis, context aware applications, data driven applications, context aware services, software engineering
I. INTRODUCTION

My thesis is on a new pattern for the creation of test data for context aware applications. Data driven applications are becoming more and more apparent, especially in user facing interfaces, web apps, dashboards, e-commerce, etc. E-Commerce is a thriving market for web applications and that is a vastly data driven domain. This data can be extremely complex in these applications with schema’s that are very difficult to reproduce in anyway but manually. The various shapes of these data sets can also be vast and complex, making it hard to create a truly random dataset that hits all the edge cases that the application can confront in the wild. The need for a way to automate this system is a great frontier of exploration that a computer scientist can exploit for his or her benefit. Many apps exist with a graphical user interface that generate data such as Synner and Mockaroo, however these applications greatly hinder the user’s ability to make custom datasets with complex schemas that have cross schema dependencies and inter schema dependencies without doing some extra programming in a true general purpose programming language. My suggestion would be to introduce domain specific language that can generate the data and be customizable to help solve the issues with complex schema validation. A DSL (domain specific language) is slightly more difficult to learn than a GUI (graphic user interface) but provides a much richer amount of customization and utilization for things like automation. With an adequate amount of training, a DSL would not be too difficult for anyone to learn who is at least semi-technically inclined (such as a software engineer or a software tester).

This is a new domain specific language will be used for generating test data with complex schema’s that have variable generation rules that are dependent on other aspects of the schema. A tool like this would be invaluable for any software tester or engineer. My motivation behind this project is my time spent as a software tester and software engineer. Software testing
is a tedious process that is not always difficult but very time consuming. Automating the testing process is rather difficult, especially making test data that is applicable to complex, data driven applications. A DSL lends itself to being easily automated with other command line tools like Bash or Zsh.

Command line tools are familiar playgrounds for most software engineers. In my experience, most developer like working with their terminals and command lines more than a graphic interface. Therefore, I pushed for more of a command line tool. Many of my fellow developers why working in the industry prefer command line tools like Git and file system manipulation with bash and Zsh, as well as running Node modules and other programs through the command line. Command line tools give super users like developer much more control over their file systems and programs than regular interfaces which is why most developer tend to lean towards using these tools. Also, many developers are “lazy” developer and like to automate tasks for themselves. Command lines are essentially mini programming environments which let them script things out. Scripting was the earliest form of programming and thus developers have an affinity for these types of tools, in my experience.

A DSL also solves an issue with the use of non-synthetic data. Non-synthetic data is the product of mining data from actual users. This comes with a vast amount of privacy issues as well as cost in maintain servers and paying for webservices to run the programs that are scraping this data. There are more and more privacy policies being ruled out to the general populace, especially in Europe. European privacy laws are cracking down on malicious software that is monitoring web activity of users without their consent. This type of legislation is slowly making its way to the US in states like California. It is important for companies to stop prying into their user’s privacy and this DSL could help them save on their monthly server budget. The DSL
would create synthetic data which is, ideally, random data the mimics true user data with outliers and a desired distribution. Synthetic data poses its own issues with the complexity of data generation and paying a data engineer to create the algorithms to generate the data. That is where the DSL will shine. Instead of paying a data engineer tens or hundreds of thousands of dollars a year or more depending on if the company needs a team of engineers instead of just one., they can let their developers create the test data, or the test engineers, or just less skilled and lower paid individuals generate the test data using the DSL.

Like any new tool it will eliminate some jobs for those who adopt it but overall, the decrease in cost of data will make other aspects of life cost less. Software will be better tested, more consistent and encompassing patches will fix more bugs because ideally more bugs would be found in development or in the testing phase than out in the wild causing problems for users. In general, it is a positive thing when software quality is increase and anyone who is reliant on that software, even if they are not using it directly, will see a modicum of improvement in their life. That is what this DSL is all about.

The problem that is commonly occurring in software testing is creating complex datasets that cover all edge cases that a program can encounter in the wild, while also being truly random in nature to not introduce bias. Testing edge cases is important, but it is also important to test the applications interaction with the normal data that it will receive daily. If the application is choking on regular data, then testing edge cases is essentially worthless. Create a domain specific language that can create complex data schema. The language needs to be able to handle variable data schema conditions, be automatable, export data to various outputs: JSON, Database, Excel, CSV. The language also needs to be easy to learn for non-technical people or else it will never be adopted in common business or industry settings.
I’m going to take a traditional approach to the compiler design and the lexicalization. I obviously want to avoid memory leaks but the first run through will have faults that need to be fixed. I’m going to be used MPS from JetBrains and any other tools that I find helpful. A goal for my first pass through will be to closely emulate Mockaroo through a DSL. I plan on using JetBrains’s MPS tool for creating the language. MPS has been a standard tool in the developer's tool kit for a few years now and it is what JetBrains used to develop the Kotlin programming language, now one of the highest utilized programming languages on the market, mostly thanks to android app developers. I have several design patters for the tokenization that I plan to follow, and I have an idea of the syntax that the language will. It will look like a crossbreed of JavaScript, Python, and SQL. The simplistic, minimal design of the three languages along with the JSON object structure will help keep the language familiar to developers while maintaining the simplicity for non-technical users to learn quickly.

After the development of the language I’m going to introduce it to a group of testers that are not technically inclined and have them use the language for a few days. After a weekend of not using it I will have them come back and see how much they retain and see if they feel more productive than if they were creating the test cases manually. The end goal of this tool is to help non-technical users feel empowered enough to make rich test suites using the DSL and for developers to be able to easily integrate the DSL into other tools like bash and Zsh in order to automate testing in their development and deployment pipelines. Many continuous integration tools like Jenkins and Team City offer shell script automation as part of their process and this DSL would fit nicely into a docker container to create the data and build a database of test cases. These test cases then be pumped into a command line runner, such as Newman, to interface with their applications. Integration tests can be written and automated and thus a more robust and
streamline deployment process is created that helps minimized bugs passed between development and testing and then testing and production.

I have received feedback from other software developer, and they have said that a tool of this caliber would be extremely beneficial. An experience software engineer that was my immediate superior at a previous position showed great interest in the idea. This type of tool, if introduced at my previous employer’s company would enhance software testing beyond any recognition in the state that it was when I left. With form-based applications, the datasets could be fairly complex because one form input could be left blank if another form input was filled in but it was still optional to fill in and all these sorts of rules would spiral out of control in testing because trying to find a way to encompass all of these outcomes in the form completion would be impossible without literally making hundreds of passes through the form. And that is what our testers did. With the DSL, the data for the form fields could have been generated randomly and then created the test data could be innumerable and then the testing would be more complete, more thorough, and much more expedited. The goal of software testing at the end of the day is to a complete test of all possible incoming data and thoroughly test all edge cases. Through this DSL those goals can be completed much faster and measurably more random and thorough at the same time.

II. BACKGROUND

To benefit comprehension of the topic, this section will briefly break down the main topics of the paper; context-aware data and domain specific languages.

I. Context Aware Data

Breaking down the three-word term helps in the understanding of the overall concept.
Context awareness in the aspect of computing and computer sciences is defined as “the ability of a system or system component to gather information about its environment at any given time and adapt behaviors accordingly.” (Rouse, 2016). In the aspect of data, we can think of the data as knowing what is in itself, the dataset, and thus telling whatever is operating on it what protocol or procedures to use. Imagine a web service that accepts a JSON (JavaScript Object Notation) payload. The web service can accept either a correctly defined JSON payload or an error JSON payload. An application that is implementing a context aware data consumption engine will be able to validate if the JSON payload schema is valid or not, as you would expect from a backend sanitization service. But, if the payload received contains a certain attribute in the schema, perhaps the decision engine for the software also requests other data from other parts of the entire system. Using the peripheral information retrieved, beyond the scope of the original request, the decision engine can then suggest better results for the request without asking the user for additional information. Using the context in which the current system resides, or the state at which the software is in, the application performed a task differently to the same request. This is not a bug, it’s a feature! A function will always return the same output if given the same input but now our smarter functions are getting inputs then also grabbing a little more data from outside sources and making non-deterministic results to help improve the quality of life for the application user.

A short anecdotal interpretation of a context-aware system in action: A smart phone consumes generous amounts of data every minute. While traveling to a meeting a user makes a request in their smart phone for food. Within the smart phone’s calendar, the user also has recorded that they have an appointment at a location approximately thirty minutes of car travel away, based on the current traffic being reported from other cell phone usage. The phone
can tell that the user is currently traveling in the direction of the appointment’s location and will arrive approximately fifteen minutes before the appointment is scheduled. Using this information, the smart phone can then use web services provided by local restaurants to receive approximate wait times for their services. This smart phone also has access to the user’s transaction history from a credit card that has been stored in the smart pay service of the phone and knows from which restaurants the user typically purchases from and frequents based on geographical data. From a simple request for food, the smart phone’s context aware service has consumed data from several external web and internal local API’s to make better suggestion for food to the user. The goal is for the top result to be the one that user selects for their meal. The smart system also has access to previous selections from when the user used this service before as well and every successive use helps better train the recommender system generates better results. A large portion of these computations are artificial intelligence-based systems, but the beginning data consumption is simply the ability for the system to be contextually aware of the request and what circumstances are “on the table” so to speak for the user. In a quote commonly misattributed to Albert Einstein but never fully confirmed in origin states: “The definition of insanity is doing the same thing over and over again and expecting a different result.” This may be true but every successive attempt for any context aware application is a new world of inputs so the same request ran multiple times can truly result in a different outcome, thus making these systems truly non-deterministic.

II. Domain Specific Language (DSL)

A DSL is a tool used commonly by software engineers, scientists, and occasionally software testers. It is a language tool similar to a traditional general-purpose programming language but with a much more limited scope. A general-purpose programming language must account for all
use cases of programming in the syntax such as looping, flow control, variables, function calls, etc. A domain specific language has a singular use but does this job extremely well, (performant), and should be relatively easy to learn (as opposed to a traditional programming language) due to its shrunken vocabulary. The languages verbs, or operational keywords should be intuitive and describe the action taken effectively to the point that limited exposure should result in proficiency with a quick turnaround. An excellent example of a DSL is SQL, or structured query language. SQL is a DSL targeted for querying databases efficiently. While there exists a vast amount of different “dialects” of SQL; PostgreSQL, MySQL, MongoDB, DB2, etc. the core of the language is the same and the skills translate easily. DSL’s are simple tools that give an increased level of customization and utility over traditional GUI’s while maintaining relative simplicity in operation.

III. LITERATURE REVIEW

I. UML Base Designs

A significant amount of work has already been done in the pursuit of simplifying context aware data generation. Unified Modeling Language (UML) based solutions are being explored by several scholars in recent years. Quan Sheng and Benatallah examine the various expenses that go into creating context-aware data similar to the points discussed previously. A UML based solution has the advantage of being more easily consumed by professionals with an engineering background; UML acting essentially as a second language for most engineers regardless of the discipline (mechanical, software, architectural, etc.). Sheng et all focus on the use of context aware datasets in the domain of web services, like the example of the JSON payloads. Their context aware service example uses the location of a user to suggest restaurants and if the weather is nice (using the location information and sending that to a weather service to get
details about the weather in the area) it will give the user suggestions of place that have outdoor seating. The service requested collected data from the user and used this data to suggest may not have been considered by the users. If the weather was harsher then perhaps the recommender system would have suggested places that sell hot beverages like tea or coffee (Sheng and Benatallah, 2005). Context awareness provides a whole new level of system analysis when absorbing data through API’s but also leads to API’s requesting larger payloads and needing more power to process the data. With the evolution of processing power in smaller devices though, this is becoming more of a non-problem. Mehmood, Khan, and Afzal also are pursuing a UML based solution for typesetting in data models. Their process is more of a working backwards model though for me, they are using context aware data to generate the UML models for the sake of typesetting (Mehmood, Khan, Afzal, 2018). Studying this though, future progress can be made going in the reverse direction making the data “smarter” and being able to easy ingest and programs running with less intelligence baked into them gives room to grow in other sections of applications.

II. Software Testing

One of the most important aspects of software engineering and development is testing. The previous scholars, Mehmood et all, also took steps in generating context aware data sets for the purpose of software testing. Data driven applications are becoming more and more mainstay and consuming data from API’s is becoming the norm. Their suggestion of needing automated data generation for context aware applications aligns with the DSL proposed my research as a possible solution to this problem (Mehmood, Khan, Afzal, 2018). Scripting a DSL’s execution is considerably easier than macroing a GUI or some other tool or program, i.e. tools like awk, and sed, in shell command line scripting languages. These tools are essentially DSL’s with limited
grammar for text processing. Enabling software testers to be more productive, running more testing cycles per release, testing more fringe case data sets, and more brute force attempts to break software ideally would lead to more robust software solutions. This also should lower the amount of knowledge a software tester possesses about the software which is better because this yields more Blackbox testing of software and thus more closely mimics live user testing in production. Again, ideally this would result in few bugs in production and more robust software being released.

III. DSL Advantage

Domain specific languages are not exactly a new concept in scientific computing but have been receiving more notoriety with imbedded systems and IoT (internet of things) devices. Hinsen in a recent IEEE (Institute of Electrical and Electronics Engineers) paper discusses exactly this. DSL’s come with some negative connotation in the software development industry due to their lack of developer creature comforts like syntax highlighting, auto-complete, and IDE (integrated development environment) integration for compilation error detection. However, with all these tools missing, a good software developer should still be able to mantle these hurdles and leverage a DSL into an excellent tool. DSL’s are generally quite performant spatially and temporally due to their limited scope needing to pull in only small sections of a larger compiler or a custom compiler all together and the function set doing a limited amount of operations (Hinsen, 2018).

DSL’s are excellent tools for use in imbedded system but are also applied appropriately and effectively in other areas of software development. Data generation is an excellent candidate for a DSL due to their limited scopes and iterative processes. Data generation at its core is iterating the same task continuously with slightly different input parameters as options. Software
engineers can liken this to iterative function calls. DSL’s can be simplified in understanding to function calls with more flexible input parameters. This is why DSL’s are excellent for tasks of this nature as well as other tasks include abstracting concepts while still giving the user a range of freedom. A DSL’s application is only limited by its grammar and the creativity of the person using the language.

IV. Context Aware Systems

Context aware systems have become such a commonplace software product that most software engineering projects are pushing towards these models. Context aware systems are projected to become even more popular over the next few years and we can expect to see almost all consumer facing software to support some type of quality of life improvements based around cognitive systems imbedded into the software (Context Aware Computing Market - Growth, Trends and Forecasts, 2019). Many endeavors have been made in recent years to help promote the growth of context aware systems and their testing. It is difficult to generate test scenarios for context aware systems with the large quantities of variable data. Developer are essentially trying to solve non-deterministic problems with a deterministic dataset, which is immensely difficult. Many scientists have tried creating test environments in which they can test cognitive systems, such as smart home devices like Google Nest and Amazon’s Alexa. Scientist at Cornell University have created a system in which they can mimic a home environment to test these tools in, however the procedures are quite taxing, and a better test data set could be presented (Nguyen, Nguyen, & Choi, 2011). Hence, the usefulness of the DSL to create context aware datasets for the experiment and other experiments would be extremely useful.
IV. SCIENTIFIC ENDEAVORS

The goal behind the work of creating this language is not simply to better the lives of software engineers and software testers. Software applications are currently consuming vast amounts of data, especially data driven applications and cognitive applications that consume data anonymously in the background, such as the “smart” applications running in web browsers, smart phones, and smart home devices like Amazon Alexa and Google Home Speakers. This is a trend in many artificial intelligence based devices and software and will likely continue to be a trend for the foreseeable future. The goals are numerous for this project: decrease the reliance in minded data that intrudes on user privacy, help research better test simulations for healthcare breakthroughs and other fields, provide academics with a tool that can be used to more specifically target their data needs with complex dataset generation and schema validation protocols, and finally reassert the value of domain specific solutions for problems in academia, in contrast to general purpose solutions such as what is used in the modern day software engineering industry.

V. USE CASE ANALYSIS

Previously in this proposal, it has been discussed the myriad of uses for such a tool. I want to analyze these uses further and their various benefits. Obviously, the overarching purpose of tool is to create datasets. Datasets in themselves can be used in nearly infinite ways, even in their most basic forms. Datasets can be used for software testing, bootstrapping data for datamining and analysis, and mocking production environments for showcasing applications, mostly for marketing purposes plus so many more useful applications of generated datasets. This tool, however, is much more than a blind data generation tool. The tool that I am suggesting and describing is a tool that can create data in a much more intelligent manner, smart data. Smart
data, as described before as context aware data, is not necessarily anything special with the data itself, it is more about how the data is created. With a smart data creation tool, users can create data that is more coherent and adheres to the rules of the system that the user is trying to replicate. More accurate data, or even just data that follows the rules of a system more accurately allows users to more closely replicate live production environments in software engineering and the testing of said software.

Beyond the scope of industry, academia needs a tool to create complex datasets for the testing of their computer simulations in fields such as infectious disease, clean energy, and cellular growth. These complex simulations often utilize artificial intelligence and cognitive or context aware data ingestion services. The validation of these systems is of higher stakes than simple profit loses or user approval rates. These simulations offer solutions to humanities greatest problems such as global warming, disease transmission, and sustainability of our species.

Software engineering has also invaded the realm of academic computer science. While industry influence is not necessarily a bad thing, modularity of design is pattern of engineering that is not highly effective in the most advanced sects of scientific computing and high-performance machines. Acute problems required acute solutions and a domain specific language is a highly effective, acute solution to the problem of dataset generation. Academic reliance on software engineering patterns has become a crutch and distancing scholastic pursuits and industry pursuits would be beneficial to both sides to help better the respectability of the computational sciences and software engineering as a trade. A line of separation helps better define the two schools of thought and make progress in both fields that benefit each other in the long run.
VI. METHODOLOGY

I. Introduction

A domain specific language is broken down into two parts, the lexical analyzer and the parser. The lexical analyzer is in charge or breaking down character inputs into the tokens or words in a sentence and parser the oversees ingesting the tokens and determining their “part of speech” or the category they belong to in the sentence adhere to the laws of the language, the grammar. There is a precursor step however of determining predefined keywords and what those tokens are and how they fit into the language. These tokens in a general-purpose programming language would be placed into buckets like constants, variables, keywords, access modifiers, etc.

II. Tokens

The tokenization starts with the keywords for the language. The keywords that are going to be presented in the language are going to correlate with the actions that the language is going to be taken. You can think of the initial set of keywords as the verb because they will be the operative function calls. The important duties that the language needs to perform are generate data, ingest a data model, determine weights and spreads of the data set, perform schema validation, allow custom schema validation, cross schema validation and dependency, and schema definition. I want to match a keyword to each action and the keywords need to be intuitive and descriptive without being verbose or wordy. The language is going to work in a two-step process. The first step will be the schema initialization whether it is one or multiple schemas working together, before the data can be generated the schema(s) need to be defined. The second step is obviously the data generation. This seems like the simpler step but in this step, users can define weights, quantities, noise levels, etc. which are important when user want a
more closely mimic organic data. Users will also be able to select their desired output whether it be a JSON payload, CSV, or database file.

- **GENERATE** – This will be the keyword that will start every sentence of actually generating a dataset. I think of this as my “SELECT” keyword similar to what is used in SQL. I also like the word PRODUCE for this keyword as I feel it is more descriptive and user friendly.

- **DEFINE (MULTIPLE)** – This will be the keyword that will start every schema validation statement. The MULTIPLE modifier will be used when the user wants to define multiple schema. Part of me thinks that multiple schema definition will be the norm, so I have plans in the future to make it so DEFINE works on multiple schemas by default and change MULTIPLE to SINGLE. I also like the word “DESCRIBE” instead of DEFINE, it might be more intuitive. I will need to go through some user testing to determine which is a more natural fit.

- **DEPENDS ON** – This will be the keyword that determines whether or not a value is applicable in a dataset conditionally on a member of the same dataset. Example: don’t collect job title if person is under legal working age. This will be in the DEFINE statement. The following token should reference a singular or a comma separated list of features in the same schema.

- **RELIES ON** – This keyword will work similar to DEPENDS ON but will be used when using cross schema validation. The following token should reference a singular or a comma separated list of features in another schema.
• **AS [JSON, CSV, DB]** – The as x will be used to determine the desired output and must come at the end of a **GENERATE** statement.

• **AUGMENT** – This keyword will be used after the definition of the feature in the dataset to “augment” the data into a spread that makes sense. Augment is not a commonly used word in the English language (Google NGram, n.d.) as opposed to a word like *change* which is used nearly one hundred times more often, (Google NGram, n.d.) but change did not feel descriptive enough to be considered intuitive. Augment I feel is descriptive and still not too obscure a word to be used in the language.
  
  o These custom augmentations are going to be tricky to create grammar rules for because the customizations I want to let the user do are literally infinite. The goal of the DSL is to let the user define their data any way they want, and have it correlate how they want.

• **PREVIEW [X]** – This keyword is similar to a **DRY RUN** keyword in an SQL based language. It gives the user a preview of the first X records to be generated. This is useful to determine if the shape of your data is to your satisfaction. The **PREVIEW** command should be followed up with an **AS** command to tell the compiler what the preview should look like. The default will be a comma separated list of values displayed in separate lines in the standard output.

For the data types of the dataset I want to have a variety of supported baked in types for ease of use. Obviously, the language will support numbers in all their varieties, integers, floats, decimals and string data types whether its single characters or multiple character strings. I also would want to support fixed length list with “*enum*” like structure. I found these to be a common use case in
selecting things like client names, days of the week, etc. They can also be useful when you want to have a random selection of single letters from the alphabet, a fix twenty-six length array index for each letter. A simple example sentence I have been basing my ideas around reads like this:

```
DEFINE FOO {ID: SERIAL, FIRST_NAME: STRING,
LAST_NAME, AGE: INT, OCCUPATION: @STRING DEPENDS
ON AGE > 18, SCHOOL: @STRING DEPENDS ON AGE <=18}
AUGEMENT FOO.AGE MEAN 35 STANDARD MIN = 13 MAX =
70 FOO.OCCUPATION = [DOCTOR, LAWYER, ARTIST,
ENGINEER, LABORER] FOO.SCHOOL = [ELEMENTARY,
MIDDLE, HIGH]

GENERATE FOO AS JSON:1000000 WITH NOISE LEVEL: 3
OUTLIERS: 5%
```

The declaring structure of the initial schema is similar to a JSON format with some minor tweaks when adding the dependencies. Another key component is the use of the @ symbol when describing SCHOOL and OCCUPATION. I used the @ symbol to declare that it was going to be a fixed length list. The compiler will not run if the user declares a fixed length array for a data type and does not provide the necessary enum values for the dataset to generate with. I think this is a more intelligent design than having it produce an empty fixed length array. The creator of the null reference Tony Hoare, referred to it as his “billion-dollar mistake”, and I agree that
null references are bad. This is why I don’t want to introduce them into my DSL. This wouldn’t create null references in my own language but could lead to null references on the code that is running against the dataset generated from the DSL. If null values are accepted in the dataset I believe this should be indicated upfront in the **DEFINE** statement, possibly with a “?” symbol as that is what is commonly used as a null safe operation in modern languages such as Kotlin (Null Safety, n.d).

I also believe there exists additional room to grow in the generation section with adding a variety of variables for data generation such as adding noise and outliers. The noise level is a generic level of variety to the data to prevent overfitting of the model. Volatility is necessary for natural data. Outliers are also a necessary evil when generating data. The next obvious step is defining clear rules of production to help with syntax highlighting to increase the usability factor of the language. Baric, Amaral, and Goulao have a thorough testing protocol for the usability of a domain specific language that I would test my language against. They also have a suggested standardized metric for grading the usability of DSL’s (Baric, Amaral, Goulao, 2012). Ideally the DSL would test highly in their metrics.

**III. Cross-Schema Dependency Tagging**

There are two schools of thought that the DSL is based around. The first is a concept of cross-schema dependency tagging when creating the schema. What this means is that when a schema is represented in the language, a tag is placed on a field that is going to have a dependency on an aspect or attribute from another field. When the author or developer of the schema declares a tag, the compiler knows to match that tagged dependency with the declared dependent. If no dependent is declared that then that would result in a syntax error.

**IV. Tagged Union**
The alternate to a cross-schema dependency tagging system would be what is known as a “tagged union”. A tagged union is an applied mathematical concept in computer science, much like many other complex terms found in computer science, as the computational sciences in large are applied mathematics. A union between two sets in mathematics and set theory is understood to be all items within two different, but not necessarily disparate sets. A tagged union, commonly referred to as a disjoint union, variant union, or just variant, is a type of data structure that holds several different types of data, but the types are a fixed set; much like mathematic sets. Tagged unions are also commonly known as discriminated unions in antiquated programming languages. The concept should be familiar to most software engineers and programmers who follow the object-oriented approach to development. At the most basic level, this is a representation of inheritance and abstraction. The programming language Kotlin, from the software engineering and technology company JetBrains, implements this idea of a tagged unions in their class structure; a *sealed class*. What a sealed class or tagged union does is essentially creates a superset like “class” or type that can be inherited from but only by other classes defined within the type definition of the sealed class or tagged union. A simple example of this would be having an “HTTPResponse” class that acts as a sealed class for all responses from HTTP requests. A HTTP response can either be the data that you requested, with all the various metadata around it (response code, headers, etc.) or it can be an error with all its various meta data. What a developer can do then is defined fields in the sealed HTTPResponse class to account for the meta data that is shared then in the classes that inherit this, let’s call them HTTPRespSuccess and HTTPRespErr, define the various fields and functions that are specific to an error or a response. Now a developer can easily do this in any language but in Kotlin then when a developer performs an action based off of an HTTPResponse the Kotlin compiler then knows that the
program only needs to account for those two eventualities, an error or a success. This greatly reduces the possibilities of nulls and extra syntax to account for any other eventualities. Thus, imagine a switch statement (called a *when* statement in Kotlin) that only needs to account for those two eventualities and does not need to have a default for fall through. Much cleaner and safer code is created. The same idea applies to the schema declaration. The DSL could define a sealed data type that says the schema can be supplied with data type *a* or data type *b*, or neither. This then allows the data schema’s to be stricter while still being flexible, though this seems paradoxical it is quite accurate. Leveraging a tagged union style implementation in the DSL would result in more freedom for the author of the data schema while easing the complexity of the schema through the limitations of dependencies. Either route would be an acceptable avenue for success, and perhaps both are worthy of pursuit and comparisons can be made to benchmark the success of both techniques for ease of usability, performance, and retainability of the langue’s syntax.

**VII. RESULTS AND FINDINGS**

Domain specific languages, in my experience as a developer, are seen more as a novelty at first before truly adopted into a system. Technology rarely takes off rapidly and disruptive technology is an oddity for sure. The intent of this endeavor was not to create the next “Uber” and take the industry by storm. In fact, the purpose of this pursuit was not industry driven, primary at least. My industry experience initially fueled the idea, but academic interest immeasurably outweighs the desire to create this language. An overarching goal for this entire research was to further cement the necessity of domain specific solutions to domain specific problems. Software engineering speaks highly of the modularity of design, which is a valid notion but for high degrees of performance, a highly specified tool will outperform a general-
purpose tool. The common phrase from Abraham Maslow, "I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail." illustrates this greatly (Maslow, 1966, p. 15).

Focusing on the industry improvements first, despite my disvaluing of the positive effects, the results are not disheartening. A theoretical industry speedup of several days in a deployment process is a powerful metric. The creation of data for software testing in a standard testing cycle can take days. The drastic decrease in time that this automated data generation platform is capable of could result in the minification of that time to a fraction of what is originally is. This tool is also beneficial for software engineers for testing before commits and pushes, thus creating a general increase in software reliability by more comprehensive testing before development cycles end.

Outside of industry, this project has helped validate the necessity of domain specific solution to problems as opposed to modular, general purpose solutions. Modularity of design is an excellent practice in the industry the promotes code reusability and productivity however, in academia, performance is key. Academia does not have the fast paced, user needs of the industry, academics are focused more on the scholastic possibilities of high-performance computing, cloud bursting, and quantum computation to solve non-deterministic problems in shorter time frames. With this in mind, the current tools in place for dataset generation are lacking in the necessary schema validation parameters and the automation control needed for researchers to better and more thoroughly test their experiments. Academic scholars require massive amounts of data for experiments, particularly in microbiology and medical concentrations. Computer simulations that need context aware datasets to simulate cellular generation, the spread of disease, or radioactive decay require complex datasets that this tool can
better provide which will benefit scientist indefinitely. Improvement in this realm of study helps better improve the lives on humanity across the earth as a whole.

**VIII. ETHICAL AND SOCIETAL EFFECTS ON THE WORLD**

A tool with these capabilities would greatly change the world of software engineering and software testing for both the positive and the negative. When creating an innovative tool like this, especially one designed to streamline a process, there is a large likelihood of eliminating the need for human interaction in a system. This human interaction was like a paid position within a company that is now being “outsourced” to an automated tool. Society can look at this as it will and will any number of people will have any number of different reactions to it, but the bottom line is that someone will be negatively affected by new technology in a way that hinders their gainful employment. Companies would likely no longer need to contract out a data scientist to create complex data sets, the developers that needed to be hired would be of a lower skill requirement and thus can feasibly be paid less. Test engineers would have the process shortened and theoretically fewer testers would need to be employed. When a new technology is introduced and is successful, workers will always bear the brunt and owners will reap the rewards. This is unavoidable.

However, as these unfortunate side effects are unavoidable, the positive impacts are also numerous and worth noting. In the day to day life of a software engineer, they would no longer have to waste time generating their own data manually to test their code before pushing to test environments. Software testing engineers would spend less time in their testing cycles generating data manually or relying on developer/data scientists to generate data for them. The developers generating data for testers is also a bad way to introduce bias into your testing procedures which can cause bugs in production where bias does not exist. When a developer provides data to a
tester, typically they are going to provide a dataset that they know works with the program, but this is not something that can be relied upon in the wild (production environment). Eliminating tester bias is a difficult task in itself for most companies and with developers providing data to testers, this becomes even more problematic.

Outside of the realm of software engineering within the industry, the societal impacts ideally would all be positive. An increase in testing in any capacity theoretically would increase the quality of a product. The product that this DSL would be targeting would be context-aware data driven applications, which are commonly found in one of modern, first world, societies most commonly used tools; smartphones. Smartphones are constantly collecting data about the user’s activities: geographic, temporal, search trends, website utilization, app utilization, what content they subscribe to, and so much more is flowing through a user’s smartphone and all of this data is used to make suggestions to help guide and benefit the user throughout their day. An improvement to the application of this tool would be noticed by literally millions who utilize their smartphones daily.

Another aspect of positive impact on society en masse would be an increase in privacy. After the previous section, this may be somewhat ironic in that smartphones are essentially tracking user’s every movement and everything they look at but, privacy is still a concern. To generate this type of data organically, web scrapers would run and pull information from users off of sites like Amazon and determine their interests and generate context aware data that way. With this tool, ideally web scrapers would become obsolete and software could be tested with data generated from the DSL and thus not invade the privacy of everyday users. This would also decrease cloud computing costs in both monetary amounts and energy wasted, decreasing global warming. This last one is a bit of a stretch but in a cloud-based world and data center’s running
day and night every night of the year for years on end, any amount of energy saved helps decrease emissions.

IX. CONCLUSION

Domain specific languages fill the need for many tasks in software engineering and software testing. With the demand of test data for data driven applications, the need for this data to be producible in an automated format, the complex validation and schema designed required makes the leveraging of a DSL fit perfectly into this design. Context aware applications are becoming more commonplace in the computer science community with the advent of artificial intelligence and increased production from smaller and smaller hardware and imbedded systems. DSL’s are regaining momentum in the software development industry after being dormant and singularly used in IoT devices and imbedded systems, to widely accepted solutions in a variety of applications. Through the power of the DSL, data can be generated to more completely test data driven context aware applications and help software testing engineers be more productive in actually testing instead of wasting countless hours generating data. Data driven applications are running on data sets that are becoming more and more complex in their schema validation requirements and weighting. Synthetic data is beginning to match organic data much more closely and the data generation algorithms are able to introduce more noise and natural outliers with smarter computing practices. The time has come to match a tool with a high degree of usability and customization to the power provided by these algorithms and the power provided by modern hardware technology. The best way to fill this void is with a well-executed domain specific language that leverages all of these new and powerful technologies in one unified platform. Thus, enters the data generation DSL that I am proposing which meets all the aforementioned criteria.
X. FUTURE WORK

I plan to further perfect the data generation language and implementing a natural language processing (NLP) portion to the language. The NLP section ideally would allow standard businesspeople and non-technical workers to define a dataset and schema in plain English and have the DSL generate a dataset that adheres to what was described by the individual. The application of this NLP part is not within software testing. I believe this could be highly leverageable for generating mock data for demo purposes in data driven applications. Imagine doing a sales pitch with data specifically tailored to the clients you are pitching to, their season sales habits, their key clients, etc. Setting this up would generally take a businessperson the better part of the day in excel then a software developer the better part of the day inserting that into a database properly for the data driven application (this from my own personal experience in the software development industry). Describing what you want the data to look like would only take a few minutes though.

Beyond the efforts to create new technology, a proper plan for detecting, handling, and giving suggestions around syntax errors is a logical next step. Most proper programming languages and even highly used DSL’s like SQL offer help when creating the syntax for a statement. These kinds of suggestions are extremely helpful for new users and even users who are returning after a hiatus of minimal use. Developer ideally won’t be using this tool daily so some kind of templating for getting “quick started” or bootstrapped or whatever they may call it would be helpful to minimize the startup time for creating a data schema definition language.

This a is a vast amount of improvement in time saving and enables developer to work on feature enhancements more than data creation to better impress clients at the demo. The applications of the NLP pattern are vast and not limited to this example, but this is what I foresee
being the most profitable and effective way to leverage the tool. The NLP portion would be a huge undertaking, but the bulk of the work would be in translating the NLP production rules to grammar compliant sentences in my DSL. There exists plenty of library for ingesting NLP so I can focus more heavily on the translation protocols thus making the tool more robust and accurate yield a less frustrating experience with non-technical users. The goal for these technologies should always be usability especially when the targeted users are non-technical, business minded professionals.
References:


https://books.google.com/ngrams/graph?content=augment&year_start=1800&year_end=2019&corpus=15&smoothing=0&share=&direct_url=t1;augment;c0#t1;augment;c0.


https://books.google.com/ngrams/graph?content=change&year_start=1800&year_end=2008&corpus=15&smoothing=0&share=&direct_url=t1;change;c0#t1;change;c0.


Insanity Is Doing the Same Thing Over and Over Again and Expecting Different Results (2019, September 7). Retrieved April 4, 2020, from

https://quoteinvestigator.com/2017/03/23/same/


voice revolution. Retrieved April 5, 2020, from


Salary: Data Scientist. (n.d.). Retrieved November 8, 2019, from

Annotated Sources:


The authors highlight a tool called Synner that focuses on usability while maintaining organic-looking results. They compare this tool to another common industry tool, Mockaroo, to determine the pros and cons of Synner.

This article gives a real life compare and contrasts of two industry standard tools that I intend to compete with essentially. Mockaroo uses the complicated Ruby programming language to do various context aware evaluations and Synner is entirely GUI driven. I feel a DSL would prove easier to leverage and I plan on using their arguments to prove that.


The authors explain that DSL usability is not thoroughly tested in recently published articles and DSL publications. They test the usability of popular DSL’s published in recent years and also suggest a standard metric of DSL usability for future DSL publications.

This paper demonstrates what makes a DSL user friendly which is a lofty goal of my research. I will use this as a benchmark for the usability of my DSL.

This article, while rather old, talks about how well people retain information from a command line interface and a menu style interface (most likely a GUI). The article explains an experiment performed to determine how well an average person retains information on how to manipulate a file system with and CLI and a GUI and a mix of the two.

While this is very old information, it is still valuable to know how well the average person takes to a CLI. The target audience of my DSL is not a fluent developer but a regular computer user and tester. With the average user more acclimated to daily computer user, a CLI may be more easily adapted to but I believe most users would be extremely comfortable with a GUI (hence the current popularity of GUI’s but I want to steer away from that for reasons I will go into in my research).


This article covers how DSL’s are not exactly new in computer science but have been on the rise of late. The author explains that software engineers and other computing enthusiasts should trust on the application of DLS’s in software engineering and scientific computing because it is a proven method.

This author argues the importance of DSL’s in modern scientific computing. DSL’s scare some people away and this paper argues the point of embracing them which is beneficial to my cause.

Iglesias, P. (1999). Notes About lex and yacc
The author introduces the primary tool for the creation of DSL’s for the past two decades. This is thorough documentation on the tool at hand and explains the need for DSL’s for domain specific problems.

This source will be used as a guide for implementation and for defending the need for more use of DSL’s in the industry.


This IEEE paper describes a DSL that is used for software deployments on a Cloud infrastructure. In their paper they describe the expenses of creating a DSL of this magnitude which is a point I make in my research to elaborate on as well. DSL’s are expensive to engineer because of how much time and effort goes into them.

This DSL may not exactly have a similar purpose in mind as my DSL will have but I find the topic of a DSL to deploy software interesting and they make several great points on why creating a DSL can be so expensive and is not always the best choice for the job in an industry environment which is where this DSL would be utilized in more heavily, similar to mine.


The authors of this article analyze the need for automated data generation for context aware data driven applications. Automated test data is a need to increase testers coverage
and quality thus they need access to large quantities of data. Context aware data is difficult to generate and there is need to make this data generation to be automated.

This article proves the need for the context aware data generating tools that can be used by testers and software engineers.


The authors propose a solution for creating context aware datasets from UML models that are designed for context aware data driven applications. UML models are excellent for typesetting of data models and are a natural fit for data generation.

This paper shows a use for a standard software engineering tool (UML) to create data sets. This would be a good avenue for success with my DSL being able to ingest certain UML languages and output context aware data.

Miller, J. Han, J. Hybinette, M. (2010). Using Domain Specific Language for modeling and simulation: ScalaTion as a case study. *IEEE*

The authors discuss the use of GPPL to create DSL’s which is commonplace now. Their target for his case study was Scala (a popular JVM language adopted by an early version of Twitter) and a subset of Scala called ScalaTion when creating simulation programs.

This paper gives me the consideration of basing my DSL of a current popular GPPL like Kotlin or Python or going straight to Lex and Yacc to create it from scratch. I like the idea of creating a subset DSL for Kotlin so I may chase that.

The authors iterate how important context aware data is important to artificial intelligence testing such as recommender systems similar to what companies like Amazon and Netflix are using. Organic data is extremely costly in resources to scrape and comes at the invasion of privacy of their customers. With context aware data generation, testing can be done to replace organic data more accurately than bootstrapping small samples of organic data.

I will use this to prove my argument for the need of context aware data generation but also highlight the complications of this. Abstracting over it with a user-friendly DSL can black box the complicated parts of context aware data generation and help testers and software engineers have access to these complex data sets.


This paper is about building tools for model driven development, (the verbose title tells leaves no secrets) specifically Microsoft DSL tools versus the Eclipse plugins that are designed for creating DSL’s.

While I have my heart set on using MPS for my language, it is nice to explore what else is available. Unfortunately, this is a rather old article so I’m not sure how helpful it will be but learning from the past can always help prevent mistakes in the future.
Quyet, T. & Tran D. & Duc, M. & Ha, N. (2019). Generating Test Data for Blackbox Testing from UML-Based Web Engineering Content and Presentation Models

The authors similarly speak of the massive data consumption in software testing. The data is obviously hard to come by. Their solution is to use a UML based web engine that is from an eclipse plugin that generates data from a model.

This source reinforces the need for test data but this generation tool is more beneficial to technically inclined members of the industry which business people and some software testers are not. This reinforces my point of DSL’s being user friendly and accessible by all.


This article talks of a DSL called PARADIGM that is used for testing UI patterns in other softwares. PARADIGM can work like a linter and intake code and determine if it the UI is following the PBGT patterns correctly.

This is great chance to learn by example for software testing. This is more of a linter but it’s a good example of how to do a proper DSL.


This paper explains how context aware data is expensive to produce and complicated to validate but required for many modern web services. Their solution uses a UML input
file to create a dataset for context aware web services to the sake of software engineering and software testing,

This paper further proves that there exists a need for context aware data generation and a lack of a proper DSL to fully automate the process. Also, this is another example of UML ingest for data production.

Sthamer, Harmen & Wegener, Joachim & Baresel, Andre (2002). DaimlerChrysler AG, Research and Technology, Alt-Moabit 96a, D-10559 Berlin, Germany

This article speaks of the importance of testing in embedded systems. Embedded systems are more and more common in today’s world. These devices are typically data driven.

Feeding these devices with data can be tricky so creating predictable datasets for these would be extremely helpful. Data generation on a mass scale helps improve the testing of these devices. Cite this paper as:


This paper discusses a DSL created for to be a conceptual modeling tool. This MM-DSL as they call it is a useful modeling tool for defining a system and its associated algorithms.

A major portion of this DSL is modeling the data that it is going to generate so having some conceptual background around the DSL would be nice. This is an excellent learn by example source that I can take inspiration from for my DSL.

Mr. Volter explains the Turing complete process of creating a DSL to developers and architects. He elaborates on making them business user friendly as well. Focusing on a Java based language for MPS is helpful as well because I’d like to use Kotlin and Java in my language.

As stated previously I wanted to use Kotlin or Java as the DSL language. This Mr. Voelter or Volter is an excellent source to follow in the creation of DSL because he has a lot of experience in the field.


Mr. Volter explains in his updated post the use of MPS and other tools when creating DSL’s. DSL’s are the apex of modular design and he speaks of the importance of doing things the write way to prevent the language from devolving into a general-purpose language or a bastardized version of LISP.

MPS is my tool of choice and his use details of use will help in my creation. He also specifically talks about making DSL’s for business users which is the idea around the DSL that I’m creating.

This article explains the issues of data driven automated testing. Automated testing tools are notoriously tricky to integrate with and have their pitfalls.

Using this information, I can avoid the typical pitfalls and a data generation tool.


This article, while very old, helps fortify the need for automated testing and how it can be applied to numerous tasks in software engineering. Automation is a key point in software engineering, and it is important to automate task wherever possible.

Automation of a DSL is significantly easier and a GUI. Using automation, the speed and precision of testing can be increased which is the goal of the research.