Enhancing Synthetic Test Data Generation with Language Models Using a More Expressive Domain-Specific Language*

Chao Tan[1,2][0000−0002−0727−0092], Razieh Behjati[2], and Erik Arisholm[2]

[1] University of Oslo
[2] Testify AS, Oslo, Norway

Abstract. Generating production-like test data that complies with privacy regulations, such as the General Data Protection Regulation (GDPR), is a significant challenge in testing data-intensive software systems. Our previous research proposed an approach for generating such test data using a language model that captures the statistical properties of production data. While effective, the limited information capacity of the domain-specific language that transforms production data into model training corpus restricted the richness of the generated data. In this paper, we present a novel approach to design high information capacity and more expressive domain-specific language, which improves the ability to leverage deep-learning technology to generate rich, production-like test data. Our experiment results show that with higher information capacity and constraints complexity, the new language performs better regarding generated data quality, with an affordable increase on computational cost.

Keywords: synthetic data generation, domain specific language, deep learning, language modelling

1 Introduction

Production-like test data are essential in higher-level testing for complex software systems. As the privacy protection regulations like GDPR prohibit the use of production data in most test scenarios, the demand for synthetic and production-like test data arises high for scenarios such as integration testing of data-intensive distributed systems. Specifically, the synthetic test data need to be statistically representative of the production data and conform with the business constraints of the domain and application.

To address the need for synthetic production-like test data, we previously proposed a language modelling approach in our research [24]. Our approach involves building a language model that captures the statistical characteristics of production data and using it to generate synthetic test data. We also proposed an evaluation framework to measure the quality of the generated data in terms of

* The authors acknowledge the collaboration and support from the the Norwegian Population Registry, and the financial support from the Research Council of Norway.

statistical representativeness and business constraint conformance. We applied our solution to our case study, the Norwegian Population Registry (NPR), for integration testing with other public and private organizations that exchange a large amount of population data. We experimented with Char-RNN algorithm and built language model that generates highly representative data that conforms to the NPR domain's buisiness constraints. Our research demonstrates the effectiveness of language modelling in generating synthetic production-like test data.

While our previous research shows the potential of a deep-learning-based approach for generating rich and high-quality test data suitable for high-level testing of large-scale and complex systems, the critical step in this solution is to construct training corpus from production data. Many modern software systems store and exchange data in structured document formats such as XML or JSON. While such data can potentially be used directly for language model training, they are unsuitable for several reasons. Firstly, for privacy protection reasons, raw data containing sensitive information are unsuitable for exposure to any environment other than production and are usually restricted from utilising for model training. Secondly, not all information in the raw data is equally important for testing purposes, with some free text data or information, for example, that does not affect system behavior being less important. Therefore, it is necessary and essential to prepare a training corpus that is a collection of text strings constructed from the most important and not privacy-sensitive information from production data.

To express the selected information in the production data of a specific application and domain in one text string is essentially to create a domain-specific language (DSL). We designed a fixed-length domain-specific language for the NPR domain named *Steve132* in our previous model experiments. While this approach is effective, there are two significant shortcomings. Firstly, the information capacity of the language is limited due to the fixed number of information fields. One possible way to overcome the limit is to include more information fields. However, when the data in a domain has many important information fields, including all of them results in sequences that are impractically long. Secondly, in a domain where the presence of the information field is flexible, and absence is common, a large part of the sequences will be filled with filler characters, making d the sequence less efficient in capturing information. As longer sequences result in a larger training corpus, longer training time and data generation time, DSLs that require filler characters are less efficient regarding the computational cost.

This paper proposes a novel approach for designing DSL. Instead of relying on the position of information fields within the text string, it uses structural tokens and grammar to indicate information fields. We apply this new design approach to the NPR domain and design a new DSL, *Steveflex*. We experiment with this language by training a Char-RNN language model with a *Steveflex* corpus and compare the generated data quality with the *Steve132*. The results show that with higher information capacity and constraints complexity, *Steveflex*

performs better than *Steve132* regarding generated data quality, with affordable increase on computational cost.

The main contribution of this paper is the proposal of an effective approach for designing DSL with high information capacity and expressiveness, which enhances the ability to utilize deep-learning techniques for test data generation. With this approach, DSL can incorporate all relevant information fields into a single string without resorting to filler characters for missing data. Additionally, the current and historical values of the information fields can be accommodated, which further enhance the expressiveness of the DSL. As a result, the new DSL achieves a higher level of expressiveness that includes a historical dimension while also being more computationally efficient.

The remainder of the paper is organized as follows. We introduce our case study in Section 2 before we present the two DSLs and discuss their expressiveness and information capacity in Section 3. Section 4 presents our language model evaluation framework, and Section 5 presents our experiments, results and comparison between the two languages. We discuss related work in Section 6 and conclude the paper in Section 7.

2 Case Study

The Norwegian National Population Registry (NPR) collects, stores and manages the personal data of all the residents of Norway and distributes electronic personal information to more than 2000 organisations (which are referred to as data consumers) so that they can provide service to society. The software system in NPR is undergoing a modernisation process, and so are the software systems of many of its data consumers. An effective setup for cross-organisational integration testing, which is critical for a successful transition, requires a large amount of production-like data that can stimulate realistic test scenarios.

Under the restriction of GDPR, production data are strictly prohibited in such testing. Therefore, our collaboration with the NPR aims to provide synthetic, dynamic and production-like data to support the cross-organisational integration testing between NPR and its data consumers. To apply our proposed solution and train a language model to generate a synthetic population representative of the Norwegian population synthetically, the first and essential step is designing a domain-specific language to form a training corpus from the NPR data.

2.1 Abstract Data Model

The software systems of the NPR and its data consumer organisations are event-based systems. In these systems, each record of personal data, and any event that happens to a person (birth, marriage or relocation) that gets registered into NPR, are stored as XML documents.

We model the NPR data as an abstract event-based model in Figure 1. An event-based system can be seen as a collection of *StatefulEntities*, whose states can be altered by events. Each stateful entity (or entity for short) consists of

a collection of *StateDescriptors*. The *StateDecriptors* are typed, and each type of *StateDescriptor* describes one group of information about the entity. At any time, an entity contains only one currently applicable instance of one type of *StateDescriptor*. An event consists of one or more *StateDescriptors*. An event happening to an entity adds its *StateDescriptor* into the entity and the added ones become currently applicable; if the entity has that type of *StateDescriptor* from before, the existing instances are set to historical.

In the NPR domain, each personal data record is an *StatefuEntity*; each aspect of the personal data, for example, birth information, civil status, residence address and many others, is a type of *StateDescriptor*. The collection of all the personal data forms the population.

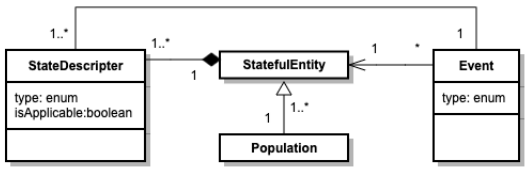

Fig. 1. Abstract data model for event-based systems. Note that the *StateDescriper* has a boolean attribute *isApplicable*, which allows the data model to accommodate historical information about an entity: *isApplicable: True* means that this *StateDescriptor* instance is currently applicable, and *isApplicable: False* indicates historical.

3 Domain Specific Language Design and Comparison

In event-based systems, in order to build a synthetic population that is dynamic and statistically representative, it is sufficient to generate statistically representative events. Statistically representative events propagate through the systems and maintain a statistically representative state of the population.

3.1 Domain Specific Formal Language - *Steve132*

Construction of *Steve132* strings follow these steps: (1) select the information fields to include from the production data, (2) encode each information field's possible values using fixed-length character-level tokens, (3) concatenate the encoded information fields in a predetermined order to form a fixed-length sequence of characters, and (4) if any included information field is absent in a specific record of the production data, use filler characters of the same length to denote the absence and take up the corresponding position in the text string.

Figure 2 illustrates the structure of a *Steve132* sentence with an example, which consists of three parts: the event type, the current state of the person, and the event details. The string contains 132 characters, representing 48 information

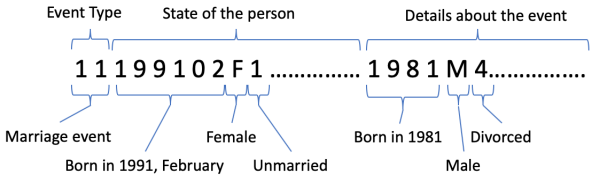

Fig. 2. *Steve132* example sentence, which describes a marriage event of a unmarried female born in Febrary 1991. The spouse is a divorced male born in 1981.

fields, with 1 for event type, 18 for the state of the person, and 29 for details of the event. The first two characters denote the event type (in this case, a marriage event of type *11*), followed by the state of the person and the details of the event, respectively. The state of the person contains information such as birth year, birth month, gender, and civil status, while the details of the event include information about the spouse and the location and date of the marriage event.

3.2 Domain Specifical language - *Steveflex*

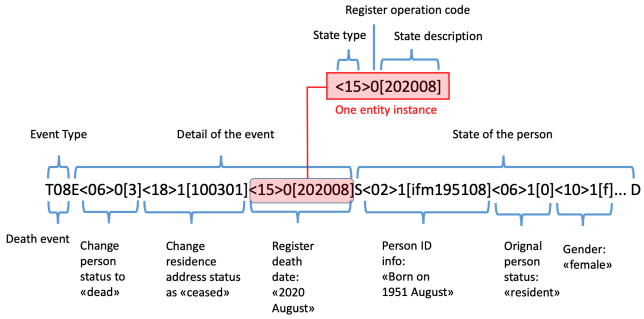

Fig. 3. *Steveflex* example sentence

The *Steveflex* language design is also based on the abstract data model. We design a set of structural tokens to denote the elements in the abstract data model, as listed in Table 1. The event type, the current state of the entity and the event details are denoted with token **T, S** and **E**. A state descriptor consists of the state type and the state description. Angle brackets <> encloses the type of the state and square brackets **[]** encloses the description of the state. Token **D** denotes the end of a sequence.

Table 1. Structural tokens

Token	Description
T	Start of event type
E	Start of event detail
S	Start of statefull entity status
D	End of sequence
<>	Encloses type of state
[]	Encloses description of state

Besides the structural tokens, we utilize number characters to denote types, specifically, event type and state type. And for construction of state description, we utilize a similar approach as for the *Steve132* language, i.e., (1) identify information fields in this state description to include, (2) for each information field, identify all its possible values and encode with fixed length character-level tokens and (3) concatenate these information fields together to form a character sequence. Further more, *Steveflex* uses lowercase characters and numbers for state description tokens for better readability for humans and to avoid confusion with the capital structural tokens.

Figure 3 shows an example of a *Steveflex* sequence. This sentence describes a death event of event type *08*. The event detail part shows that this event modifies three states of a person: alters *status* to *dead*, changes the *residence address* status to *ceased*, and registers a *date of death*, August 2020. The state of the person part contains the person's state before this event happens. This person is a female born in August 1951. In "date of death" state in the event detail part, we see that a pair of angle brackets enclose the state, which is of type *15*. The state description part is a sequence of six digits, denoting the year and the month, and is enclosed in rectangular brackets. Note that the length and content of state descriptions vary with state types, as each type of state contains different information fields.

A *Steveflex* sentence captures all the important information fields in a production data record. If any states are absent, they are left out and do not need any position filler in the sequence.

3.3 Historical dimension in expressiveness

The *Steveflex* language designates one character between the state type and the state description part, i.e. the position between the right angle bracket '>' and the left rectangular bracket '[', for extra information for each state descriptor, i.e. meta data. The grammar for meta data in the NPR domain, as shown in Table 2: The meta data field in a state in the event detail represents the register operation code in the NPR, as shown in Figure 3. It takes value from $0, 1, 2$ and 3, representing four types of register operations: register a new state, alter the current state, cancel the current state or alter the historical state. The meta data field in the person state part takes value from 0 and 1, representing whether the state is historical or currently applicable. Furthermore, if one state type has

multiple instances present in the person state, only one can have meta data 1, meaning currently applicable, and all the others must have meta data 0, i.e. historical.

Introducing the meta data highly increases the expressiveness of the *Steveflex* language. Comparing to the *Steve132* language, which cannot describe a person's historical state or different type of action of an event, the *Steveflex* language provides a new dimension for expressing information in the NPR domain.

Table 2. Meta data grammar in *Steveflex*

meta data in state in the event detail		meta data in state of the person state description	
0	Register new state	0	historical
1	Alter current state	1	currently applicable
2	Cancel state	If there are multiple instances of one type state, only one of them	
3	Alter historical state	can have meta data 1, and all the others must have meta data 0	

3.4 Higher information capacity

Due to its limited length, the *Steve132* language excludes many important information fields from the production data, such as contact information for death residence, identification document from another country, shared residence for children with divorced parents, parental responsibility and use of the Sami language, and more. In contrast, the *Steveflex* language captures all types of states and includes as many information fields as needed. *Steveflex* defines 30 types of states; if all the defined states and information fields present, a *Steveflex* sequence contains 141 information fields, which is almost 3 times as many as that of the *Steve132*.

Not requiring filler characters makes the *Steveflex* language more efficient in conveying information. To illustrate, in our *Steve132* training corpus, the sequences have an average of 68.03 filler characters for each, indicating that only about 48.5% of the sequence length conveys information. In contrast, 100% of the *Steveflex* sequence length conveys information.

4 Evaluation Framework

Language models are evaluated based on their performance in downstream tasks. For our application of generating synthetic test data, we evaluate the quality of the generated data based on three aspects: syntactic validity, statistical representativeness, and semantic validity. Only syntactically valid data translates to valid value for testing, while representative and semantically valid data creates realistic testing scenarios.

Syntactic validity The syntactic validity rate represents the percentage of generated data that comply with the syntax and grammar of the DSL.

Representativeness Representativeness is evaluated by measuring the similarity between the distributions (and joint distributions) of information fields in

the generated and training data, but only on syntactically valid data. A person's data record in the NPR has many fields, including name, gender, birth date, address, marital status, family relation, and many others. The events in NPR also have many fields, and event type is one of them, and many event specific information fields, such as spouse information for marriage event, and original country for immigration event. Similarity of distributions and joint distributions of these information fields indicates how well the generated person and events represents the real data in the NPR

We measure distribution similarity with the symmetric and bounded Jensen-Shannon divergence (JSD) metric [5], which is zero when two probability distributions are identical and grows to a maximum value of 1.0 as the distributions diverge.

Semantic validity The semantic validity rate represents the percentage of generated data that conforms to the application domain's constraints. These constraints can be specified in any logic language, and we use Python in Example 1 to illustrate the format of the constraints: given a logical expression as the condition, the semantic validity equals the evaluation result of a logical expression. A constraint with such a definition specifies a relationship between two or more data fields.
Example 1 (Constraint definition).

```
if condition:
  semantic_validity = expression
```

The following is an example of a constraint in the NPR domain:

```
if EventType == "Marriage":
  semantic_validity = (person.age >= 18)
    and (person.civilStatus not in ["pertnership", "married"
   ])
    and (person.currentSpouseOrPartnerInfo = null)
    and (event.spouseAge >= 18)
```

This constraint specifies that if an event is of type marriage, the person it happens to should be at least 18 years, has a civil status that is neither married nor partnership and has no registered spouse or partner in the national registry. Additionally, the event must state that the new spouse is also at least 18 years old.

The validity rate for each constraint is the ratio of the number of generated strings that are valid for this constraint, to the total number of generated strings. Given a set of constraints, the aggregated or total validity rate is the ratio of the number of total data records for which all of the constraints are valid to the total number of data records.

Due to the complexity of real-world domains and applications, exhaustively checking every constraint in a domain is impossible. High conformance to a representative subset of the constraints can indicate that the model has learned the business rules of the domain well. Therefore, a high semantic validity rate for a subset of the constraints implies that the majority of the other constraints will also hold for the majority of the generated data.

5 Experiment, Result and Comparison

We experiment with the Char-RNN algorithm to train the language model for the *Steveflex* language. We obtain more than 108k of data records from the NPR domain and transform the records into *Steveflex* sequences to form our training corpus. For privacy protection reasons, we opt anonymized data instead of raw production data. The anonymization process in NPR is standard, automated and managed by a dedicated data processing team. The algorithm used for anonymization, to a great extent, preserves the statistical properties of the production data. The amount of anonymized data available equals the amount of real population data. However, for our experiments, we collected event documents recorded for 100 days to keep the data size and computational cost manageable. Note that although the anonymized data is available for internal testing in NPR, the anonymization algorithm is not sufficiently privacy-preserving to make the data eligible for cross-organizational integration testing. However, the synthetic data from the language model does not suffer from this restriction. This is because no one-to-one correspondence exists between the generated event or person documents and the event and person documents in production.

5.1 Result and Comparison

Syntactic validity We evaluated 850k generated sequences from our trained Char-RNN language model. The syntactic validity rate of the generated data is 99.55%, demonstrating that the model learns the *Steveflex* syntax exceptionally well.

The definition of syntactic validity for *Steve132* is the same, and the generated data has a syntactic validity rate of 96.06%. The model learns the *Steve132* language well, but the *Steveflex* language model outperforms it for this criteria.

Representativeness Table 3 summarizes the results for both *Steveflex* and *Steve132*, including single information field JSDs in the first part of the table and joint JSDs of multiple information fields in the second part.

The first item in the table, *seqLength*, indicates the similarity of the sequence length distributions for training and generated data. Although sequence length is not an information field in the *Steveflex* or *Steve132* sequence, it is an important indication of the similarity between generated data and training data.

For *Steve132*, the JSD for sequence length is 0, indicating that the distribution of sequence length is identical between the generated data and training data. In fact, all the generated data from the *Steve132* model have a fixed length of 132 characters, matching the training data. In contrast, the length of the *Steveflex* sequences varies significantly. The sequence length in the training corpus ranges from 82 to 1049 with an average of 179.8. For *Steveflex*, the JSD for sequence length is 0.0073, indicating that the sequence length distributions of the generated data and training data are highly similar. The JSD values for the other single information fields for both *Steveflex* and *Steve132* are relatively small, indicating that the distributions of the generated data are similar to those of the training data.

Table 3. *JSD* and joint *JSD*

distributions	*Steveflex*	*Steve132*
seqLength	0.0093	0
eventType	0.0073	0.0032
person_birth_year	0.0061	0.0102
person_birth_month	0.0036	0.0072
person_civil_status	0.0011	0.0028
person_residence_municipality	0.0127	0.0267
person_death_yearn	0.0015	0.0001
person_death_month	0.0109	0.0002
person_immigrant_from_country	0.0088	0.0066
person_emigrate_to_country	0.0013	0.0018
partner_birth_year	0.0000	0.0021
joint_eventType_municipality	0.0267	0.0404
joint_eventType_birthYear_civilStatus	0.0310	0.0383
joint_birthYear_gender_fromCountry	0.0266	0.0285
joint_eventType_stateChange	0.0083	-
joint_personStatus_state	0.0173	-

Note that in the second part of the table, two joint distributions are missing for *Steve132*: *joint_ eventType_ stateChange* and *joint_ personStatus_ state*. The *joint_ eventType_ stateChange* represents the joint distribution of event types and the type of states they alter, i.e., what type of events alters what type of information. and the *joint_ personStatus_ state* is the joint distribution of the person status, and the presence of type of state descriptors presence for the person. These distributions involve typed states, which do not exist in the *Steve132* language.

It is not practical to examine all the distributions and joint distributions of the synthetic data and training data. Overall, however, we can confidently conclude that both the *Steveflex* and *Steve132* models learn the statistical properties of the training data of its language, and can generate highly-representative data. Despite the higher complexity of *Steveflex*, the JSDs and joint JSDs are comparable to or even better than those of *Steve132*.

Semantic validity Table 4 provides a summary of the constraint checks for 8 event types in the generated *Steve132* sequences, including marriage, birth, and death events, etc. The constraints are expressed as Python functions(see Appendix A). To evaluate of the complexity of these constraints, we check the cyclomatic complexity, i.e., the number of decisions in the validation function for each event type, as listed in the table. On average, each event type has a cyclomatic complexity of 2.73. In the generated data, two types of events have 100% valid rate, five types have more than 99% valid rate, and one has 98.25%. In total, the generated data have a 97.12% valid rate.

Steveflex has more complex constraints than *Steve132*. These constraints apply to either a single state, between multiple states or to the whole event.

Table 4. Constraint conformity - Steveflex132

Event type	Cyclomatic complexity	Valid rate
Marriage	4	98.25%
Birth	3	99.94%
Death	2	100%
Relocation within municipality	3	99.95%
Relocation between municipality	3	99.89%
Immigration	2	99.43%
Emigration	3	99.64%
Change name	2	100%
Over all	2.73(average)	**97.12%**

Table 5. Constraint conformity - Steveflex

State or event type	Cyclomatic complexity	Valid rate
State ID number	2	99.99%
State civil status	11	99.99%
Event death	20	99.97%
Event Change in civil status	27	99.76%
Overall	15(average)	**99.74%**

Table 5 shows the constraint-checking results for two types of states and two types of events, which are shown in Appendix B.

Steveflex defines 10 types of civil status, and in the NPR domain, there is one rule for each civil status type; besides, the state must have valid meta data indicating it is currently applicable. Hence, the constraint for state "civil status" has 11 checks. The constraints for events are more complex than those for the states because they usually involve checking multiple states in both the event details and person state descriptions. For example, the constraint for event type "death" involves the checks for state *Death, Residence Address, and Status*, and their operation codes and metadata. This constraint has 20 checks. The constraint for "change of civil status" event has 27 checks. On average, these four constraints have a cyclomatic complexity of 15, which shows that *Steveflex* constraints are much more complex than *Steve132* constraints. The valid rates for all these constraints are more than 99%, and the overall valid rate is 99.74%. Despite the higher complexity of the constraints, the *Steveflex* generated data overperforms that of the *Steve132* regarding semantic validity.

The above three subsections demonstrate that, despite its higher information capacity and greater complexity in terms of language syntax and domain constraints, the data generated using *Steveflex* is of comparable or superior quality to that of *Steve132*. These results showcase the impressive learning ability of the Char-RNN algorithm for complex DSLs and suggest its potential application for data generation with language modeling in other domains.

5.2 Experimental setup

Our training utilized a Tesla T4-4C virtual GPU with 4GB of available GPU memory for both *Steve132* and *Steveflex* experiments. We employed a PyTorch Char-RNN implementation [1] and fine-tuned various hyperparameters to identify the best model based on model loss and generated data quality.

Table 6. Experiment setup comparison

	Steveflex	*Steve132*
Network size	Two layers network, each with 400 GRU units	Two layers network, each with 100 GRU units
Training data size	113M	23M
Training epochs	13	28
Training time	26 hours	4 hours
Generation time for 100k sequences	100 minutes	less than 5 minutes

Our training setup is summarized in Table 6. The best performing *Steveflex* model has two layers, with 400 GRU units in each layer, and employs the Adam optimizer [10]. The network has over 2 million parameters and was trained for 27 epochs. The lowest validation loss is achieved after 26 hours of training, at the end of the 13th epoch, and we use the model at the end of this epoch for data generation. Using this model, it takes approximately 100 minutes to generate $100k$ sequences.

Comparing to the *Steve132* setup, the network size for the *Steveflex* model is four times larger, and the training time is more than six times longer. This is expected due to the larger model size and training corpus size. As a consequence, the data generation time is also longer for the same number of sequences.

The NPR domain expects population data statistics to change slowly, so retraining the model frequently is not necessary. The training corpus is composed of 100 days of production data, making quarterly retraining a reasonable option. A 26-hour training time for quarterly retraining is feasible for this application. On average, there are about 1,000 events per day in the NPR domain, so generating 100,000 sequences is enough to simulate a production-like data flow for a quarter. A 100-minute data generation time every quarter is manageable s well. Overall, the increase on computational cost for adopting *Steveflex* is entirely affordable.

6 Related Work

Generating test data is a well-researched topic, with various techniques proposed, such as combinatorial [13], [20], [19], metaheuristic search [15], [8], [6], model-based [23], [26], [2], fuzzing [12], [16], and machine learning algorithms [7], [9], [27], [4]. While many of these approaches focus on increasing test coverage at lower testing levels, high-level testing, such as integration and end-to-end testing, is essential for quality assurance of large-scale complex software systems. However,

simulating realistic test scenarios at this level is challenging without access to production-like test data. This is where our research comes in, as it addresses this challenge.

Along with the flourishing of deep learning techniques, language models, especially large language models (LLMs), have progressed rapidly in recent years [3], [11], [18], [21], [25]. It is also reported that LLMs are used for bug fixing [17], [22]. However, the computational cost of training such models is huge. Hoffman et al. [25] proposed a relation between scaling up model size and increasing the number of training tokens, However, this relation cannot be applied to our approach of training language models for DSL for data generation purpose. Our language models, *Steve132* and *Steveflex-NPR*, differ from LLMs in multiple ways, including the learning objective, complexity, and evaluation methods. Additionally, our models are trained on epochs, and we do not have an unlimited corpus like many of the LLM training, making it challenging to scale up the number of training tokens as language size scales up. Nonetheless, the proposed relation offers a potential direction for optimizing model training.

7 Conclusion

In this paper, we present a novel approach for designing domain-specific languages for synthetic data generation using language models. We apply this design approach to the Norwegian Population Registry domain and design a more expressive DSL with higher information capacity and constraint complexity, the *Steveflex*. Through the training of a language model for *Steveflex* and an evaluation of the generated data, we show that the new language outperforms the previous DSL, *Steve132* in terms of data quality. Moreover, we demonstrate that the increase in computational cost associated with using *Steveflex* is affordable, making it a feasible choice for synthetic data generation in the NPR domain. Our approach can also be applied to other domains to design DSLs for synthetic data generation, which can result in high-quality synthetic data with a lower cost and effort compared to manual data synthesis. Overall, our work contributes to the advancement of synthetic data generation in various domains, with potential applications in privacy-preserving data collection, data analysis and machine learning.

References

1. char-rnn.pytorch. https://github.com/spro/char-rnn.pytorch, accessed: 2019-08
2. Ali, S., Iqbal, M.Z., Arcuri, A., Briand, L.C.: Generating test data from ocl constraints with search techniques. IEEE Transactions on Software Engineering **39**(10), 1376–1402 (2013)
3. Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al.: Language models are few-shot learners. Advances in neural information processing systems **33**, 1877–1901 (2020)
4. Čegiň, J., Rástočnỳ, K.: Test data generation for mc/dc criterion using reinforcement learning. In: 2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 354–357. IEEE (2020)

5. Fuglede, B., Topsoe, F.: Jensen-shannon divergence and hilbert space embedding. In: International Symposium onInformation Theory, 2004. ISIT 2004. Proceedings. p. 31. IEEE (2004). https://doi.org/10.1109/ISIT.2004.1365067
6. Gois, N., Porfírio, P., Coelho, A.: A multi-objective metaheuristic approach to search-based stress testing. In: 2017 IEEE International Conference on Computer and Information Technology (CIT). pp. 55–62. IEEE (2017)
7. Ji, S., Chen, Q., Zhang, P.: Neural network based test case generation for data-flow oriented testing. In: 2019 IEEE International Conference On Artificial Intelligence Testing (AITest). pp. 35–36. IEEE (2019)
8. Khari, M., Kumar, M., et al.: Analysis of software security testing using meta-heuristic search technique. In: 2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom). pp. 2147–2152. IEEE (2016)
9. Kim, J., Kwon, M., Yoo, S.: Generating test input with deep reinforcement learning. In: 2018 IEEE/ACM 11th International Workshop on Search-Based Software Testing (SBST). pp. 51–58. IEEE (2018)
10. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 arXiv:1412.6980 (dec 2014)
11. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.t., Rocktäschel, T., et al.: Retrieval-augmented generation for knowledge-intensive nlp tasks. Advances in Neural Information Processing Systems **33**, 9459–9474 (2020)
12. Li, J., Zhao, B., Zhang, C.: Fuzzing: a survey. Cybersecurity **1**(1), 1–13 (2018)
13. Li, N., Lei, Y., Khan, H.R., Liu, J., Guo, Y.: Applying combinatorial test data generation to big data applications. In: 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 637–647. IEEE (2016)
14. MacKay, D.J.: Information theory, inference and learning algorithms. Cambridge university press (2003)
15. McMinn, P.: Search-based software testing: Past, present and future. In: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops. pp. 153–163. IEEE (2011)
16. Padhye, R., Lemieux, C., Sen, K., Papadakis, M., Le Traon, Y.: Semantic fuzzing with zest. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 329–340 (2019)
17. Prenner, J.A., Babii, H., Robbes, R.: Can openai's codex fix bugs? an evaluation on quixbugs. In: Proceedings of the Third International Workshop on Automated Program Repair. pp. 69–75 (2022)
18. Rae, J.W., Borgeaud, S., Cai, T., Millican, K., Hoffmann, J., Song, F., Aslanides, J., Henderson, S., Ring, R., Young, S., et al.: Scaling language models: Methods, analysis & insights from training gopher. arXiv preprint arXiv:2112.11446 (2021)
19. Salecker, E., Glesner, S.: Combinatorial interaction testing for test selection in grammar-based testing. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. pp. 610–619. IEEE (2012)
20. Simos, D.E., Kuhn, R., Voyiatzis, A.G., Kacker, R.: Combinatorial methods in security testing. IEEE Computer **49**(10), 80–83 (2016)
21. Smith, S., Patwary, M., Norick, B., LeGresley, P., Rajbhandari, S., Casper, J., Liu, Z., Prabhumoye, S., Zerveas, G., Korthikanti, V., et al.: Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. arXiv preprint arXiv:2201.11990 (2022)
22. Sobania, D., Briesch, M., Hanna, C., Petke, J.: An analysis of the automatic bug fixing performance of chatgpt. arXiv preprint arXiv:2301.08653 (2023)

23. Soltana, G., Sabetzadeh, M., Briand, L.C.: Synthetic data generation for statistical testing. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 872–882. IEEE (2017)
24. Tan, C., Behjati, R., Arisholm, E.: Application of deep learning models to generate representative and scalable synthetic test data for the norwegian population registry. Submitted to Journal of Systems and Software (2020)
25. Thoppilan, R., De Freitas, D., Hall, J., Shazeer, N., Kulshreshtha, A., Cheng, H.T., Jin, A., Bos, T., Baker, L., Du, Y., et al.: Lamda: Language models for dialog applications. arXiv preprint arXiv:2201.08239 (2022)
26. Yano, T., Martins, E., de Sousa, F.L.: A model-based approach for robustness test generation. In: 2011 Fifth Latin-American Symposium on Dependable Computing Workshops. pp. 33–34. IEEE (2011)
27. Zhou, X., Zhao, R., You, F.: Efsm-based test data generation with multi-population genetic algorithm. In: 2014 IEEE 5th International Conference on Software Engineering and Service Science. pp. 925–928. IEEE (2014)

A Steve132 constraints

```
""" check state validity """
""" State ID number """
def isStateValid_IDnumber(state):
  sementic_validity = (state.
      hasOneApplicable == True)
    and (state.year in range(1900,2023))
  return sementic_validity

""" State Civil Status """
def isStateValid_CivilStatus(state):
  sementic_validity = True
    if state.hasOneApplicable == False
      sementic_validity = False
      return sementic_validity
  match state.civilStatus:
    case "single":
      sementic_validity = not state.
     hasSpouseInfo
    case "married":
      sementic_validity = state.
     hasSpouseInfo
    case "widowed":
      sementic_validity = not state.
     hasSpouseInfo
    case "divorced":
      sementic_validity = not state.
     hasSpouseInfo
    case "separated":
      sementic_validity = not state.
     hasSpouseInfo
    case "registeredPartnership":
      sementic_validity = state.
     hasSpouseInfo
    case "separatedPartner":
      sementic_validity = not state.
     hasSpouseInfo
    case "divorcedPartner":
      sementic_validity = not state.
     hasSpouseInfo
    case "survivingPartner":
      sementic_validity = not state.
     hasSpouseInfo
    case "unknown":
      sementic_validity = not state.
     hasSpouseInfo
  return sementic_validity


""" check event validity """
""" Event Death """
def isEventValid_death(event, personState
    ):
  if event.hasEntityDeath == False:
    return False
  match event.entityDeath.operationCode:
    case "RegisterNew":
      if personState.hasEntityDeath ==
     True:
        if personState.entityDeath.
     hasOneApplicable == True:
          return False
    case ["AlterCurrentState" | "
     CancelState" ]:
      if personState.hasEntityDeath ==
     False:
        return False
      if personState.entityDeath.
     hasOneApplicable == False:
        return False
    case "AlterHistoricalState":
      if personState.hasEntityDeath ==
     False:
        return False
      if personState.entityDeath.
     hasOneNonApplicable == False:
        return False
  if personState.
     hasEntityResidencyAddress == True
    if event.hasEntityResidencyAddress ==
      False
      return False
    if event.entityResidencyAddress.
     hasOneApplicable == False
      return False
  if personState.hasEntityStatus == False
    return True
  if person.entityStatus.hasOneApplicable
     == False
    return True
  if event.hasEntityStatus == True:
    if event.entityStatus.operationCode
     == "CancelState" and
       event.entityStatus.stateus == "
     Dead":
      return True
  if personState.hasEntityStatus == False
     :
    return False
  if not (personState.entityStatus.
     hasOneApplicable and
    personState.entityStatus.status == "
     Dead"):
    return False
  if (event.entityDeath.hasOneApplicable
     and
    event.entityDeath.hasDate):
    return True
  return False


""" Event Change Civil Status """
def isEventValid_ChangeCivilStatus(event,
     personState):
  sementic_validity = True
  if event.hasEntityCivilStatus == False
    return False
  match event.entityCivilStatus.
     civilStatus:
```

```
    case ["married" | "
     registeredPartnership"]:
      if event.entityCivilStatus.
     hasSpouseInfo == False:
        return False
    case ["unknown"]:
      pass
    case _:
      if event.entityCivilStatus.
     hasSpouseInfo == True
        return False
  match event.entityCivilStatus.
     operationCode:
    case "RegisterNew":
      if personState.hasEntityCivilStatus
     == False:
        return True
      if personState.entityCivilStatus.
     hasOneApplicable == False:
        return True
      match event.entityCivilStatus.
     civilStatus:
        case "unknown":
          if personState.
     hasEntityCivilStatus == True
            return False
        case ["married" | "
     registeredPartnership"]:
          if personState.
     entityCivilStatus.civilStatus in ["
     married", "registeredPartnership"]:
            return False
        case _:
          if personState.
     entityCivilStatus.civilStatus in ["
     married", "registeredPartnership"]
          return True
      return False
    case "AlterCurrentState":
      if personState.hasEntityCivilStatus
     == False:
        return False
      if personState.entityCivilStatus.
     hasOneApplicable == False:
        return False
      if event.entityCivilStatus.
     civilStatus == personState.
     entityCivilStatus.civilStatus
        return False
      return True
    case "CancelState":
      if personState.hasEntityCivilStatus
     == False:
        return False
      if personState.entityCivilStatus.
     hasOneApplicable == False:
        return False
      if event.entityCivilStatus.
     civilStatus == personState.
     entityCivilStatus.civilStatus
        return True
      return False
    case "AlterHistoricalState":
      if personState.hasEntityCivilStatus
     == False:
        return False
      if personState.entityCivilStatus.
     hasNonApplicable == False:
        return False
      if event.entityCivilStatus.
     civilStatus == personState.
     entityCivilStatus.history.
     civilStatus
        return False
  return sementic_validity
```

B Steveflex constraints

```
""" check state validity """
""" State ID number """
def isStateValid_IDnumber(state):
  sementic_validity = (state.
     hasOneApplicable == True)
    and (state.year in range(1900,2023))
  return sementic_validity

""" State Civil Status """
def isStateValid_CivilStatus(state):
  sementic_validity = True
    if state.hasOneApplicable == False
      sementic_validity = False
      return sementic_validity
  match state.civilStatus:
    case "single":
      sementic_validity = not state.
     hasSpouseInfo
    case "married":
      sementic_validity = state.
     hasSpouseInfo
    case "widowed":
      sementic_validity = not state.
     hasSpouseInfo
    case "divorced":
      sementic_validity = not state.
     hasSpouseInfo
    case "separated":
      sementic_validity = not state.
     hasSpouseInfo
    case "registeredPartnership":
      sementic_validity = state.
     hasSpouseInfo
    case "separatedPartner":
      sementic_validity = not state.
     hasSpouseInfo
    case "divorcedPartner":
      sementic_validity = not state.
     hasSpouseInfo
    case "survivingPartner":
      sementic_validity = not state.
     hasSpouseInfo
    case "unknown":
      sementic_validity = not state.
     hasSpouseInfo
  return sementic_validity


""" check event validity """
""" Event Death """
def isEventValid_death(event, personState
    ):
  if event.hasEntityDeath == False:
    return False
  match event.entityDeath.operationCode:
    case "RegisterNew":
      if personState.hasEntityDeath ==
     True:
        if personState.entityDeath.
     hasOneApplicable == True:
          return False
    case ["AlterCurrentState" | "
     CancelState" ]:
      if personState.hasEntityDeath ==
     False:
        return False
      if personState.entityDeath.
     hasOneApplicable == False:
        return False
    case "AlterHistoricalState":
      if personState.hasEntityDeath ==
     False:
        return False
      if personState.entityDeath.
     hasOneNonApplicable == False:
        return False
  if personState.
     hasEntityResidencyAddress == True
    if event.hasEntityResidencyAddress ==
      False
      return False
    if event.entityResidencyAddress.
     hasOneApplicable == False
      return False
  if personState.hasEntityStatus == False
    return True
  if person.entityStatus.hasOneApplicable
     == False
    return True
  if event.hasEntityStatus == True:
    if event.entityStatus.operationCode
     == "CancelState" and
       event.entityStatus.stateus == "
     Dead":
      return True
  if personState.hasEntityStatus == False
    :
    return False
```

```
  if not (personState.entityStatus.
     hasOneApplicable and
     personState.entityStatus.status == "
     Dead"):
     return False
  if (event.entityDeath.hasOneApplicable
     and
     event.entityDeath.hasDate):
     return True
  return False


""" Event Change Civil Status """
def isEventValid_ChangeCivilStatus(event,
      personState):
  sementic_validity = True
  if event.hasEntityCivilStatus == False
     return False
  match event.entityCivilStatus.
     civilStatus:
     case ["married" | "
     registeredPartnership"]:
       if event.entityCivilStatus.
     hasSpouseInfo == False:
          return False
     case ["unknown"]:
       pass
     case _:
       if event.entityCivilStatus.
     hasSpouseInfo == True
          return False
  match event.entityCivilStatus.
     operationCode:
     case "RegisterNew":
       if personState.hasEntityCivilStatus
       == False:
          return True
       if personState.entityCivilStatus.
     hasOneApplicable == False:
          return True
       match event.entityCivilStatus.
     civilStatus:
          case "unknown":
            if personState.
     hasEntityCivilStatus == True
               return False
          case ["married" | "
     registeredPartnership"]:
            if personState.
     entityCivilStatus.civilStatus in ["
     married", "registeredPartnership"]:
               return False
          case _:
            if personState.
     entityCivilStatus.civilStatus in ["
     married", "registeredPartnership"]
               return True
       return False
     case "AlterCurrentState":
       if personState.hasEntityCivilStatus
       == False:
          return False
       if personState.entityCivilStatus.
     hasOneApplicable == False:
          return False
       if event.entityCivilStatus.
     civilStatus == personState.
     entityCivilStatus.civilStatus
          return False
       return True
     case "CancelState":
       if personState.hasEntityCivilStatus
       == False:
          return False
       if personState.entityCivilStatus.
     hasOneApplicable == False:
          return False
       if event.entityCivilStatus.
     civilStatus == personState.
     entityCivilStatus.civilStatus
          return True
       return False
     case "AlterHistoricalState":
       if personState.hasEntityCivilStatus
       == False:
          return False
       if personState.entityCivilStatus.
     hasNonApplicable == False:
          return False
       if event.entityCivilStatus.
     civilStatus == personState.
     entityCivilStatus.history.
     civilStatus
          return False
  return sementic_validity
```