Artifact Evaluation of Paper "DeepSTL - From English Requirements to Signal Temporal Logic"

Jie He
jie.he@tuwien.ac.at

ABSTRACT

This documentation is used for artifact evaluation of our accepted paper titled "DeepSTL - From English Requirements to Signal Temporal Logic". This material is a "Read Me" on how to use our database and codes for Section 4-6, and provides guidance on how to reproduce the results obtained in the paper including all information in the figures and tables. We also offer exhaustive descriptions of the design principles of the user interfaces for our tool, making it convenient for people to reuse our artifact to customize their own translation tasks. Therefore, this documentation is used to claim the "Artifacts Available" badge, and the "Artifacts Evaluated" badge preferably with Reusable level. Our artifact is based on Ubuntu 20.04 (amd64) operating system. Anyone with basic experience of using Linux Operating system and machine-learning based Python environment, will be able to evaluate our artifact.

ACM Reference Format:
Jie He. 2022. Artifact Evaluation of Paper "DeepSTL - From English Requirements to Signal Temporal Logic". In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3510003.3510171

1 INTRODUCTION

This short introduction will describe the organization of this documentation, and the file structure of our artifact, which is publicly archived on *figshare* website with the URL link provided in [1] and DOI number: 10.6084/m9.figshare.19091282.

Our artifact is named `Archive.zip`. After extracting this file, there are two folders: `/documents` and `/material`. There are six files in folder `/documents`, and they are `README.pdf`, `REQUIREMENTS.txt`, `STATUS.pdf`, `LICENSE`, `INSTALL.pdf` and `PAPER.pdf`. They cover all information related to the artifact and the associate paper. In folder `/material`, there are two folders. `/material/VM` incorporates the submitted Virtual Machine with all environment set up. Our artifact is independently stored in the directory of `/material/artifact`.

The rest of this documentation is organized as follows. Section 2 introduces information about experimental environment. Section 3 describes how to obtain empirical statistics in Section 4 of the main paper. Section 4 provides a detailed illustration on how to use programs for Section 5 and 6 in the main paper. In Appendix A, we briefly introduce the main file structures and functions of the source codes to facilitate reusing and re-purposing.

2 EXPERIMENTAL ENVIRONMENT

2.1 Experimental Environment - Main Paper

For conducting experiments for the main paper, we use a Mac M1 laptop to connect to a remote computer supporting `jupyter lab` through `ssh` command. All the experiments are run on that remote computer. It has 10 AMD EPYC-Milan CPU Processors, 24 GB RAM Memory, and 4 `Nvidia Tesla T4` GPUs. The Operating System is Ubuntu 20.04.2 LTS, and CUDA version is 11.2. The recommended free space to save all experimental data is at least 20 GB.

The software packages and their corresponding versions to run the programs in the main paper are as follows:

- `python`: 3.8.10;
- `numpy`: 1.21.1;
- `matplotlib`: 3.4.3;
- `pandas`: 1.3.1;
- `torch`: 1.9.0+cu102;
- `d2l`: 0.17.0;
- `tokenizers`: 0.10.3.

2.2 Experimental Environment - VM

The Virtual Machine [1] can be found in folder `/material/VM` after extracting the `Archive.zip` file. The software packages installed in this virtual machine are the same as mentioned in 2.1 except some minor changes of the versions.

Besides, this Virtual Machine only supports CPU, and does not support GPU. This means, by using this VM, the reviewer will only be able to adopt the Fast Track (will be described in 4.1) to reproduce the main results of the paper. This track only involves testing the neural networks that have been trained so GPU is not needed. For evaluating this VM, the minimum hardware requirement is a computer with 4-core CPU, 16 GB RAM Memory and 30 GB free space.

Our artifact is stored in folder `/material/artifact`. We have copied this folder to the desktop of this VM, so the reviewer can directly evaluate our artifact following the instructions of Section 3 and Section 4. Before evaluation, please use the VM to run `setup.py` in folder `/artifact`. If the screen prints the versions of all the software libraries listed in Section 2.1 without raising any error, this means the environment has been correctly set up.

2.3 Experimental Environment - DIY

If the reviewer fails to run the submitted VM, or the reviewer wants to adopt the Complete Track (will be described in 4.2) by training neural networks from scratch, she/he may need a

[1] username & password: icse22ae

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510171

computer with hardware configurations similar to ours as mentioned in 2.1, and should have at least one GPU with memory no smaller than 10 GB. For more information regarding setting up environment, after extracting the `Archive.zip` file, please refer to `/documents/REQUIREMENTS.txt` and `/documents/INSTALL.pdf` for more detailed instructions.

Finally, make sure to copy folder `/material/artifact` to the reviewer's machine, and run `setup.py` in this folder to check whether all software libraries are correctly installed.

3 EMPIRICAL STATISTICS IN SECTION 4

The files for empirical statistics can be found in folder `/artifact/reference_analysis`. In this folder, our database is stored in an Excel file named `database.xlsx`. The reviewer needs to install Microsoft Office or other software to open this database.

There are also some auxiliary programs to facilitate analyzing this database in an automatic or semi-automatic way.

Enter folder `/reference_analysis/reference_statistics`, running `operator_statistics.py` [2] will count and display the occurrence times of STL operators, which corresponds to Figure 1 in the main paper. `stl_analysis.py` is used to transform STL formulas into templates. Based on the printed results by running `stl_analysis.py` [3], further analysis needs to be done manually to obtain the proportions of the four STL templates in our database (shown in statistics of Section 4.1.1). The more detailed information regarding template can be found in Tab 2 of `database.xlsx`. The data for Figure 2 in the main paper is also based on Tab 2.

The data in Section 4.2.1 (Example 1 and 2) and Section 4.2.2 (in the end) are from Tab 3 and Tab 4 in `database.xlsx`. In this part, all analysis regarding natural language specifications is done manually.

4 PROGRAMS IN SECTION 5 AND SECTION 6

In this Section, since the software/hardware requirements are not identical for different levels of evaluation, and it is also time-consuming to generate the data-set in Section 5 and train neural network in Section 6, we provide two tracks for the flexible evaluation of our artifact:

(1) **Fast Track** : In this track, reviewers are able to obtain the main results in Section 5 and Section 6 by simply running several scripts within minutes. These scripts will load the minimum data required that has been stored in advance. For those interested in this track, please enter the folder `/artifact/fast_track`.

(2) **Complete Track**: This track only provides pure codes without any data generated from them. Therefore, reviewers will start from scratch to run programs written for Section 5 and 6. In this case, based on our hardware configuration, it will take several hours to generate the data-set and get corresponding results mentioned in Section 5, and take approximately 2 days (with 4 `Nvidia Tesla T4` GPUs running in parallel) for going through the whole procedure of training and testing the deep neural networks mentioned in Section 6. For those interested in this track, please enter the folder `/artifact/complete_track`.

Reproducibility of Fast Track Since there is no randomness for computation with the loaded data, all the main results (Table 1-6, Figure 6-8, translation examples for extrapolation in Section 6.3.3) can be reproduced. The only minor differences may result from different numerical precision of different platforms, or the changes of library versions. However, these differences are often negligible.

Reproducibility of Complete Track For data-set generation, although random sampling is involved in the program of the generator, since only CPU is used, there is a large likelihood that the reviewer will be able to regenerate the same data-set used in the paper with the given random seed in the scripts. The successful reproduction of the same data-set has been tested on a Macbook M1 laptop and a Thinkpad laptop.

For training neural network, however, there is no guarantee that the same results are reproducible across different GPUs and different Pytorch release versions [2]. For comparison of different hardware, we tested the training results step by step using another `GeForce GTX TITAN X` GPU, and found that even with the identical random seed, the training results were different from what we got using one `Nvidia Tesla T4` GPU.

For Figure 7-8 and Table 5 in Section 6, although the exactly identical results are difficult to get on different platforms or with different versions of software packages, very similar results can be still obtained. The training procedure and the metrics calculated in inner-test are very stable. However, for extrapolation tests mentioned in Section 6.3.3, the prediction results are sensitive to minor changes in the parameters of the neural network, so we do not guarantee that very closed results can be derived.

Despite the inherent non-determinism that is unavoidable, our artifact makes sure that: (1) *The same results are reproducible on the same platform with the same versions of software packages*; (2) *On the same platform, if anything else is fixed, then every time when the same file recording the parameters of the network is loaded, the stable reproduction of the identical results can be guaranteed.*

4.1 Fast Track

In this subsection, we only provide instructions about how to obtain relevant results in a quickest way. Therefore, we only mention the files that are needed for this track. For a more detailed description of the file system of the artifact, and the functions of relevant codes, please refer to the the introduction of Complete Track. All related folders and files in this track are in the folder of `/artifact/fast_track`. For simplicity, let's assume all operations in this subsection are in this folder.

4.1.1 Results in Section 5. This part explains how to run the code producing the corpus statistics shown in Figure 6, and Table 1-4 in Section 5 of the main paper.

Enter folder `/fast_track/data_generation`, where in the `/corpus_statistics` folder there are three `Python` scripts, and they work as follows:

(1) `operator_statistics.py` : This program counts the number of different STL operators in `corpus_split.csv` within

[2] For how to run this Python script please refer to Sectoin 4.1.1 (1).

[3] The displayed results include: (1) Templates for the 130 collected STL formulas; (2) Template frequency; (3) Total number of formulas in our database; (4) STL templates with their occurrence frequencies printed. These results are not directly presented in the main paper.

the folder. These results are used to plot Figure 6 in the main paper.
Execution: Open terminal in folder `/corpus_statistics`, and input: `python3 operator_statistics.py`. If an IDE (like Pycharm) is installed, the user can also run this program using the IDE. All following `Python` scripts can be run in the same way.
Observation: The user will see the occurrence number of each STL operator in our generated corpus. They are directly printed on the screen.
Consistency: The printed numbers are consistent with Figure 6 shown in the main paper.

(2) `stl_analysis.py`: This program generates the statistical results given in Table 1-3 in the main paper.
Observation: The user will see all data in Table 1-3 of the main paper. The printed results are self-contained.
Consistency: The printed results are consistent with the numbers in Table 1-3 of the main paper.

(3) `eng_analysis.py`: This program generates the statistical results presented in Table 4 in the main paper.
Observation: The user will see all data in Table 4 of the main paper. The printed results are self-contained.
Consistency: The printed results are consistent with the numbers in Table 4 of the main paper.

Through running the above three `Python` scripts, relevant results in the main paper can be obtained. One can also find the data-set that is used in the main paper, which is just `corpus_split.csv` in folder `/corpus_statistics`.

4.1.2 Results in Section 6. This part explains how to run the codes producing data for plotting Figure 7-8 and entering Table 5-6. Guidance of how to get translation results for extrapolation test shown in Section 6.3.3 is also provided.

First enter folder `fast_track/NLP`, and strictly follow the steps below [4]:

(1) *Plot Figure 7-8* : Run `/NLP/data_plot.ipynb` using `jupyter notebook`, then the program will directly load relevant data and does some computation to plot Figure 7 and Figure 8.
Execution: Open terminal in folder `/NLP`, and input: `jupyter notebook`. Then the browser will be initiated. On the web page, please enter `data_plot.ipynb`. In the first cell, please press `shift + return` on keyboard, then the first cell will be executed. All `jupyter notebook` files can be executed in the same way.
Observation: Under the first cell of `data_plot.ipynb`, the user will see two figures plotted.
Consistency: The two figures are exactly Figure 7 and Figure 8 of the main paper.

(2) *Compute Data for Table 5-6*: Run `/NLP/data_analysis.py`, and this program will also load relevant data. With some computation, all data used to enter Table 5-6 will be printed.
Observation: The user will firstly see a line printed as "Table 5 - inner test". Under this line are the testing results for each model. For example, the five testing results of seq2seq model will be initially printed. Under these results are three metrics averaged on these five tests, and they are "string acc", "template acc" and "bleu" respectively. The data for "string acc" corresponds to row 2, column 2 of Table 5; The data for "template acc" corresponds to row 2, column 3 of Table 5; The data for "bleu" corresponds to row 2, column 4 of Table 5.
The rest data of Table 5 and Table 6 can be observed in the same way.
Consistency: The printed results are consistent with the numbers in Table 5-6 of the main paper.

(3) *Get translations in Section 6.3.3*: Run `transformer_test.py` (Transformer), `attention_test.py` (Att-seq2seq) and `seq2seq_test.py` (Seq2seq) without any modification to the codes, then these three models' translation results for extrapolation tests will be printed. The three translation examples in Section 6.3.3 of the main paper are from test case 5, 12 and 14.
Observation: For each model, the user will see the results of 14 test cases. For each test case, the output sequence is: (1) English requirement; (2) Reference STL formula; (3) Reference STL template; (4) Predicted STL formula (the first output is an intermediate result with identifiers and constants separated by whitespaces, while the second output shows the final translation that is displayed in the three examples listed in Section 6.3.3 of the main paper); (5) Predicted STL template; (6) Criterion (corresponding to confidence level denoted as C_t, C_a and C_s in Section 6.3.3 of the main paper).
Consistency: The translation is consistent with what is presented in the main paper, while the confidence level may be slightly different from the numbers in the main paper with very minor changes. This difference is caused by different numerical precision of different platforms, e.g., GPU and CPU.

4.2 Complete Track

In this subsection, we provide step-by-step instructions of how to run our artifact from scratch without any intermediate data stored in advance. Let's enter folder `/artifact/complete_track`. Now we assume that all operations in this subsection are in this folder.

4.2.1 Guidance for using the corpus generator. Please enter folder `/complete_track/data_generation`, in which you can find `main_generation.py`. This is the program for generating the corpus. To better use this script, the user is recommended to read the following two preparations before start.

Preparation 1 Before formally introducing the procedure of data generation, it is better to firstly see what the generated corpus looks like by going back to the root folder of Complete Track and

[4] The results computed in (3) will change slightly on different platforms, and these data will be written into corresponding recording files. However, the computation results of (2) will depend on these recordings. Hence, executing (3) before (2) will potentially influence checking the consistency of (2). Strictly following the below sequence will make sure that all data in (2) are from original files without change.

entering folder /data_set. This will help the user understand the outputs of the generator.

In folder /data_set, you will find four files respectively named as STL_formulas.txt, corpus_id.csv, corpus_no_split.csv, and corpus_split.csv. These four files are what we generated for our main paper, among which corpus_split.csv is used for computing statistical results. It also serves as the data-set for machine translation in Section 6. Please open these four files. We will provide short descriptions for each of them.

(1) STL_formulas.txt : This file contains 24,000 STL formulas that have been randomly generated. It can be easily observed that the identifiers and constants in these formulas are occupied with indexed placeholders. We will explain the reason for using placeholders soon after.
(2) corpus_id.csv: This .csv file has three columns. The first column lists STL formulas and the second column shows their corresponding English translations. The third column is the type of the STL formula in the first column. For the STL formulas in the first column, you will find they all come from STL_formulas.txt, with the same formula copied for 5 times. Then in the second column, one by one, there are 5 different synonymous translations associated to these 5 copies. Now our generation strategy becomes clear for understanding: We randomly generated 24,000 STL formulas (this corresponds to a hyper parameter called formula_num in main_generation.py), and for each formula, 5 different English translations were randomly assigned (the number of English translations for one formula generated in STL_formulas.txt corresponds to a hyper parameter called limit_num_formula in main_generation.py). Consequently, there are overall $24,000 \times 5 = 120,000$ parallel STL-English pairs generated.
(3) corpus_no_split.csv: Now the main problem left is the generation of different identifiers and constants. This will be solved by the placeholders. We defined two sets of rules for using placeholders to respectively represent identifiers and constants in the formula. Through these rules, we can easily replace any placeholder with either a specific identifier or specific constant randomly generated from something like regular expressions. The methods for generating identifiers and numbers can also be flexibly designed. corpus_no_split.csv is then obtained after the generation and the replacement. In this file it can be found that the 5 copies of the same formula in corpus_id.csv now have different identifiers and constants, and so do their English translations. This operation not only preserves the diversity of translations for STL formulas sharing the same structure, but also avoids repeatedly using the same identifiers and constants so that for the 120,000 STL-English pairs generated, each formula and its translation are unique.
(4) corpus_split.csv: In principle corpus_no_split.csv is already a complete data-set. However, it cannot be directly used for machine translation in Section 6 because in the data pre-processing phase, the tokenization method chosen in the main paper requires each identifier and constant to be split into characters and digits, and for each pair of two adjacent characters and digits, there should be a whitespace between them. corpus_split.csv is used to fulfill this requirement [5], so only this file will be used for computing corpus statistics and machine translation in the main paper. The reason to keep the other three files is to make our data generator compatible to other data pre-processing approaches.

Preparation 2 Every time when one would like to start generating a new data-set, the first thing is to make sure that all the data should be written into empty files. Therefore, please enter folder /data_generation/data/empty. There are four empty files respectively named STL_formulas.txt, corpus_id.csv, corpus_no_split.csv, and corpus_split.csv. They are the same files as introduced in **Preparation 1** except that they are empty. Then copy these four files to folder /data_generation/data so that the generated data can be correctly written to them.

If the user does not want to start generation from scratch, then the generator will append new data to the four files that already existed in /data_generation/data. Please remember this folder because it is where the generated data-set gets saved.

Start Generation Let's go back to folder /data_generation to formally start data generation after finishing the above two preparations. Open main_generation.py and now the meaning of the following hyper parameters are easy to understand:

(1) formula_num : The value of this hyper parameter determines the number of STL formulas that can be randomly generated in the first round to STL_formulas.txt.
(2) limit_num_formula: As introduced for the description of corpus_id.csv in **Preparation 1**, this hyper parameter specifies the number of copies that should be made for one formula from STL_formulas.txt to corpus_id.csv. Correspondingly, this hyper parameter equals to the associated number of synonymous translations for these copies. Hence, the total number of parallel STL-English pairs is formula_num × limit_num_formula.
(3) limit_num_clause: This hyper parameter specifies how many English translations can be provided to one clause, for example, the pre-condition or the post-condition. This value will influence generation speed. The default value is set to 100.

The random seed set to generate our data-set in the main paper is 100, which can be found in the beginning of main_generation.py. After fixing random seed, the user will be able to reproduce the same generation data on the same computer.

Corpus Statistics After generation finishes, if one wants to compute statistics for the generated data-set, please first go to folder /data_generation/data, and copy corpus_split.csv to folder /data_generation/corpus_statistics. Then the next steps are the same as introduced in 4.1.1.

[5] In this file only identifiers are split, because numbers can be easily recognized and split using regular expressions. This function is internally supported by the third-party library used in the data pre-processing stage of neural translation.

Special Notes Since global variables in Python are used for generating data, if the user wants to do some unit test to other scripts in sub-folders of /data_generation, errors may arise. In this case, please go to folder /data_generation/corpus/unit_test, copy the two files in this folder, and replace the files with the same name in folder /data_generation/corpus. If the user wants to return to the generation mode, in the same way, please go to folder /data_generation/corpus/generate, and replace corresponding files in folder /data_generation/corpus.

The second note is about the proportions of the four types of STL formulas. Their generation probabilities can be modified in file /data_generation/console/training_sample_generator.py. Go to function /formula_type_select() (line 47-64), then these numbers can be directly changed. If the user has some advanced needs to modify other relevant hyper parameters in the program, please refer to file /data_generation/public/parameters.py.

4.2.2 Guidance for neural translation. To start with, please enter folder /complete_track/NLP. All codes for the neural translation program are in this folder. Please note that the root folder is only allowed to be called NLP, otherwise errors will arise. This requirement will facilitate locating correct paths in different hierarchies for reading and writing files. These operations happen frequently in the program. The introduction of this part follows the sequence of data pre-processing, training, testing, and data analysis.

Data Pre-processing This step will load the data-set generated, divide train/dev and test sets, generate tokenizer, and calculate maximum sequence length.

First, copy the generated data-set corpus_split.csv to folder /NLP/data_preprocessing/subword/dataset, then go back to folder /NLP and run subword_preprocessing.py. Once this procedure is done, a corresponding Python Dictionary file called preprocess_info_dict will be generated in folder /NLP/data_preprocessing/subword. This file will be used for both training and testing.

The hyper parameters in data pre-processing can be adjusted in file NLP/public/hyperparameters.py. One thing that should be noted is the meaning of dev_ratio. It is the proportion of the validation set in the remaining data-set after the test set has been split out. The random seed used in data pre-processing is 100, which can be found in subword_preprocessing.py.

Training For training each architecture, go to folder /NLP and run seq2seq_train.py (Seq2seq), attention_train.py (Att-seq2seq) and transformer_train.py (Transformer) respectively. We will use the Transformer architecture as an example to illustrate important settings and files that are relevant to training. It is the same for the other two architectures.

(1) **Step 1:** Please go to folder /NLP/transformer, in which the python script transformer_hyperparas.py is used to set hyper parameters for the Transformer model and its training and testing procedure; File info.txt is used to record highlight settings in training.
(2) **Step 2:** Next go to folder /NLP/transformer/record, in which you can find there are 5 folders named 100, 200, 300, 400 and 500 respectively. The name of these folders represents the random seed number used in the main paper for each training experiment. For example, if a training experiment is run using random seed 100, then all intermediate data is generated in the /100 folder, which incorporate:
   (a) data_iter_dict: This is a dictionary file including two iterators for respectively generating a batch of training and validation data in a randomized way.
   (b) checkpoint_dict: This is a dictionary file including all related information during training: (1) The whole network model; (2) The network state (e.g., the values of all parameters); (3) The state of optimizers; (4) maximum epochs; (5) epochs that have finished; (6) Next training step used to calculate learning rate (training one batch corresponds to one step); (7) The device that is used for training (GPU/CPU); (8) Lists of average training loss and training accuracy of each epoch; (9) Lists of average validation loss and validation accuracy of each epoch; (10) Randomness states of the following three libraries python.random, numpy and torch.
   (c) net_state_dict: Since the size of checkpoint_dict tends to be very large, and the parameters of the network will be used in testing, (2) in checkpoint_dict will form an independent file.
   (d) info_dict: We also hope that the data source of Figure 7-8 and Table 5-6 can be stored in one file, and this file should be portable, so this dictionary file includes (8) and (9) of checkpoint_dict plus results of testing accuracy.
   (e) log.txt: This text file records all the log information during training and validation (e.g, loss and accuracy for each batch). This file is helpful for tracing back to the performance of each training step, and it can also be used for reproduction checking.
   Note: Considering the large size of file (a) and file (b), in the Fast Track, we only reserve files (c)-(e). These three files are enough for testing, computing data and reproduction checking.
(3) **Step 3:** Go back to /NLP/transformer_train.py. In the beginning of this script, there is a comment saying "IMPORTANT SETTINGS", below which there are two hyper parameters that must be set before running this script.
   (a) seed: The user has to set a random seed x in advance. Besides, the user also needs to create an empty folder named x in /NLP/transformer/record. Otherwise error will arise.
   (b) train_from_start: This is a Boolean variable only with value True or False. Normally, this hyper parameter should be always set to True. However, there are occasions where the training program gets interrupted accidentally. For example, network failure when connecting to remote server, or power outage. In this case, suppose the user uses random seed x, and training is interrupted in epoch m, then firstly go to /NLP/transformer/record/x, and under this directory open log.txt, delete all the log information for epoch m. Then go back to set train_from_start as

`False` and run `transformer_train.py`. The program will read checkpoint data, and continue training from the beginning of epoch `m`.
Note: For all training experiments in the main paper, we always set `train_from_start` as `True`. Even if we encountered accidental interruption, we still chose to train from start. This is because our network was trained on GPU, and if we continued from middle during training, for the subsequent results, there will be minor differences in loss compared to occasions without interruption. Even though we restored randomness states, this problem still occurred, which affected strict reproduction in training. We suspect this problem may be due to CUDA, because we did not observe this phenomenon when using CPU for training.

(4) **Step 4:** Run `transformer_train.py` after all configurations are set up. It will take hours to finish training. The user will see the training results printed on screen in real time, and also could see the comparison between prediction and reference for the last sample of each batch.

For the Transformer model, we ran `transformer_train.py` five times with random seed set to 100, 200, 300, 400 and 500 respectively. Since we have multiple GPUs, the training was conducted in parallel on different GPUs. For example, GPU 1 is used to train the model using seed 100, and GPU 2 is used to train the model using seed 200. The user could go to `/NLP/transformer/transformer_hyperparas.py` to set the value of `device` to decide which GPU should be used. It is the same with the other two models.

Testing After all the training experiments (15 experiments, 5 for each model) for the three models finish, testing can start. Still in the folder `/NLP`, run `seq2seq_test.py` (Seq2seq), `attention_test.py` (Att-seq2seq) and `transformer_test.py` (Transformer) respectively, the user could get testing results for each model. For simplicity, we still take Transformer model as an example for illustration.

(1) **Step 1:** Open `/NLP/transformer_test.py`. In the beginning of this script, there is a comment saying "IMPORTANT SETTINGS", below which there are three hyper parameters that must be set before running this script.
   (a) `predictor`: This hyper parameter specifies the translation mode for testing. There are two options. The first one is `predictor='greedy'`. This option adopts greedy search for decoding each token, which means for every step, the decoder selects the token with the highest probability for prediction. The second option is `predictor='beam'`, which adopts beam search algorithm for prediction. Briefly speaking, for every step, a beam-search based decoder will keep the best k candidate sequences based on the criterion of average outputting confidence until this step. This extends the searching scope in some way. The value of k can be tuned by adjusting `topk` in file `/NLP/transformer/transformer_hyperparas.py`.
   Note: When $k = 1$, beam search is equivalent to greedy search. In our program, by default we set `predictor='beam'` and `topk = 1`, so we actually used greedy search algorithm. The comparison between these two decoding approaches are not discussed in the main paper because for our translation problem, we did not collect enough evidence to prove which one is better. But it is an interesting research direction in the future.
   (b) `seed_list`: This is a list including the folder name(s) (represented as random seed number used in training) where network data is stored. In this way, the testing program will initially load the network parameters from the folder named after the first element (e.g., If the first element is 100, then the path will be located to `/NLP/transformer/record/100`), and does relevant tests for it. Then one after another, the testing program will read the network parameters from the folder named after the second element, and does the same sets of tests.
   (c) `mode_list`: This list specifies what kinds of tests will be made in a sequential way for one element in `seed_list`. There are three options:
      (i) `inner_test`: This option corresponds to Section 6.3.2 in the main paper, and will perform tests on the testing set split in the data pre-processing phase. These testing cases are within our generated data-set but have never been trained and validated. In this mode, for every 100 testing cases, the screen will print the testing result of the last one, and the average accuracy metrics until the last testing case. Finally the average accuracy metrics over all testing cases will be printed after inner test finishes.
      (ii) `extrapolate`: This option will conduct extrapolation test on the 14 testing cases found in our database. This test corresponds to Section 6.3.3 in the main paper. In this mode, the screen will print the predicted sequence and the reference translation for each testing case, and finally print the average values of accuracy metrics. The English requirements of the 14 testing cases can be found in file `/NLP/test_cases/test_case_eng.txt`, and the corresponding reference STL formulas are written in a text file in the same folder named `test_case_stl.txt`.
      (iii) `output_results`: This option will directly load `/NLP/test_cases/test_cases.txt`, and one after another, translate the English requirements written in this file into STL formulas. Currently, many of the English requirements in this file are modified from our synthetic and extrapolation examples. The user can manually add new requirements and test the translation results.

(2) **Step 2:** Run `transformer_test.py` after all configurations are set up.

The testing procedure for the other two models are identical. For each model, it will take one to four hours to finish inner test and extrapolation test from the 5 training results. When this program is running, corresponding testing accuracy metrics will be written to `info_dict` as mentioned before. For example, suppose the Transformer model has finished testing the network trained with random seed 100, then these testing accuracy metrics will be written to `/NLP/transformer/record/100/info_dict`.

Data Analysis After finishing all training and testing experiments, data analysis can be started. The next steps are the same as introduced in 4.1.2. We also created three `jJupyter notebook` scripts named `seq2seq_analysis.ipynb`, `attention_analysis.ipynb` and `transformer_analysis.ipynb` to facilitate observing training and testing results for each model when random seed is provided.

REFERENCES

[1] Artifact link. https://figshare.com/s/e35ccfcd8c60b78a8486.
[2] Pytorch reproducibility. https://pytorch.org/docs/stable/notes/randomness.html.

A APPENDICES

A.1 Corpus Generation

This subsection mainly describes the main structure and functions of the source codes for our corpus generator, which covers Section 5 in the main paper.

A.1.1 File structure. This part explains the file structure of the corpus generator. Please first enter folder `/artifact`, then go to folder `/complete_track/data_generation`, which incorporates all pure source codes.

In this folder, there are three `Python` scripts. The first one is `main_generate.py`. As introduced before, this script is the main entering port of the program, and is responsible of generating the whole corpus. The other two scripts are `global_variables_id.py` and `global_variables_num.py`. They will participate in controlling the generation of identifiers and constants, which will be firstly generated as global variables and then get replaced with randomized strings or digits according to their unique placeholders. Normally the latter two scripts do not need any modifications, so please keep them unchanged if the user does not have specific needs.

The functions of each folder in directory `/data_generation` are briefly illustrated as follows:

(1) `/data`: As introduced in **Preparation 2** of Section 4.2.1, this directory stores all the generated data. Please note that, in the beginning, there are no files in this directory except a sub-folder named `/empty`. The user should firstly enter `/empty`, and copy all the four empty files in it to folder `/data`. This is to make sure that if the user wants to generate a data-set from scratch, the program will always start writing data to clean and empty files.

(2) `/public`: As introduced in the end of **Special Notes** of Section 4.2.1, this directory only includes a script named `parameters.py`. They are some hyper-parameters in the program and can be tuned easily according to the comments. But please note that these hyper-parameters will be adjusted only in special occasions, e.g., the user has some advanced needs. In normal cases, it is better to leave them unchanged.

(3) `/corpus`: The scripts in this folder contain the words or expressions for organizing the translation templates. For example:
   (a) `basic_words.py` contains all the verbs and their variants used in the generated corpus.
   (b) `adverbial_modifiers.py` contains all the temporal adverbial modifiers used in different occasions; They are the source of *Adverbial Information* in Figure 3 of the main paper.
   (c) `conjunctions.py` includes all the conjunctions considered to concatenate two clauses.
   (d) `template_words.py` provides all the words or expressions for organizing translations of various atomic propositions and temporal phrases.
   (e) `sig_mode_name_generate.py` will randomly generate identifiers, and `number_generate.py` will randomly generate constants. For the usage of these two scripts in different scenarios, please refer to the first paragraph of **Special Notes** in Section 4.2.1 for details.

   Please note that, since `Python List` is pervasively used in the above files, we have reserved enough space to add synonymous utterances that can be typically used but that are not included in our generator.

(4) `/template`: There are two scripts in this directory. The codes in `atom_expr_template.py` will leverage the structure of `Python Dictionary` to assemble the templates of subject, predicate and object of the English translation for different atomic propositions. `TP_template.py` mainly does the same job but pays more attention to assembling the templates of adverbial modifiers for translating different temporal phrases.

(5) `/commands`: The scripts in this directory will form *Predicate Commands* (see Figure 3 and Figure 4 in the main paper) in different scenarios, which will influence the choice of verbs, their format, and the use of modal verbs and of adverbial modifiers.

(6) The scripts in the above five directories actually play an auxiliary and supporting role. The main functions of the corpus generator are provided by the scripts in the following three directories: `/grammar`, `/selection` and `/translation`. The file structures of these three directories are very similar - they all incorporate the below four sub-folders:
   (a) `/level_1`: The codes in this sub-folder from `/grammar`, `/selection` and `/translation` respectively function as grammar definition, random selection and translation of the **simple-phrase (SP)** layer mentioned in Section 5.1.1 of the main paper.
   (b) `/level_2`: The codes in this sub-folder from `/grammar`, `/selection` and `/translation` respectively function as grammar definition, random selection and translation of the **temporal-phrase (TP)** layer mentioned in Section 5.1.1 of the main paper.
   (c) `/level_3`: The codes in this sub-folder from `/grammar`, `/selection` and `/translation` respectively function

as grammar definition, random selection and translation of the **nested-temporal-phrase (NTP)** layer mentioned in Section 5.1.1 of the main paper.

(d) `/medium`: The codes in this sub-folder from `/grammar`, `/selection` and `/translation` leverage the codes in `/level_1` and `/level_2` to manage the grammar definition, random selection and translation of STL formulas with a special template - the *Invariance/Reachability* category as defined in ψ in Section 5.1.1 of the main paper.

Figure 3 and 4 in the main paper actually demonstrate the cooperation of the codes in sub-folder `/level_1/atom` from `/grammar`, `/selection` and `/translation`. Figure 5 and its explanation in the main paper actually illustrate the cooperation of the codes in sub-folder `/level_2` of the above three directories.

(7) `/corpus_statistics`: As mentioned before, the scripts in this folder will compute the statistics of the generated corpus.

(8) `/console`: The script `training_sample_generator.py` in this directory takes charges of the main workflow of data generation. The function of this file will be briefly described in A.1.2.

A.1.2 Console of Generation. In `training_sample_generator.py`, there is a class `TrainingSampleGenerator` that functions as a console for data generation:

(1) **Step 1:** To begin with, function `formula_type_select()` will randomly select a type of STL formulas for generation according to the probabilities obtained from corpus statistics in Section 4 of the main paper.

(2) **Step 2:** Based on the selected type, we come to a branching scenario:

(a) *Invariance/Reachability*: For this type, the program will execute function `group1_processing()` to generate an STL formula belonging to this category and its associate English translations.

(b) *Immediate response*: For this type, the program will execute function `group2_processing()` to generate an STL formula of this category and its associate English translations. As this category involves a condition and an obligation, `SP_clause1_processing()` and `SP_clause2_processing()` will be called in succession for further processing.

(c) *Temporal response*: For this type, the program will execute function `group3_processing()` to generate an STL formula belonging to this category and its associate English translations. The formula of this type also includes two clauses, so `P_clause1_processing()` and `TP_clause2_processing()` will be called sequentially for further processing.

(d) *Stabilization/Recurrence*: For this type, the program will execute function `group4_processing()` for generation and translation. `P_clause1_processing()` and `NTP_clause2_processing()`, similarly, will be called to assemble the two clauses of the formula and the translation.

(3) **Step 3:** After the end of the last step, the generated formula and its raw translations are stored in a `Python Dictionary`. Then this dictionary will be sent to finalize processing so that a complete formula and complete English translations will be assembled. Before writing to local files, we will then substitute the placeholders of identifiers and constants with randomly generated strings and digits.

(4) **Step 4:** In this final step, all the generated data will be written into corresponding files in folder `/data`.

A.2 Neural Translation

This subsection mainly describes the main structure and functions of the source codes for our neural translator, which covers Section 6 in the main paper.

A.2.1 File structure and functions. This part explains the file structure of the neural translator. Please first enter folder `/artifact`, then go to folder `/complete_track/NLP`, which incorporates all pure source codes.

In Section 4.2.2, we provided a detailed introduction for the file structure, and described the functions of different scripts used in data pre-processing, training and testing. Here as a supplementary, we summarize how the codes for each neural translation architecture is organized. Similar to Section 4.2.2, we will use the Transformer model for illustration. It is the same with the other two models.

In directory `/complete_track/NLP`:

(1) `transformer_train.py`: This script is used to train the Transformer model.

(2) `transformer_test.py`: This script is used to test the Transformer model after training. There are three main `Python functions` in this file.

(a) `multi_acc_compute()`: This function is used to conduct inner test as mentioned in Section 6.3.2 of the main paper.

(b) `extrapolate_acc_compute()`: This function is used to conduct extrapolation test as mentioned in Section 6.3.3 of the main paper.

(c) `output_test_cases`: This function is used to provide translations for testing cases as described in **Testing** (1) (c) (iii) in Section 4.2.2.

The testing procedure will frequently conduct several operations and computations on strings. The codes for operating strings are stored in folder `/NLP/str_process`:

(a) `string_accuracy.py`: This script is used to compute string accuracy as mentioned in Section 6.3.2 of the main paper.

(b) `stl_template_extractor.py`: This script is used to extract the template of a translated STL formula based on a maximum-match approach.

(c) `bleu.py`: This script is used to calculate the BLEU score as mentioned in Section 6.3.2 of the main paper.

(d) `eng_split.py`: This script is used to recognize and split identifiers for an input English requirement.

(e) `stl_compact.py`: This script is used to delete extra whitespaces in identifiers, constants, logical/algorithmic operators of the translated STL formula.

(3) `transformer_analysis.ipynb`: This `jupyter notebook` file is used to observe training and testing results for the Transformer model when random seed is provided.

There is another important directory related to the Transformer model, and it is `/NLP/transformer`. In this folder:

(1) `transformer_hyperparas.py`: This script is used to set hyper-parameters for the Transformer model and its training and testing procedure.
(2) `info.txt`: This file is used to record highlight settings for training.
(3) `/model`: This directory contains scripts for defining the model of Transformer.
(4) `/train_validate`: This directory contains two python scripts:
    (a) `transformer_trainer_validator.py`: This script controls the whole training and validation procedure.
    (b) `transformer_train_dev_iterator.py`: This script will divide training set and development set, and prepare two randomized data iterators for these two data sets.
(5) `/test`: This directory also contains two python scripts:
    (a) `predict_greedy.py`: This script is used to adopt a greedy-search decoding strategy for testing.
    (b) `transformer_predict_beam.py`: This script is used to adopt a beam-search decoding strategy for testing.
(6) `/record`: This directory stores all intermediate data generated in training and testing, as mentioned in **Training** (2) in Section 4.2.2.