2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Using Machine Learning in Register Automation and Verification

Nikita Gulliya, R&D Engineer, nikita [at] agnisys.com, nikitagulliya@gmail.com
Abhishek Bora, R&D Engineer, abhishek [at] agnisys.com, abhi12567@gmail.com
Nitin Chaudhary, R&D Engineer, nitin.ch [at] agnisys.com, chaudharyece@gmail.com
Amanjyot Kaur, R&D Engineer, amanjyot [at] agnisys.com, amanjyotkaur88@gmail.com
Agnisys Technology Pvt. Ltd. Noida, India

***Abstract* -Machine Learning (ML) is a powerful concept where trainable engine and a dataset can be made to predict future outputs. This concept has been leveraged to help users create IP and SoC specification. The ML algorithms have been embedded in software that is not visible to the end user, which makes creation of the register specification a simple and a less erroneous task. Having defined the specification, one can automatically create design code and verification environment. We present a novel way to capture design intent vis-à-vis addressable registers and sequences, use ML algorithms to identify patterns in the natural language description, and to automate the whole design and the verification environment from it. We also present our experience in converting System Verilog Assertions into plain English language using ML algorithms.**

I. INTRODUCTION

The machine learning algorithms have been implemented in two different areas a. the automatic register specification creation, which further create the corresponding RTL code from it and b. in verification by transforming assertions into plain English language.

The user can provide the description of a register in simple English language. This information is fed into the machine learning engine which identifies the type of register and provides the output according to it. The output prediction of the type of register which is generated is further processed to generate the appropriate RTL and UVM code.

A. Machine Learning Algorithm for Register automation

For the implementation of machine learning algorithm first all the registers are categorized into broader categories such as:

Status Registers: Under this category registers such as counter registers, interrupt registers, FIFO exists.

Special Registers: Registers such as paged, virtual registers, TMR, shadow registers etc. are put under the following category.

Control Register: Enumerations, FIFO, counter, lock registers can be categorized under these types of registers. The hierarchy can be further expanded by categorizing the lock register into independent (Key) and dependent (Lock).

Implementation Defined: This category is further divided in hierarchal form and defines registers which remains constant, registers which are reserved and registers which depends on external signal.

The classification has been made by putting together various industry level specification and commonly used registers. The machine learning algorithm has been trained with different description for each type of register. All the possibilities have been covered in which functionality of the register can be defined. Therefore, the machine learning algorithm identifies the register and its automatic code can be generated. The machine learning algorithm is programmed in python language using Keras library for deep learning for development of neural networks. Any register level specification can be converted into automatic code after giving it to the machine learning algorithm which predicts the type of register and generates code according to it.

B. Machine Learning in Verification

In terms of verification, one of the most crucial part of verification are assertions. However, they can be difficult to understand, and their intent is often lost in translation. The assertions can be made easy to handle with the help of machine learning algorithms. A System Verilog Assertion "decoder" has been implemented for this purpose. It can be fed with an assertion and it can provide the user with its output in English, or it can take the English language as an input and provide a corresponding output assertion for it. The machine learning model has been fed with lots of different types of assertions. The level of assertion varies from simple to complex. The focus of decoder is to simplify and communicate the intent of the concurrent assertion.

Different assertions written in the format described by the SystemVerilog standard are first parsed and converted into a machine learning engine trainable form. A machine learning algorithm then runs on this parsed data to train the engine for its corresponding output in English. The machine learning algorithm is also trained on data in the English language which contains the description of different concurrent assertions with its corresponding output assertions in SystemVerilog.

Machine Learning algorithms can be used for various purposes such as to interpret assertions, write different assertions, and validate the assertion written for its correctness by checking its description in English. Also, the special registers can be interpreted from its functionality described in English language and automatic code can be generated. Hence, these features hold the potential of making the complex life of an electronics engineer much simpler.

II. REGISTER AUTOMATION FLOW

The two main important aspects of working with Machine Learning is the Data and the algorithm. A huge dataset with different scenarios is considered which further helps the machine learning algorithm to make the right predictions.

A. Dataset creation

After the categorization of the different types of registers, thousands of samples for the dataset has been created for each type of register. Industry level specifications were studied closely and analyzed. The dataset is based on the data entry from the user point of view. The technical specification by which a user defines the functionality of the register is used to make the datasets.

Considering an example for the lock register, different datasets were created for different functionality of the register. The software write access of a register or register field can be locked based on the value of another register field or based on an expression consisting of different register fields or some external signal declared as input. Such a register for which the write access is locked is a "lock register". Other functionalities of a lock register are also supported which includes a register being set based on another register or register field value, a register being cleared based on some other register or a register fields value, locking of software toggle access of a register depending on value and locking of read access of a register depending on the value of another register or a register field. The diagram below shows the functionality of a lock register.

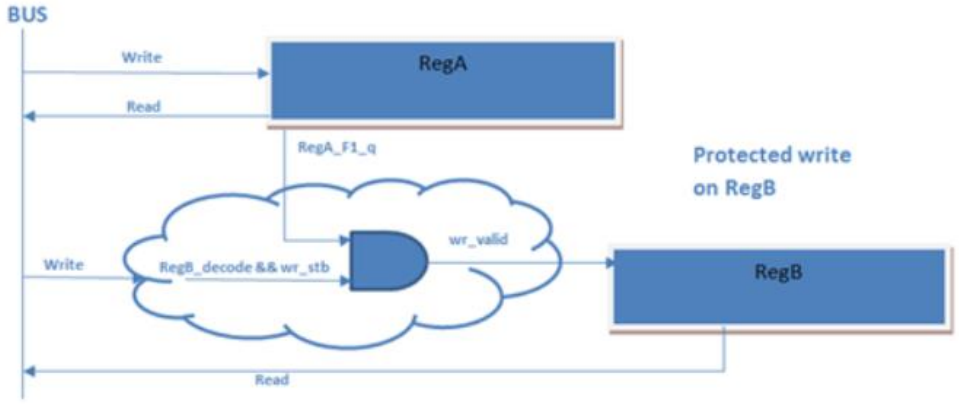


Figure 1. Lock Register operation

The dataset is created to identify above mentioned different types of functionality of lock register. For instance, a user can specify a functionality of register simply as “It will be set by writing a 1 to KEY_FIELD of KEY_REG” and the algorithm will identify that “it” in the sentence refers to the current register, and KEY_FIELD and KEY_REG refers to the register on which the locking depends. The algorithm will give the result that the lock register in which a value can be set depending on another register is been described and the RTL code will be generated according to it. Thousands of such descriptions are part of the dataset. The number of dataset has been kept equal for all the types of registers, since providing more number of data for one register can cause false results and machine learning model will start pointing towards the register for which more number of data is provided because the model will be trained according to it. The picture below shows the dataset of lock register.

```
seed = 1
np.random.seed(seed)
train = pd.read_csv("impDataTOtrain1.csv", sep=',',error_bad_lines=False,encoding = "latin1")
train.head()
```

	Sno	Description	output
0	1	When KEY_FIELD in KEY_REG is set then LOCK_FIE...	lock
1	2	When KEY_FIELD in KEY_REG is active then this ...	lock
2	3	When KEY_FIELD of KEY_REG is low, then write a...	lock
3	4	This field will be locked when KEY_FIELD of KE...	lock
4	5	Freezes pdiff signal and makes it unwriteable	lock

Figure 2. Sample of dataset

Similarly, lot of functionalities can be predicted with the help machine learning and automatic RTL code can be generated based on them. Another example can be taken for interrupt register, in which equal number of datasets are created for different functionalities of interrupt channel such as enabling the interrupt, pending of interrupt, status of interrupt, detection of interrupt, masking and overflowing of an interrupt. Apart from different types of register some additional information can also be specified, which are important while writing a specification, are helpful and often used. For example, declaring a field as reserved, clock domain crossing, repetition of a register or field, reset types, alias, counters to name a few.

B. Machine learning Algorithm

The Machine Learning algorithm has been made to predict the type of register and its functionality. It takes the description provided by the user as input and predicts the type of register, the result is further processed to generate the relevant RTL code for the same. The machine learning algorithm is written in python programming language and Keras Deep Learning Library has been used where, Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. TensorFlow an open-source software library which is a symbolic math library, also used for machine learning applications such as neural networks has been used. Therefore, installation of Keras, python and TensorFlow is prerequisite for the algorithm.

There are two main steps in which the algorithm has been implemented. The first step includes the defining the model in which the model is trained with data and complied, and the next step includes using the model for the appropriate output prediction.

After importing the necessary libraries. The seed value has been specified which is the random number generator, this specifies the number of times the code will be run and give the same result. The next important step is to load the data. The data is loaded in csv format. Some other important variables have been initialized such as the batch size, vocabulary size, embedding dimensions and maximum sequence length. The validation split is provided next. After that, the training data is split into two variables in which the unnecessary columns are dropped, and the input and output columns are assigned to different variables. For example, from the Figure 2, the SNo column is dropped and the description column is taken as input and output column is taken as output. The next step is tokenization in which text is to split into words, where these words are called tokens.

```
# Batch Size
batch_size=32
# Vocab size
vocabulary_size=20000
# Embedding Dims
embedding_size=EMBEDDING_DIM=300
# Max sequence length
MAX_SEQUENCE_LENGTH=50
VALIDATION_SPLIT=0.8
```

```
train=train.drop(['Sno'],axis=1)
```

```
train_df=train['Description']
```

```
y=train['output']
```

```
Using TensorFlow backend.
```

```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences

# Tokenizing
tokenizer = Tokenizer(num_words=vocabulary_size)
tokenizer.fit_on_texts(train_df)
sequences = tokenizer.texts_to_sequences(train_df)

word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))

data = pad_sequences(sequences, maxlen=MAX_SEQUENCE_LENGTH)


print('Shape of data tensor:', data.shape)
print('Shape of label tensor:', y.shape)
```

```
Found 190 unique tokens.
Shape of data tensor: (118, 50)
Shape of label tensor: (118,)
```

Figure 3. Training data and Tokenization

The data is then split into two forms the training and the test data. The training data is the data from which the model will be trained, and the test data will be the one on which the model will run to have predictions and to test the accuracy result of the model. The label encoding is done next to convert the text data into numbers for better prediction. OneHotEncode is done after the label encoding to avoid any kind of existence of hierarchy confusion.

```
# For splitting training and testing data
from sklearn.model_selection import train_test_split
```

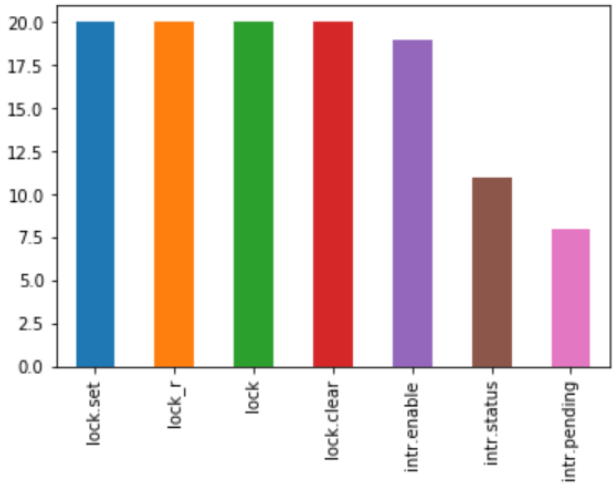
```
# Output : train_df['output']
encoder = LabelEncoder()
encoder.fit(y)
encoded_Y = encoder.transform(y)
# convert integers to dummy variables (i.e. one hot encoded)
Y = np_utils.to_categorical(encoded_Y)
print(Y)
```

```
[0. 0. 0. 1. 0. 0. 0.]
[0. 0. 0. 1. 0. 0. 0.]
[0. 0. 0. 1. 0. 0. 0.]
[0. 0. 0. 1. 0. 0. 0.]
[0. 0. 0. 1. 0. 0. 0.]
[0. 0. 0. 1. 0. 0. 0.]
[0. 0. 0. 1. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 1.]
[0. 0. 0. 0. 0. 0. 1.]
[0. 0. 0. 0. 0. 0. 1.]
[0. 0. 0. 0. 0. 0. 1.]
[0. 0. 0. 0. 0. 0. 1.]
[0. 0. 0. 0. 0. 0. 1.]
[0. 0. 0. 0. 0. 0. 1.]
[0. 0. 0. 0. 0. 0. 1.]
```

Figure 4. OneHotEncoding Output

```
train['output'].value_counts().plot(kind='bar')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x28e5b66f828>
```




```
X_train, X_test, y_train, y_test = train_test_split(data, Y, test_size=0.10, random_state=seed)
```

Figure 5. Sample plot of lock register and interrupt data

```
embedding_matrix = np.zeros((len(word_index) + 1, EMBEDDING_DIM))
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        # words not found in embedding index will be all-zeros.
        embedding_matrix[i] = embedding_vector
```

```
[-1.7490e-01  2.2956e-01  2.4924e-01 -2.0512e-01 -1.2294e-01  2.1297e-02
 -2.3815e-01  1.3737e-01 -8.9130e-02 -2.0607e+00  3.5843e-01 -2.0365e-01
 -1.5518e-02  2.5628e-01  2.2963e-01  1.1985e-03 -8.9833e-01  1.3609e-01
  1.8861e-01 -3.3359e-01  1.8397e-02  6.2946e-01 -1.3167e-01  6.4819e-01
 -2.1750e-01  9.3853e-02 -3.9050e-02 -5.0846e-01 -2.5540e-01  3.2361e-01
  2.3231e-01  4.9105e-01 -4.1841e-01  7.3934e-02 -6.5639e-01  4.8608e-01
 -1.1219e-01 -2.9994e-01 -7.2501e-01  8.5377e-02 -5.0447e-02  2.3105e-01
 -6.4843e-02  3.9056e-03  9.9742e-02 -2.0334e-02  3.8845e-01  2.4464e-01
 -8.6308e-02 -1.1308e-01  1.9281e-02 -1.1205e-01  6.5642e-02  1.8120e-01
 -1.0949e-01  5.5968e-02 -1.9700e-01  4.9184e-01  6.1818e-01 -3.3190e-02
  7.3289e-02 -2.2823e-02  6.6946e-01  1.8233e-01 -4.0082e-01 -3.3717e-01
 -2.8521e-01 -2.8222e-01 -4.4373e-02  1.4881e-01 -4.2135e-01  5.1545e-02
  2.7605e-01 -1.9959e-01 -2.9766e-01 -8.7712e-02  4.6210e-01  1.6891e-01
 -1.9415e-01  2.8327e-01 -2.5327e-01 -6.3275e-02  9.0945e-02 -1.8623e-01
  2.8891e-01  4.3534e-02 -1.0303e-01  3.9545e-01  8.8457e-02  5.4829e-02
 -4.5487e-01  3.8226e-01  1.5458e-01 -4.2001e-01  2.0908e-01  1.0261e-03
 -3.7166e-01  2.8856e-01 -7.2666e-03 -2.3869e-01  1.8698e-01  2.1457e-01
  2.4625e-03 -2.2166e-01 -1.0549e-01  2.6366e-01  6.3795e-01 -2.1856e-01
```

Figure 6. Embedding Matrix

A sequential model is made, and different layers have been implemented in the neural network of the model. The model is majorly based on 3 layers which are the embedding layer, LSTM and the Dense layer. The diagram below is representation of the three layers and their flow.

The first layer of the model is the embedding layer. This layer converts the integers into fixed sized dense vectors. The second layer is the LSTM which stands for Long short-term memory. It is a kind of recurrent neural network. It models time and sequence dependent behavior. The diagram below shows how LSTM works. Number of LSTM cells are defined here.

```
embed_dim = 300  # embedding dimensions
lstm_out = 128 # number of lstm cells

model = Sequential()
model.add(embedding_layer)
model.add(LSTM(lstm_out, dropout_U=0.25, dropout_W=0.25))
#model.add(Dense(4,activation='softmax'))
model.add(Dense(7,activation='softmax'))
model.compile(loss = 'categorical_crossentropy', optimizer='adam',metrics = ['accuracy'])
```

```
from keras.layers import Embedding

embedding_layer = Embedding(len(word_index) + 1,
                            EMBEDDING_DIM,
                            weights=[embedding_matrix],
                            input_length=MAX_SEQUENCE_LENGTH,
                            trainable=False)
```

Figure 7. Code snippet of embedding layer, LSTM and Dense layer

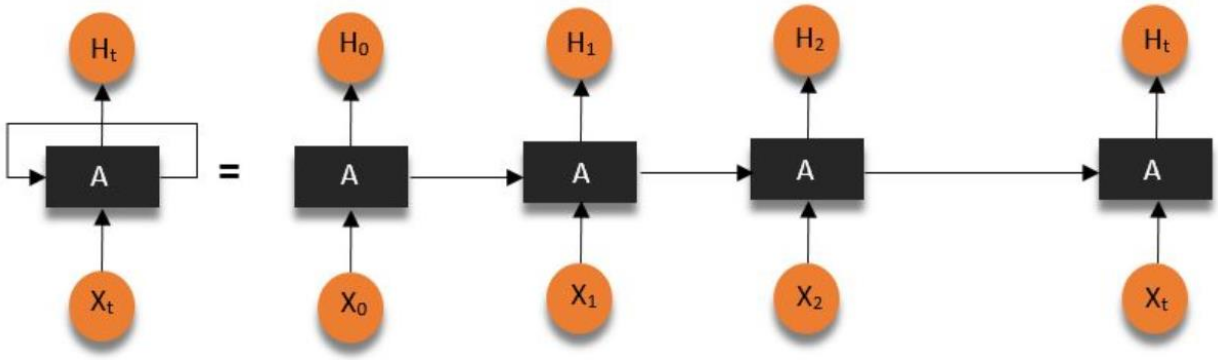


Figure 8. LSTM cells representation

The third and the final layer is the Dense or the softmax layer. This layer is used for the activation of the dense layers. The model is now ready, and the next step is to execute the model on data to train the model. The training process will run for a fixed number of iterations through the dataset called epochs, all the arguments including the training data's input and output, the batch size and the epochs values are given to the fit() function using the model. For example, the result below shows the training of model which contains 1006 samples running on the epochs value 50, and at the end the accuracy been calculated as 75%.

```
print("Accuracy: %.2f%%" % (scores[1]*100))
```

```
Train on 1006 samples, validate on 200 samples Epoch 1/50 1006/1006
[==============================] - 1s 11ms/step - loss: 1.9463 - acc:
0.1981 - val_loss: 1.8278 - val_acc: 0.3333 Epoch 2/50 1006/1006
[==============================] - 0s 3ms/step - loss: 1.6012 - acc:
0.3679 - val_loss: 1.8820 - val_acc: 0.0833 Epoch 3/50 1006/1006
[==============================] - 0s 2ms/step - loss: 1.3465 - acc:
```

```
0.5660 - val_loss: 1.4663 - val_acc: 0.4167 Epoch 4/50 1006/1006
[==============================] - 0s 3ms/step - loss: 1.0703 -

.     .     .

acc: 1.0000 - val_loss: 1.1568 - val_acc: 0.7500 Epoch 50/50 1006/1006
[==============================] - 0s 3ms/step - loss: 0.0031 - acc:
1.0000 - val_loss: 1.1459 - val_acc: 0.7500

  Accuracy: 75.00%
```

The second major step after the model is ready is to use this model. In which the first step is to evaluate the model. The model can be evaluated using the evaluate() function and the same input and outputs are provided for the evaluation which were used during the training. This will generate a prediction for each input and output pair and collect scores, including the average loss and any metrics you have configured, such as accuracy. Finally, the prediction is done using the predict() function for the model. The desired is printed using the print command. This output is predicted in form of a property which IDS understands. The model is dumped in JSON and integrated with IDS and the IDS understands the property and automatically generates the output RTL code based on it.

```
# serialize model to JSON
import h5py
model_json = model.to_json()
with open("model.json", "w") as json_file:
    json_file.write(model_json)
# serialize weights to HDF5
model.save_weights("model.h5")
print("Saved model to disk")
```

```
Saved model to disk
```

Figure 9. Model dumped in JSON

```
# Predicting
pred1=model.predict(X)

print('OUTPUT : ',encoder.inverse_transform(np.argmax(pred1))) # Predict
output column
```

Figure 10. Prediction of the output

III. SYSTEM VERILOG DECODER RING

The system verilog decoder ring has been implemented to covert the concurrent assertions into plain English text. The process can be reverted to obtain System Verilog assertions from plain English text.

The system verilog assertion (SVA) grammar has been written in ANTLR(Another Tool For Language Recognition) form. The SVA is then first parsed and converted into hierarchal form based on the grammar. The SVA is implemented in a rule-based form and in the next step simple NLP (Natural language processing) is used to define rule based English output for every system Verilog assertion operation. Both the inputs, the parsed SVA and simple NLP, are fed into the interpreter to provide the output in plain English text. The generated English output provided from this system

may not be grammatically correct, therefore, it is considered as bad English format. This bad English text is converted into the good English format with the help of machine learning algorithm. The Machine learning engine is first trained with the input as the bad English and output as its corresponding good English sentence. The BLEU (Bilingual Evaluation Understudy) score is then calculated for the sentences. The diagram below shows the flow of system Verilog assertion to the English output.

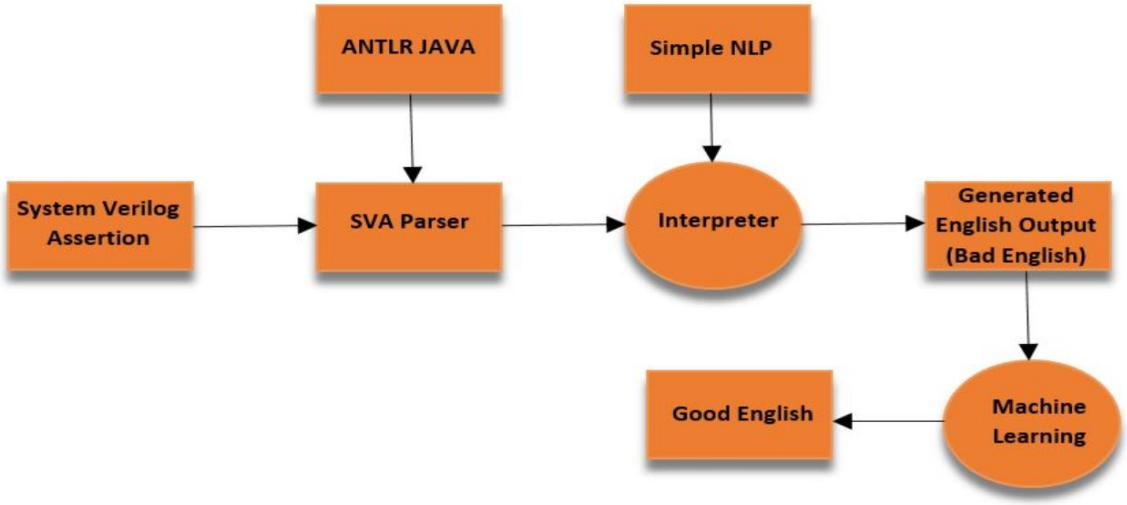


Figure 11. System Verilog assertion to English flow

III. RESULT AND CONCLUSION

The machine learning has been used in both register automation and System Verilog assertions. The diagram below shows the register automation flow.

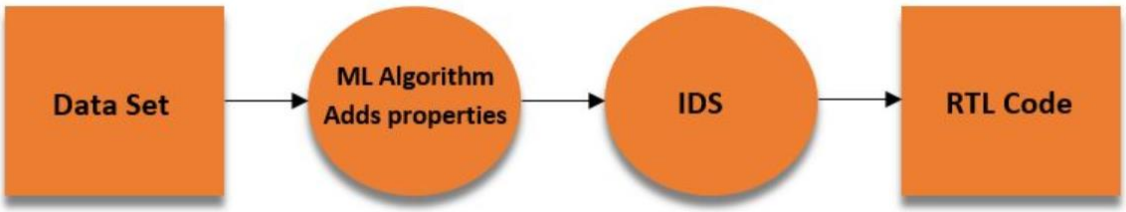


Figure 12. The Register Automation flow

The figure below shows the model summary. Where all the three layers have been represented.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 50, 300)           57300
_________________________________________________________________
lstm_1 (LSTM)                (None, 128)               219648
_________________________________________________________________
dense_1 (Dense)              (None, 7)                 903
=================================================================
Total params: 277,851
Trainable params: 220,551
Non-trainable params: 57,300
_________________________________________________________________
None
```

Figure 13. Model Summary

The below result is representation of sample system Verilog assertion given as input and the corresponding English output obtained.

System Verilog Assertion Input:

```
$rose(a) |-> (a throughout b [->1]) ##1 !a
```

English text Output:

Whenever a goes high, a must be high until b is asserted and after 1 clock cycle, a must be low

System Verilog Assertion Input:

```
a ##1 b [*1:$] ##1 c
```

English text Output:

a must be true on the first clock tick, c must be true on the last clock tick, and b must be true at every clock tick strictly in between the first and the last

System Verilog Assertion Input:

```
req ##2 gnt
```

English text Output:

req must be true on the current clock tick, and gnt must be true on the second clock tick after that

System Verilog Assertion Input:

```
start |-> (req ##2 gnt)
```

English text Output:

Whenever start is true, it implies that req must be true at the same clock tick and gnt must be true on the second clock tick after that

System Verilog Assertion Input:

```
a ##1 b [->2:10] ##1 c
```

English text Output:

a must be true and after one clock tick b must be true for at least 2 and at most 10 clock ticks and after 1 clock tick c must be true

System Verilog Assertion Input:

```
start |-> (req ##2 gnt)
```

English text Output:

whenever start is true, it implies that req must be true at the current clock tick and gnt must be true on second subsequent clock tick.

System Verilog Assertion Input:

```
req ##1 gnt ##1 !req
```

English text Output:

req must be true on the current clock tick, gnt must be true on the first subsequent tick, and req must be false on the next clock tick after that.

System Verilog Assertion Input:

```
req ##[4:$] gnt
```

English text Output:

req must be true at the current clock tick, and signal gnt must be true at any clock tick after the current clock tick

System Verilog Assertion Input:

```
$rose(a) |-> (a throughout b [->1]) ##1 !a
```

English text Output:

Whenever a goes high, a must be high until b is asserted and after 1 clock cycle, a must be low

To assure accurate results, the model has been validated by splitting the dataset into training and test data, 75% of the data is in the training data set of the model and 25% of the data is the test set which is a subset to test the trained model. A knob has been set at 75% which assures the model training until the desired value of accuracy is attained. For the SVA to English assertion translation BLEU score is calculated for the language where the threshold knob has been set to 0.30 currently.

We drew inspiration from standard machine translation trends in the industry. The machine translation (say from English to German) was historically all rules based. After parsing, the phonemes were identified and there was an extensive rules engine to generate the output. All this has been replaced by the ML algorithms these days due to the superior quality of results.

We have been experimenting with various approaches for SVA to English and what we have presented is just one of the several approaches. We have now mentioned all the approaches with a subjective score.

1. Only Rule based
2. ML (RNN with Attention) based
3. Rule based followed by ML

Currently we have found the 3rd approach to be the best due to the quality of results.

The concepts discussed here in this paper will bring the ease for a designer to create an executable specification. It will help them in generating optimal logic design code. Also, system verilog decoder can play a key role in any engineer's life to help them understand and create assertions. The decoder can also help them to verify whether the created assertions are correct or not.

V. REFERENCES

[1] Machine Learning in Python https://gallery.mailchimp.com/dc3a7ef4d750c0abfc19202a3/files/704291d2-365e-45bf-a9f5-719959dfe415/Ng_MLY01.pdf
[2] SystemVerilog UVM http://accellera.org/downloads/standards/uvmconevrt
[3] IDS Documentation https://www.agnisys.com/release/docs/ids/
[4] Comprehensive Register Dictionary https://www.agnisys.com/crd/
[5] Trustworthy specifications of ARM® v8-A and v8-M system level architecture https://ieeexplore.ieee.org/document/7886675/?reload=true
[6] Keras: The Python Deep Learning library https://keras.io/