Machine learning pseudo-natural language for temporal logic requirements of embedded systems.

Gaétan Hains  
LACL  
UPEC  
Créteil, France  
ORCID 0000-0002-1687-8091  
gaetan.hains@u-pec.fr

Ouarda Fenek  
Intellig’IA  
Object’ive  
Paris, France  
ouardafenek@gmail.com

Abstract—Requirements formalization is a critical part of any verification methodology for embedded systems like those in the automotive industry. There is a strong tension between techniques that enter requirements as logic- or code-like formal expressions and others that use natural language. The former are much safer but require user training and have low productivity. As a compromise we proposed a context-free grammar for entering real-time system requirements and translating them to temporal logic (TL) unambiously and reversibly. It has been demonstrated on hundreds of examples and became validated by a recent patent. But building or extending the grammar itself requires a precise understanding of the translation rules. To alleviate this new hurdle we have found that neural nets inspired by NLP can learn and then replace the pseudo-English-to-TL translation, and allow extending it without the explicit use of a grammar. A brief bibliographic survey shows that this technique has been proposed several times before, but that its application to model-checking and reversible nature are novel.

The next sections introduce the grammar-based translation and puts it into context for model-checking of automotive real-time systems. An initial imperfect neural net is then described together with its mix of real-life and synthetic datasets. Finally the better solution is described and shows a high degree of prediction quality.

Index Terms—Requirements formalization, real-time embedded systems, natural language processing, deep learning, software engineering productivity, safety-critical systems.

I. INTRODUCTION

Requirements formalization is a critical part of any verification methodology for safety-critical embedded systems like those in the aerospace [1], train [2] and automotive industry [3]. There is a strong tension between techniques that enter requirements as logic- or code-like formal expressions and others that use natural language. The former are much safer but require user training and have low productivity. The latter are very user friendly but error-prone due to the ambiguous and incomplete nature of natural language. As a compromise we proposed a context-free grammar for entering real-time system requirements and translating them to linear temporal logic (LTL) unambiously and reversibly. This has been demonstrated on hundreds of examples taken from the Kansas State University database of formalized requirements [4] and became validated by a recent patent [5].

But building or extending the grammar itself requires a fine understanding of the translation. To alleviate this new hurdle we have found that neural nets adapted from NLP can learn and then replace the pseudo-English-to-TL translation, and allow extending it without the explicit use of a grammar. A brief bibliographic survey shows that this technique has been proposed several times before, but that its application to model-checking and reversible nature are novel.

The next sections introduce the grammar-based translation and puts it into context for model-checking of automotive real-time systems. An initial imperfect neural net is then described together with its mix of real-life and synthetic datasets. Finally the better solution is described and shows a high degree of prediction quality.

II. BOUNDED LTL AND VERIFICATION

When requirements for safety-critical systems are written in a formal language, they can be used for the automatic verification of interaction models with solvers and model-checkers. This methodology forms an important component of Model-Based Software Engineering because of its mathematical precision and automation. We believe that, Bounded Linear Temporal logic (BLTL) [6] is a convenient formalism for specifying and verifying properties of reactive systems. There are many variants of temporal logic and several ways of defining any given variant. The variant we use BLTL has been extensively applied to requirements formalization and system verification at the VERIMAG laboratory in Grenoble (France) and elsewhere. Its definition is summarized in appendix A.

BLTL is very expressive for writing real-time requirements about safety, liveness, reaction time. Moreover BLTL formulae can be directly used for model-checking of system models (transition systems, petri nets etc) leading to early discovery of system bugs or inadequacies.

Unfortunately, engineers are not trained in formal languages and even trained experts can find the extensive use of temporal logic slow and at times clumsy. As a result, many existing requirements management systems are based on a form of structured text management whose elementary parts are natural-language descriptions of system behavior. This is error prone and can lead to dangerous ambiguities. As a compromise we proposed a context-free grammar for entering real-time system requirements in a form of pseudo-English and translating them to temporal logic (BLTL) unambiously and reversibly.
III. FROM PSEUDO-ENGLISH TO TEMPORAL LOGIC

A BLTL formula such as
G((trigger and F_{120}(bounded)) ⇒ (engineStable U reply ))
is much easier to produce, understand and every modify if written in structured pseudo-English as
(engineStable until reply ) if
(trigger and (ultimately bounded within 120 seconds))

We have defined a context-free grammar called NL2TL to produce such a translation. This invention presented in the context of real-time requirements engineering is protected by patent [5]. Its novelty over similar prior work [7], [8] is that it has been implemented in two forms, extensively tested against real-life examples, and allows for the inverse translation of model-checking error results that come as (traces of) formulas.

A fragment of the grammar is listed in appendixB. There are multiple "NL" (pseudo-natural language) forms that translate to the same "TL" (BLTL) formula. This simplifies the task of the user who may remember one or the other form for the same meaning. For example the following two phrases translate to
((PEDAL == 0.6) U\{900\} false) or (PEDAL != 0.6)
when PEDAL is 0.6 then PEDAL shall be 0.6 until 900 milliseconds
if PEDAL==0.6, then (PEDAL == 0.6) until 900 ms. Parentheses can be used but are not necessary: as in normal English, there is only rarely a syntactic ambiguity. The grammar has an ad-hoc implementation in OCaml whose advantage is to cover all of the possible input forms for a target formula. There is also an Xtext (Eclipse plugin) implementation whose grammar is weaker but which comes with more user support: auto-complete, formula static checks, menus, etc.

In practice it is necessary to be able to extend the grammar: when details of the output logic change, to provide convenient input abbreviations, to adapt to a new output logic etc. Defining new context-free rules is quite straightforward for a software engineer. But defining the new abstract-syntax tree is error prone, and updating the Eclipse-Xtext implementation is much more delicate.

To bridge this serious productivity gap, we propose an AI-based approach i.e. training an NLP neural-net with (english, temporal logic) pairs produced or checked by the grammar, and using it to replace the grammar altogether. When this is reliably realized, extending the translation becomes a matter of producing new pairs and retraining the neural-net. How to change the existing implicit grammar embodied in a neural-net, without manipulating the grammar is an open problem.

In the next sections we describe this application of AI techniques to our requirements engineering problem.

IV. RNN WITH ATTENTION

We had a set of 35 000 [ pseudo-English, BLTL ] pairs that includes real-life examples taken from the Kansas State University database of requirement patterns [4]. Other examples were generated with Data Augmentation techniques. This dataset has been used for neural-net training. We also had 7000 pairs for validation of the model and finally 15000 for testing its quality. The model we used was a recurrent-neural-net (RNN) for sequence-to-sequence translation with an attention mechanism (figure 4). The trained model had a 17% accuracy and 70% BLEU score. It could predict correctly most of the syntax tree’s internal nodes but was completely incorrect on the terminal nodes. For example it translated
globally idle=previousValue until 97 seconds
into
G\{12\} (idle == idle) and not
G\{97\} (idle == previousValue) as it should have. This unexpected behavior was systematic so we searched for a way to improve, but not replace the model. Indeed a second, failed approach used an Encoder-Decoder technique inspired from natural language. But its predictions were even worse than the first one. The final solution was an improvement of the first.

V. HYBRID SOLUTION

Our best model was a hybrid between a sequence-to-sequence architecture, to learn the syntactic structure and a named entity recog-
Fig. 4. RNN with attention mechanism

Fig. 5. Syntax terminals

in figure 4. We used 44 000 [ pseudo-English, BLTL ] pairs to train it and 19 000 to test it. The final test results gave 99.99% accuracy and 100.0% recall which we considered to be a complete success. Example translations are shown in figure 7. As a final notice the translation tools are complete by design: the initial grammar because it was built from BLTL syntax, and the final neural-net because it was trained with a very large dataset.

Appendix C gives more examples of the possible translations.

VI. CONCLUSIONS

We have described an application of AI-techniques, namely NLP-inspired neural-nets, to the safety-critical problem of real-time requirements formalization. A classic solution using context-free grammars, itself based on a substantial set of real-life examples served as our first stage. Having high confidence in this first solution but finding it unproductive for users, we searched for flexible and efficient machine-learning models and succeeded after three attempts. Among the lessons learned were the need for a hybrid model, the preference for an “explainable” but imperfect first (RNN) solution and the main reason for this effort: to make the translation tool easily upgraded.

Ongoing work has been considering the application of this technique to requirements classification. Future work will investigate the possibility of modifying (for example removal of existing rules) the implicit grammar once it has been encoded into a machine-learning model. A more ambitious goal could be the direct learning of a NL-to-TL translation from examples given by experts in formal verification.

Other projects could see the application of such highly user-friendly translation techniques to the manipulation of formal system models such as transition systems or Petri nets, and for applying system changes semi-automatically from model-checking results.

ACKNOWLEDGMENT

The first author thanks Saddek Bensalem and Ayoub Nouri for introducing him to current research on requirements formalization.

REFERENCES


APPENDIX A: BOUNDED LINEAR TEMPORAL LOGIC

Bounded Linear Temporal logic (BLTL) [6] is defined as follows. Assume t is a positive integer representing a global time point in computation. Assume given a set of atomic propositions P and p ∈ P. In practice p can either be an formula constant like \( P_{\text{SYSTEM_ACTIVE}} \) or an atomic formula built from state variables like \( \text{ACTIVATION} = \text{ON} \) or \( \text{DELAY} < 10 \). The formulas of bounded linear time logic (BLTL) are defined by induction as follows: T (true), F (false), p and (not p) are all formulae. If f1, f2 are formulae then (f1 and f2), (f1 or f2), (f1 if f2) and (f1 if and f2) are also formulae. If f1, f2 are formulae then (X f1) and (U f1) and (G f1) are also formulae. Operator X is called next, operator U is called until and operator G is called globally within delay t. Other temporal operators can be defined from the above: for example F(t) (finally, ultimately within delay t) and G(t) (globally within delay t) that are both unary i.e. qualifiers for a single formula. When the time parameter t is absent, the meaning is that it is infinite.

Informal meaning of a temporal formula:

A computation satisfies (X f1) if its state next after the initial state satisfies f1. A computation satisfies (U f1) if there exists a state st2 that satisfied f2 and such that all the previous states satisfy f1 and such that the delay until st2 is no more than t time steps. The unit of time delays t depends on context and can be either: the number of system transitions (steps), milliseconds or seconds.

APPENDIX B: NL2TL PSEUDO-ENGLISH TO BLTL

A fragment of the context-free grammar NL2TL is given below. Parts between braces are the abstract syntax tree constructors in Yacc style.

```
// |::= term
term ::= "term". (Term_name $1)
// plus formula
/\ formula
// conjunction formula
// disjunction formula
// implication formula
// negation formula
// formula
// formula
// formula
// formula
// formula
// formula
// formula
// formula
// formula
// formula
// formula
// formula
```

APPENDIX C: [ PSEUDO-ENGLISH, BLTL ] EXAMPLES

Here we give more examples of the translations that NL2TL or its neural-net encoding can produce.

Pseudo-English:

```
DOORS_CLOSED shall never ultimately cause UPDATE
DOORS is not RUNNABLE or GROSS_EXECUTION_TIME is no less than PREVIOUS_VALUE shall never cause REACTION
```

BLTL:

```
G(\neg \neg (((ECU_TIMING<CLOCK) \& DOORS(ENGINE)) \rightarrow F(UPDATE))))
```

Pseudo-English:

```
whenever UNKNOWN is DOORS_CLOSED or SYSTEM_TIMING is no less than MASTER_CLOCK
```

BLTL:

```
G((\neg DOORS_CLOSED \& (SYSTEM_TIMING\ge MASTER_CLOCK)) \rightarrow F(Update))
```

APPENDIX A: BOUNDED LINEAR TEMPORAL LOGIC

Bounded Linear Temporal logic (BLTL) [6] is defined as follows. Assume t is a positive integer representing a global time point in computation. Assume given a set of atomic propositions P and p ∈ P. In practice p can either be an formula constant like \( P_{\text{SYSTEM_ACTIVE}} \) or an atomic formula built from state variables like \( \text{ACTIVATION} = \text{ON} \) or \( \text{DELAY} < 10 \). The formulas of bounded linear time logic (BLTL) are defined by induction as follows: T (true), F (false), p and (not p) are all formulae. If f1, f2 are formulae then (f1 and f2), (f1 or f2), (f1 if f2) and (f1 if and f2) are also formulae. If f1, f2 are formulae then (X f1) and (U f1) and (G f1) are also formulae. Operator X is called next, operator U is called until and operator G is called globally within delay t. Other temporal operators can be defined from the above: for example F(t) (finally, ultimately within delay t) and G(t) (globally within delay t) that are both unary i.e. qualifiers for a single formula. When the time parameter t is absent, the meaning is that it is infinite.

Informal meaning of a temporal formula:

A computation satisfies (X f1) if its state next after the initial state satisfies f1. A computation satisfies (U f1) if there exists a state st2 that satisfied f2 and such that all the previous states satisfy f1 and such that the delay until st2 is no more than t time steps. The unit of time delays t depends on context and can be either: the number of system transitions (steps), milliseconds or seconds.

APPENDIX B: NL2TL PSEUDO-ENGLISH TO BLTL

A fragment of the context-free grammar NL2TL is given below. Parts between braces are the abstract syntax tree constructors in Yacc style.

```
// |::= term
term ::= "term". (Term_name $1)
// plus formula
/\ formula
// conjunction formula
// disjunction formula
// implication formula
// negation formula
// formula
// formula
// formula
// formula
// formula
// formula
// formula
// formula
// formula
// formula
// formula
// formula
// formula
```

APPENDIX C: [ PSEUDO-ENGLISH, BLTL ] EXAMPLES

Here we give more examples of the translations that NL2TL or its neural-net encoding can produce.

Pseudo-English:

```
COMPLETE_GEAR_CHANGE shall always ultimately hold within 610ms
```

BLTL:

```
G(((DOORS_CLOSED(UNKNOWN) || (SYSTEM_TIMING\ge MASTER_CLOCK)) \rightarrow F(8\text{steps}, REACTION)))
```

Pseudo-English:

```
DOORS is not RUNNABLE or GROSS_EXECUTION_TIME is no less than PREVIOUS_VALUE
```

BLTL:

```
\neg (((ECU_TIMING<CLOCK) \& DOORS(ENGINE)) \rightarrow F(UPDATE)))
```

Pseudo-English:

```
DOORS_CLOSED shall always ultimately cause REACTION within 8\text{steps}
DOORS_CLOSED shall never cause REACTION within 8\text{steps}
DOORS_CLOSED shall always ultimately cause REACTION within 8\text{steps}
```

BLTL:

```
(((DOORS_CLOSED || (SYSTEM_TIMING\ge MASTER_CLOCK)) \rightarrow F(8\text{steps}, REACTION)))
```

Pseudo-English:

```
DOORS_CLOSED shall never cause REACTION
```

BLTL:

```
\neg (((ECU_TIMING<CLOCK) \& DOORS(ENGINE)) \rightarrow F(UPDATE)))
```

Pseudo-English:

```
DOORS (is true) and DOORS (does not hold).
```

BLTL:

```
\neg (((ECU_TIMING<CLOCK) \& DOORS(ENGINE)) \rightarrow F(UPDATE)))
```

Pseudo-English:

```
DOORS_CLOSED is not TRUE or CET is no more than GROSS_EXECUTION_TIME
```

BLTL:

```
\neg ((\neg DOORS_CLOSED \& (CET\le GROSS_EXECUTION_TIME)) \rightarrow F(8\text{steps}, REACTION)))
```

Pseudo-English:

```
DOORS_CLOSED is not TRUE or CET is no more than GROSS_EXECUTION_TIME
```

BLTL:

```
\neg ((\neg DOORS_CLOSED \& (CET\le GROSS_EXECUTION_TIME)) \rightarrow F(8\text{steps}, REACTION)))
```

Pseudo-English:

```
DOORS is not RUNNABLE or GROSS_EXECUTION_TIME is no less than PREVIOUS_VALUE shall never cause REACTION
```

BLTL:

```
G(\neg \neg (((ECU_TIMING<CLOCK) \& DOORS(ENGINE)) \rightarrow F(UPDATE))))
```

Pseudo-English:

```
while LENGTH is less than SYSTEM_TIMING
```

BLTL:

```
G((\neg DOORS_CLOSED || (CET\le GROSS_EXECUTION_TIME)) \rightarrow F(8\text{steps}, REACTION)))
```

Pseudo-English:

```
DOORS is not RUNNABLE or GROSS EXECUTION TIME is no less than PREVIOUS VALUE shall never cause REACTION
```

BLTL:

```
G(\neg \neg (((ECU_TIMING<CLOCK) \& DOORS(ENGINE)) \rightarrow F(UPDATE))))
```

Pseudo-English:

```
while LENGTH is less than SYSTEM_TIMING
```

BLTL:

```
G((\neg DOORS_CLOSED || (CET\le GROSS_EXECUTION_TIME)) \rightarrow F(8\text{steps}, REACTION)))
```

Pseudo-English:

```
DOORS is not RUNNABLE or GROSS EXECUTION TIME is no less than PREVIOUS VALUE shall never cause REACTION
```

BLTL:

```
G(\neg \neg (((ECU_TIMING<CLOCK) \& DOORS(ENGINE)) \rightarrow F(UPDATE))))
```

Pseudo-English:

```
DOORS is not RUNNABLE or GROSS EXECUTION TIME is no less than PREVIOUS VALUE shall never cause REACTION
```

BLTL:

```
G(\neg \neg (((ECU_TIMING<CLOCK) \& DOORS(ENGINE)) \rightarrow F(UPDATE))))
```

Pseudo-English:

```
DOORS is not RUNNABLE or GROSS EXECUTION TIME is no less than PREVIOUS VALUE shall never cause REACTION
```

BLTL:

```
G(\neg \neg (((ECU_TIMING<CLOCK) \& DOORS(ENGINE)) \rightarrow F(UPDATE))))
```

Pseudo-English:

```
DOORS is not RUNNABLE or GROSS EXECUTION TIME is no less than PREVIOUS VALUE shall never cause REACTION
```

BLTL:

```
G(\neg \neg (((ECU_TIMING<CLOCK) \& DOORS(ENGINE)) \rightarrow F(UPDATE))))
```

Pseudo-English:

```
DOORS is not RUNNABLE or GROSS EXECUTION TIME is no less than PREVIOUS VALUE shall never cause REACTION
```

BLTL:

```
G(\neg \neg (((ECU_TIMING<CLOCK) \& DOORS(ENGINE)) \rightarrow F(UPDATE))))
```

Pseudo-English:

```
DOORS is not RUNNABLE or GROSS EXECUTION TIME is no less than PREVIOUS VALUE shall never cause REACTION
```

BLTL:

```
G(\neg \neg (((ECU_TIMING<CLOCK) \& DOORS(ENGINE)) \rightarrow F(UPDATE))))
```