Leveraging Natural Language Processing for a Consistency Checking Toolchain of Automotive Requirements

Vincent Bertram
Software Engineering
RWTH Aachen University
Aachen, Germany
bertram@se-rwth.de

Hendrik Kausch
Software Engineering
RWTH Aachen University
Aachen, Germany
kausch@se-rwth.de

Evgeny Kusmenko
Software Engineering
RWTH Aachen University
Aachen, Germany
kusmenko@se-rwth.de

Haron Nqiri
Software Engineering
RWTH Aachen University
Aachen, Germany
nqiri@se-rwth.de

Bernhard Rumpe
Software Engineering
RWTH Aachen University
Aachen, Germany
rumpe@se-rwth.de

Constantin Venhoff
Software Engineering
RWTH Aachen University
Aachen, Germany
constantin.venhoff@rwth-aachen.de

Abstract—**In the automotive industry, specifications often consist of a large number of textual requirements. These requirements are linguistically ambiguous and written in informal language. Utilizing Structured English for requirements eliminates ambiguity, improves data quality, and supports further automated processing while maintaining readability. The recent development of large language models enables a fully automated translation approach using few-shot learning. To deal with the limited context size of large language models, an improved algorithm, OptKATE, is presented to find an ideal set of requirements for few-shot learning. Structured English can be used as a basis for further formalization. This capability is key in creating an interface between natural language processing and verification, in our case, consistency analysis using the Z3 SMT solver. We implemented a grammar for translating Structured English into TCTL using the MontiCore workbench. Furthermore, since SMT-based methods currently rely on manual precondition satisfaction and do not tackle conflicting preconditions automatically, we propose a scenario generation algorithm that generates potential scenarios using the specification and checks the requirements against them. Through this approach, we can better identify and resolve conflicting preconditions, ultimately improving the consistency of requirements. Our toolchain is evaluated using an automotive requirements dataset provided by former Daimler AG.**

Index Terms—**Classification, Natural Language Processing (NLP), Neural Networks, Requirements Engineering**

I. Introduction

In automotive engineering, specifications, in part, consist of textual requirements that are linguistically ambiguous and written in an informal language. This is especially impactful, as most product errors are traced back to faulty specifications [1] and can harm people, the environment or property. It is important to find such errors as early as possible, but large specification documents make the manual search for ambiguous language and inconsistencies difficult.

New specifications are often not written down completely as a single textual requirement, but rather existing requirements are reformulated or supplemented with new information. In addition, not all requirements are written down one after the other, but this happens in stages and sometimes only after approval by other stakeholders. This leads to the fact that several instances are involved in defining and finalizing a requirement. This procedure makes ensuring the consistency of requirements a great challenge. To address this problem, on the one hand, automatic analysis of requirements is preferred over difficult and error-prone manual analysis. On the other hand, for automatic analyses of natural language requirements, ambiguity must be removed. Using Structured English [2] [3] for requirements removes this ambiguity and builds the foundation for further verification while maintaining readability. It has been used successfully in automotive case studies [4] [5]. The translation process from informal textual requirements to a pattern language is a non-trivial task, it is usually supported by software but is performed manually by the relevant requirements engineer [6]. Recent development of Large Language Models (LLMs) enables an automated transformation approach. We use a LLM for pattern classification and translation into Structured English.

II. Preliminaries

A. Specification Patterns

Specification patterns were originally proposed by Dwyer et al. [7] to model the behavior of systems using logic formula templates. These templates used Linear Temporal Logic (LTL) or Computation Tree Logic (CTL) and the user could pack propositional formulas into the template to instantiate the semantics of the pattern. Each pattern also has a description of its purpose in natural language to help the user select the correct patterns. The patterns were developed by analyzing 555 requirement specifications from more than 35 different sources to find the most frequent language structures that occur. However, the patterns were still hard to use for people without a background in formal methods and also did not support real-time constraints.

Konrad and Cheng [3] proposed a real-time extension of specification patterns and also proposed a grammar that defined a General Purpose Language (GPL) called *Structured English* that allowed for a natural language representation of requirements, which can be mapped 1:1 to a logic formula. Thus, the requirements are readable for requirement engineers who are not trained using formal methods, but are still easy to express in formal logic. Since LTL and CTL do not support real-time expressions, real-time extensions of these logics were used, in this case Timed Computation Tree Logic (TCTL) [8].

Each requirement in Structured English is made up of a scope and a pattern. The scope is used to specify when the properties of a requirement have to hold and the pattern expresses the actual properties of the requirement. The properties are expressed with propositional formulas.

The consistency of requirements is checked in the logical sense. A set of requirements is consistent if and only if there are no contradictions. This also includes indirect contradictions that might only surface when checking multiple requirements together. To verify consistency, logical formulas representing requirements are used.

B. Timed Computational Tree Logic

As TCTL is a real-time extension of CTL, that models properties as a temporal tree structure. Thus, the future has multiple possible paths, and it is not specified which path will be realized. The temporal operators in TCTL always occur as pairs of operators, where the first operator shows whether the property holds on all the following paths or if only one path must exist where the property holds. The second operator expresses whether the property must hold all the time, occur at some point, or hold until another property holds. The last three operators are timed in TCTL, thus expressing that some property must hold all the time within an interval, must occur within an interval, or must hold until another property holds. This allows the modeling of very complex real-timed systems.

C. Satisfiability Modulo Theories Library

The Satisfiability Modulo Theories Library (SMT-LIB) [9] is an approach to provide a common input and output language for Satisfiability Modulo Theories (SMT) solvers. SMT are an extension of the Boolean Satisfiability Problem (SAT), which interprets some symbols with a background theory (for example arithmetic operators like "+" have a fixed meaning). SMT-LIB allows for more complex formulas than in SAT since it allows for real numbers, integers, and various data structures such as lists, arrays, bit vectors, and strings. The SMT-LIB scripts consist of *assertions* that represent formulas and *declarations* that introduce variables and functions. The variable and function parameters have a type and can also be declared as constant. SMT solvers are tools that aim to find a model for a set of SMT-LIB assertions and declarations. Since the SAT problem is already NP-hard, the same holds for the SMT problem. We used SMT-LIB version 2, as this is the default input of the Microsoft Z3 SMT solver [10] that is used in the proposed toolchain.

D. Large Language Models

LLMs are deep learning models that are trained on language data. Training is carried out with the target that the model learns to predict the most likely next token, based on an input stream [11] [12]. A token is a chunk of text that represents a single unit of meaning. However, this training does not teach the model how to solve specific tasks. Traditionally, these models are used as a basis for fine-tuning, hence using the weights of the pre-trained model as a basis and then training the model on task-specific data [13].

Recent LLMs, such as the GPT-3 model [11], have reached a point where basic training on language data suffices to execute different NLP tasks with the model out of the box. This is possible because of a large increase in the size of the model and the amount of data the model was trained on. To allow these models to execute an NLP task, a *context* is given to the model. The context can contain a *task description* that is used to describe the task and some examples of the task the model should perform. The *test prompt* is the entity on which the model should perform the task and is therefore placed at the end of the context, leaving out the task solution. The model then predicts, based on the context, the next most likely token. If the model was trained on a sufficient amount of data, it will predict the solution to the task. This procedure is called *Few-Shot-Learning* [14].

The model we used in this paper is the GPT-J-6B language model with 6 billion parameters [15]. The model is freely available and was trained on the Pile dataset [16], an 825 GB open source language modeling dataset consisting of 22 smaller high-quality datasets combined together. Evaluations have shown that the performance of GPT-J-6B is comparable to the GPT-3 version with 6.7 billion parameters [15].

III. Methodology

We propose an end-to-end toolchain to perform automated consistency checking of the behavior of an automotive system, specified by its requirement specification. The methodology for this toolchain is illustrated in Figure 1 and consists of three main components. The first component "*Transformation to Structured English*" transforms the natural language specification into a Structured English specification. For this step, we introduce a new formalization method, using large language models. We used Google Colaboratory (Colab) [17] to carry out our experiments. The GPT-J-6B [15] model was used since the original paper "Language Models are Few-Shot Learners" [11] suggests that the few-shot learning capabilities of LLMs increase with the size of the model. The translation technique leverages the few-shot learning capabilities of large language models. On top of this, the translation was split into three

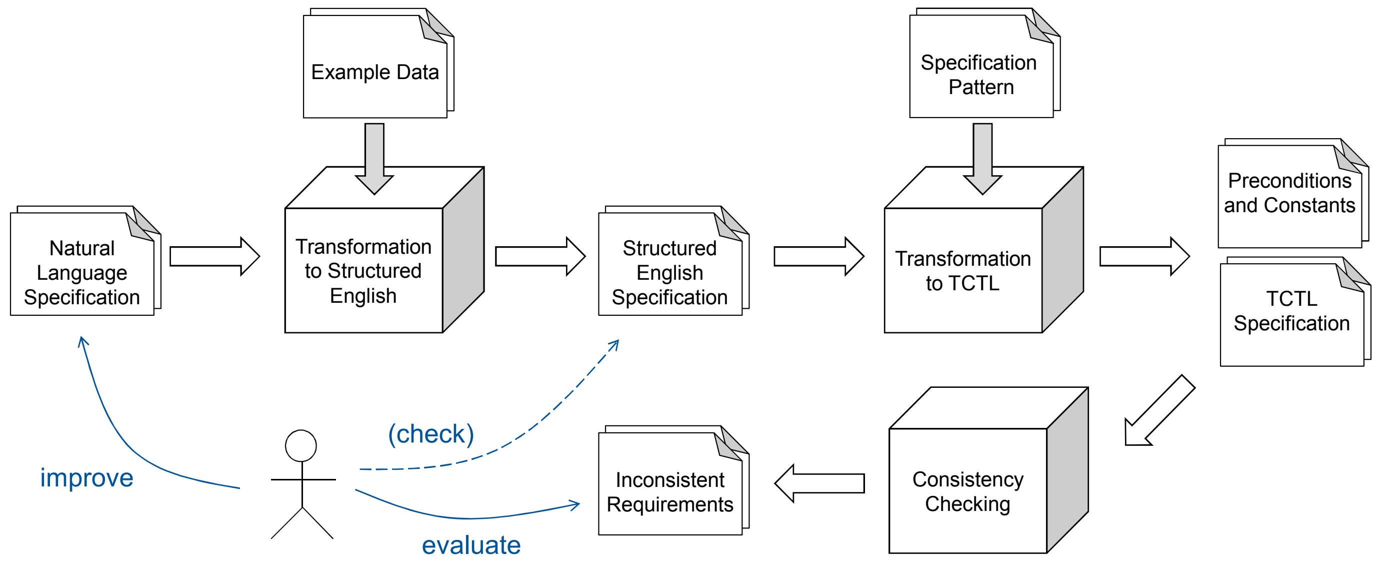

Example Data
Specification Pattern
Natural Language Specification
Transformation to Structured English
Structured English Specification
Transformation to TCTL
Preconditions and Constants
TCTL Specification
improve
(check)
evaluate
Inconsistent Requirements
Consistency Checking

Fig. 1. Overview of Methodology

steps that consecutively transform a natural language requirement into Structured English. Therefore, this transformation needs some example data (few-shot data) consisting of natural language requirements and the individual transformation steps. The second component ”*Transformation to TCTL*” encodes the obtained Structured English specification in TCTL formulas and extracts the requirements preconditions and constants. Preconditions are the left part of each implication that is a part of a requirement of the specification. The constants are extracted by storing which propositions occur on the right and which on the left side of a binary operator. If a proposition is only on the right side, it is only assigned and never changes its value; thus, it is a constant. The TCTL formulas are obtained by applying the mapping from Structured English to TCTL, which is derived from the specification pattern [18]. The third component ”*Consistency Checking*” checks the consistency of the logical formulas by generating scenarios from preconditions. Scenarios are sets of formulas that are generated from the specification. Each scenario represents situations that could occur according to the specification, also encoded in TCTL logic. The TCTL specification is then checked against each scenario. To do so, the current scenario and the TCTL specification are transformed into a SMT script that can be checked for inconsistencies by a solver. Therefore, the consistency checking procedure finds and returns the inconsistencies. In an iterative process, the user corrects the inconsistencies in the original natural language specification or the Structured English specification and then applies consistency checking again to check if all inconsistencies are eliminated.

IV. Transformation to Structured English

The transformation of natural language requirements into Structured English is done by first determining the target pattern for the requirement using a few-shot classifier and second by providing a context of transformation examples formalized in the target pattern for a LLM, in this case, GPT-J-6B [15]. The language model then performs a transformation with few-shot learning. The context has to contain example transformations that support the model to derive the transformation steps necessary to transform the current natural language requirement into Structured English. The classification is performed because in early experiments we have seen that restricting the context to the target pattern of the test prompt significantly improves the transformation quality. The context selection procedure is based on the finding that a low distance between the test prompt, in this case, the requirement to be transformed, and the examples result in a much better few-shot learning performance [19] compared to a random context selection. Since requirements with similar semantics should get mapped to similar Structured English representations, a context selection based on the distance between the natural language requirements is used.

The *Knn-Augmented in-conText Example selection (KATE)* algorithm [19] performs context selection, using a distance function $\mu_\Theta(\cdot)$. In our case, we used semantic distance functions, because semantically similar natural language requirements should be formalized into equivalent Structured English representations. The distance functions used are presented in Section VII. We use KATE to select semantically similar requirements and their patterns as labels as a few-shot context for the classification. However, this approach does not work for our Structured English transformation since the *KATE* algorithm does not take the limited context size of language models into account and thus the size of the examples has to be considered. In other words, some requirements can be semantically similar to a test prompt, but their length prevents multiple other, shorter, examples from occurring in the context that might also be semantically similar to the test prompt. We decided that in such a situation multiple shorter requirements should be chosen for the context. To ensure this, the results of the original algorithm are optimized in terms of length using the knapsack algorithm, interpreting the length of the requirements as weight and the distance to the test prompt as profit (a smaller distance is more profitable). The knapsack algorithm calculates the optimal selection of examples, such that the profit is maximal, and hence the overall distance of the examples in the context is minimal. This selection procedure we called *OptKATE* and the algorithm according is given in Algorithm 1. The selected examples build the context for transforming the test prompt.

Algorithm 1 OptKATE Algorithm

```
Require: $x_{test}$: Test prompt, $D_t$: Training set, $\mu_\Theta(\cdot)$: Distance function, $y_c$: Capacity of Language Model
 1: function OPTKATE($x_{test}, D_t, \mu_\Theta(\cdot), y_c$)
 2:     $D_{dist} \leftarrow []$
 3:     for $x_{example}$ in $D_t$ do
 4:         $d \leftarrow \mu_\Theta(x_{test}, x_{example})$
 5:         add $(x_{example}, d)$ to $D_{dist}$
 6:     end for
 7:     $y_v \leftarrow [y$ for $(x, y)$ in $D_{dist}]$
 8:     $y_w \leftarrow [$len(tokenizer($x$)) for $x$ in $D_t]$
 9:     $D_{opt} \leftarrow$ knapsack($y_v, y_w, y_c$)
10:     return $D_{opt}$
11: end function
```

The last step in the transformation process is to pass the context, together with the test prompt, to the language model. The model will generate a solution for the transformation task. Mistakes in the translation process are possible; hence, a manual check and needed error corrections are performed.

V. Transformation to TCTL

Despite the existing tool support for processing Structured English [5] [18], a new tool was implemented, since the existing tools do not have functionalities such as extracting preconditions and constants, which are needed by our proposed toolchain.

The Language Workbench MontiCore [20] was used to implement Structured English. MontiCore is a framework that generates Java code from formal grammars, which provides the functionality to process the language defined by the grammar. Since Structured English is defined by a formal grammar [18] the definition was straightforward.

An excerpt of the grammar used in the MontiCore implementation is shown in Figure 2. The grammar consists of production rules, where the left side is a non-terminal and the right side is a mix of terminals and non-terminals. The starting non-terminal is *Specification*. Each *Specification* consists of arbitrary many properties. Each property consists of a scope and a pattern, which together resemble a Structured English pattern. MontiCore generates for each non-terminal on the left side of a production rule a Java class. MontiCore also generates a parser component that can parse a Structured English specification. The parser is used to determine the

```
scope Specification = (Property)*;

Property = Scope ", " Pattern ".";


interface Scope;

Globally implements Scope = "Globally";

Before implements Scope = "Before " R:Formula;

...


interface Pattern;

Universality implements Pattern =
  "it is always the case that " P:Formula (" holds")? (" "
  time:Time)?;

Response implements Pattern =
  "if " P:Formula (" has occurred")?  ", then in response "
  S:Formula (" eventually holds")? (" " time:Time)?;

...
```

Fig. 2. MontiCore Grammar: Scope and Pattern

scope and pattern used to formulate the Structured English requirements. Based on this information, a logic formula template is selected according to the specification patterns.

```
token Formula = AtomicFormula+;

fragment token AtomicFormula = ("not ")?  (Lit Operand Lit
(Connector)?);

…



token Time = UpperTimeBound | LowerTimeBound | Interval |
TimeIndication;

fragment token UpperTimeBound = "within " TTU;

fragment token LowerTimeBound = "after " TTU;

fragment token Interval = "between " T " and " TTU;

…
```

Fig. 3. MontiCore Grammar: Formulas and Time Bounds

The logic formula template needs to be instantiated with the propositional formulas and time constraints used in the Structured English requirement. To make this information accessible, each production rule for the *Formula* and *Time* non-terminal is marked as a token in the grammar, as shown in Figure 3. This tells the parser that once recognized during the parsing procedure, the formulas and time constraints should be stored. To do so, classes for the tokens are also generated and can be accessed through the respective pattern and scope objects. After extracting the formulas and time constraints from the recognized scope and pattern, the formula template is instantiated, and the resulting formula is added to the output.

However, as described in Section III an extraction of preconditions and constants is needed. To do so, a routine was implemented that stores the preconditions for each pattern and scope after parsing the specification. For example, if a requirement has the scope "*After Q, ...*", the formula Q is stored as a precondition. These preconditions are then transformed into TCTL formulas by instantiating a universality pattern with the propositional formulas in the preconditions. The resulting formulas are annotated with their original requirement and then added to the output. Constants are extracted by building up two sets during the mapping described previously TCTL. One set stores all the propositions occurring on the left side of binary operators. The other set stores all propositions occurring on the right side of binary operators. If a proposition only occurs in the set that stores the right-hand propositions, it is interpreted as a constant and assigned a unique value. This is needed because otherwise the solver could treat propositions as equivalent, by assigning them the same value (for example, ratios like "1:1" and "1:2" should not be treated as equal). The constants are added to the output as a list of propositions.

VI. Consistency Checking

A. SMT-Based Consistency Checking

The consistency checking procedure described in this section is based on the method introduced by Predrag et al [21]. This approach takes a TCTL specification and maps it to first-order logic. The resulting formulas are then encoded in assertions and declarations in a SMT-LIB script, which can be checked for satisfiability by a solver. The approach has low modeling costs and there exist various solvers, such as the Z3 SMT solver [10], CVC5 [22], or SMTInterpol [23], that can take a SMT-LIB script and return a model, or in case of inconsistencies, return an unsatisfiability core containing the inconsistency. In our case, the Z3 SMT solver was used.

The original paper [24] shows that in the SMT-Based approach, the solver might not terminate with a quantifier depth greater than three in the assertions. A migration strategy was proposed that abstracts specification patterns to weakened variants with fewer quantifiers that still ensure a correct consistency checking procedure. In detail, they abstracted requirements formalized with the universality pattern and an "After... Until..." scope by replacing the "Weak Until" operator in the TCTL formula with an until operator that had a fixed time constraint. With this abstraction, the quantifier depth changed from three to two, and termination was ensured. However, this migration strategy works only for this specific pattern and scope.

The migration strategy does not work, for example, for a response pattern with an "After... Until..." scope since the response pattern adds an additional quantifier. Thus a total of four nested quantifiers would be used, and now two of them would have to be omitted to ensure the termination of the solver. However, this is not possible without significantly weakening the semantics of the formula. Thus, when performing the consistency checking procedure with specifications containing such patterns, it is not always possible to formally prove the consistency by finding a model. However, with our approach, the solver was still able to find inconsistencies in a specification, which makes the approach valuable for checking a specification for inconsistencies, but the user has to keep in mind that not finding inconsistencies does not mean that the specification is consistent.

The main issue of the original approach is the handling of implications, as a solver can obtain a trivial satisfiability by never evaluating preconditions as true. A user can systematically satisfy preconditions that do not contradict each other. However, this procedure can be time-consuming, and therefore an automation strategy is proposed in this paper using scenarios.

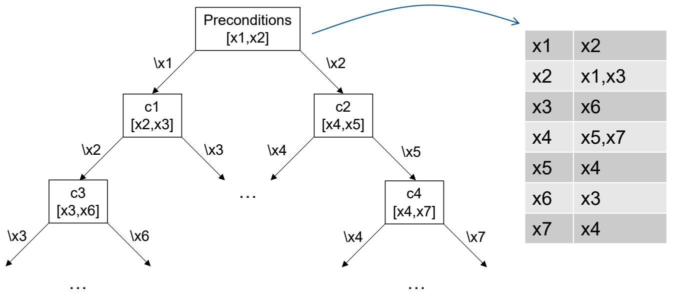
B. PROPAS Adjustments

The PROperty PAttern Specification and Analysis Tool (PROPAS) can perform the transformation from TCTL into a SMT-LIB script and was presented in the original paper by Predrag et al. [24]. However, we have added some required features to the tool, for example, floating point numbers for time constraints, since the Daimler dataset [25], [26] contains such time constraints. Another important adjustment was the conversion of the interval notation for time constraints, since in PROPAS only the "$<$" operator was applicable to temporal operators. However, the notation used by the Structured English grammar is based on intervals. According to [2], the interval notation "$op[t1, t2]...$", where op is an arbitrary temporal operator, can be expressed as "$op = t1(op <= t2 - t1...)$". Thus, the MontiCore implementation was adopted to find the intervals in the formulas and replace them using the notation described above. As mentioned already, only the "$<$" operator was implemented for temporal operators; thus,the operators "$<=$" and "$=$" had to be implemented by us to support the notation of time constraints.

Lastly, the preconditions and constants had to be treated separately. The PROPAS tool defines each variable as a function that takes a real number and returns a real number. This was done because the variables should take a time variable that is used to express the time constraints in the SMT-LIB script. However, a constant does not change over time, and thus the function declarations for constants were changed so that they do not take any arguments. Then assertions were added to the script, where each of these functions was assigned a unique value, to prevent constants from being assigned equal values by the solver. The preconditions were transformed into normal assertions. However, each precondition is marked with a name that declares from which requirement the precondition originates.

C. Scenario-Generation

A scenario is a maximal consistent subset of a potentially inconsistent set of preconditions. Therefore, adding any precondition from the set to a scenario where it is not already included would result in inconsistency in the scenario. In order to find all scenarios, one would need to find all inconsistencies between the preconditions to make sure that no contradicting preconditions are in the same scenario. A first naive approach to find all such inconsistencies is illustrated in Figure 4. The naive search to find the scenarios takes advantage of the fact that no pattern has nested preconditions. Hence, no precondition contains another precondition. This fact can be obtained by analyzing the available patterns of Structured English [18]. Thus, the only kind of inconsistency that can be found is a direct contradiction between two preconditions. Therefore, when the set of preconditions is checked for inconsistencies, either two preconditions that contradict each other are returned, or no inconsistency is returned. When contradicting preconditions are found, these inconsistencies are unpacked as x_1 and x_2. Two subsets of the preconditions are created where one subset omits x_1 and the other omits x_2. Then, the contradicting preconditions are saved in an inconsistency map. Since x_1 and x_2 can both be part of more inconsistencies, both subsets have to be checked recursively. In the worst case, there are as many recursions as preconditions. There would be 2^n computation paths for n preconditions. Therefore, the search must be optimized.



x1	x2
x2	x1,x3
x3	x6
x4	x5,x7
x5	x4
x6	x3
x7	x4

Fig. 4. Naive Inconsistency Search

The problem of finding the maximum consistent subsets was investigated by Robert Malouf [27]. It was shown that the problem of finding these subsets can be reduced to the hitting subset problem, which is NP-Complete. In the work of Malouf, it is already mentioned that the search for maximal consistent subsets can be optimized by heuristics if suitable information is available. This is a promising approach since the preconditions have a predictable form, and thus certain information about the likelihood of inconsistencies found with a precondition can be obtained. When an inconsistency is found between two preconditions x_1 and x_2 then two subsets C_1 and C_2 are generated, where each subset includes one of the inconsistencies (x_1 is in C_1 and x_2 is in C_2). The decision to be made is which of the inconsistent preconditions is more likely to be part of another inconsistency since, apart from the inconsistent preconditions, the subsets are identical. An inconsistency between two preconditions can only occur if there are some shared propositions in the preconditions. Furthermore, if there are many propositions of x_1 in the subset C_1 the likelihood of another inconsistency including x_1 is high. The number of potential inconsistencies that include x_1 is limited by the number of occurrences of the propositions of x_1 in the subset C_1 (without x_1 itself). The same holds for x_2 and C_2. Thus, the heuristic to estimate how many inconsistencies are found with the combination of inconsistent preconditions and the subset to which it is added is the number of occurrences of the propositions of x_i in C_i for $i \in \{1, 2\}$. The algorithm to calculate this number is shown in the algorithm 2.

Algorithm 2 Inconsistency Heuristic

```
Require: $C$: Current branch, $x$: Inconsistent precondition
 1: function H($C, x$)
 2:     $C_{\tilde{x}} \leftarrow$ remove $x$ from $C$
 3:     $prop_x \leftarrow$ get_propositions($x$)
 4:     $prop_C \leftarrow$ get_propositions($C_{\tilde{x}}$)
 5:     $n \leftarrow 0$
 6:     for $p_x$ in $prop_x$ do
 7:         $n \leftarrow n + \sum_{y \in prop_C} [p_x == y]$
 8:     end for
 9:     return $n$
10: end function
```

This heuristic can now be used as a profit estimate of the computation branches. The informed search is implemented using the A*-search algorithm [28], but instead of a cost function, the profit function h is used to estimate how many inconsistencies can be found with the inconsistent precondition and the subset that includes it. Thus, when contradicting preconditions are found, the A*-search algorithm decides in which subset to be searched for further contradictions, so that the most inconsistencies between the inconsistent preconditions are found. The algorithm guarantees that if a solution exists, it will always be found. Since at least the subset containing zero preconditions will be consistent, a solution will always be found. Another important property is that the solution is guaranteed to be optimal, thus no computation path yielding a higher profit can be found if the heuristic is *admissible*, i.e., the cost of a path to the goal is never over-estimated. Thus, in the profit case, the heuristic should never underestimate the profit of the computation path to the goal. Since the heuristic estimates the maximum possible number of inconsistencies found with the inconsistent preconditions and the subset, it is admissible, and thus the A*-search will return an optimal solution with respect to the number of possible inconsistencies. The A*-search algorithm also needs a function g that returns the profit from the root node to the current node. This is simply the depth of the tree at the current node since, at every node in the path, an inconsistency is found. Thus, the profit from the root to the current node is the number of inconsistencies found on the computation path and can be expressed by the depth of the current node. The algorithm to calculate the depth is shown in Algorithm 3.

Algorithm 3 Path Inconsistencies

```
Require: $D_{pre}$: Set of all preconditions, $C$: Current branch
 1: function G($D_{pre}, C$)
 2:     $depth \leftarrow$ len($D_{pre}$) $-$ len($C$)
 3:     return $depth$
 4: end function
```

For the A*-search algorithm, the heuristic value and the current profit value need to be added and calculated for every node. The algorithm is shown in Algorithm 4.

Algorithm 4 Branch Evaluation Score

```
Require: $D_{pre}$: All preconditions, $C$: Current branch, $x$: Inconsistent precondition
1: function F($D_{pre}, C, x$)
2:     $g_{val} \leftarrow$ G($D_{pre}, C$)
3:     $h_{val} \leftarrow$ H($C, x$)
4:     return $g_{val} + h_{val}$
5: end function
```

Now, the naive search algorithm can be adjusted to perform the A*-search. First, a branch map is needed that saves the current f-values for every computation path. The determination of inconsistencies and the construction of the subsets C_1 and C_2 is taken from the naive algorithm and works identically. The first adjustment is that the f-value is calculated for every subset and inconsistent precondition found. Then the subset and the corresponding f-value are stored in the branch map. The next subset is computed by finding the current branch with the lowest f-value. This branch is then expanded by recursing the algorithm with the current best branch. The A*-search algorithm used to find inconsistencies between the preconditions is shown in Algorithm 5.

Algorithm 5 Informed Scenario Generation

```
Require: $D_{pre}$: Set of all preconditions, $\Delta_{check}(\cdot)$: Returns two contradicting preconditions, $f(\cdot)$: The f-value function, $Next(\cdot)$: Returns branch with highest f value
1: $M_I \leftarrow \{\}$
2: $B \leftarrow \{\}$
3: function GetInconsistencies($D_{pre}, \Delta_{check}(\cdot), f(\cdot)$)
4:     $T_{contra} \leftarrow \Delta_{check}(D_{pre})$
5:     if $T_{contra}$ is empty then
6:         return $M_I$
7:     else
8:         $(x_1, x_2) \leftarrow T_{contra}$
9:         $C_1 \leftarrow$ remove $x_1$ from $D_{pre}$
10:        $C_2 \leftarrow$ remove $x_2$ from $D_{pre}$
11:        add $x_2$ to $M_I[x_1]$
12:        add $x_1$ to $M_I[x_2]$
13:        $B[C_1] \leftarrow f(D_{pre}, C_1, x_1)$
14:        $B[C_2] \leftarrow f(D_{pre}, C_2, x_2)$
15:        $C_{next} \leftarrow$ Next($B$)
16:        return GetInconsistencies($C_{next}, \Delta_{check}(\cdot), f(\cdot)$)
17:    end if
18: end function
```

To build the scenario from the produced inconsistency map algorithm 6 can be used. This scenario-building algorithm simply iterates over all preconditions and adds them to a list. Before adding the precondition, the Z3 SMT solver checks whether the current precondition contradicts any preconditions already in the list. If not, it is added, but if there is an inconsistency, a new list is created. The new list is a copy of the current list, but each precondition that contradicts the current precondition is omitted. Then the current precondition is added. Thus, there are now two lists of preconditions that have no inconsistencies and are maximal concerning their preconditions. This procedure is iterated until all preconditions are added to all possible lists.

Algorithm 6 Building Scenarios

```
Require: $D_p$: Set of preconditions, $M_I$: Inconsistency map
1: function GetScenarios($D_p, M_I$)
2:     $Res \leftarrow [[]]$
3:     for $x_{pre}$ in $D_p$ do
4:         for $D_{curr}$ in $Res$ do
5:             if $x_{pre}$ not in $\bigcup_{x \in D_{curr}} M_I[x]$ then
6:                 add $x_{pre}$ to $D_{curr}$
7:             else
8:                 $D_{new} \leftarrow$ copy of $D_{curr}$
9:                 for $x_{contra}$ in $M_I[x_{pre}]$ do
10:                    if $x_{contra}$ in $D_{new}$ then
11:                        remove $x_{contra}$ from $D_{new}$
12:                    end if
13:                    add $x_{pre}$ to $D_{new}$
14:                    add $D_{new}$ to $Res$
15:                end for
16:            end if
17:        end for
18:    end for
19:    return $Res$
20: end function
```

In the experiments, the search algorithm 5 always terminated within seconds even for the translated Daimler dataset [26]. However, since the heuristic is only an estimate of the inconsistencies to be found, inconsistent preconditions may occur in the same scenario. Without checking the preconditions themselves for inconsistencies, this may lead to missed inconsistencies in the requirement set. This rarely occurred in our tests, but once it occurs, one of the inconsistencies has to be removed. Another adjustment was made to the formulas using the $AF(\varphi)$ operator. Following the current state, the operator indicates that φ will eventually hold in all computation paths. However, since this can be arbitrarily far away, the solver may not terminate and push the occurrence of φ arbitrarily far away. Thus, the operator was abstracted to $AF_{=0}(\varphi)$ to force the occurrence of φ. Since the unsatisfiability resulting from such an abstraction implies the unsatisfiability of the version without abstraction, it does not affect the correctness of the consistency checking procedure. In the case of the timed response pattern, preconditions, and postconditions get the same time constraint to enforce postconditions immediately after preconditions are fulfilled.

VII. Evaluation

A. Dataset

To implement and evaluate the methods proposed in this document, an industrial automotive dataset made by Daimler [25] was used. The dataset consists of 117 industrial requirements and design decisions. The requirements are formulated for two automotive systems: First, the Adaptive Light System (ALS) and second, the Advanced Driver Assistance System (ADAS).

The ALS requirements specify a range of standard system functions, such as the vehicle's direction indicators flash in response to the steering column lever or the function of the hazard warning switch. It also includes a function for daytime running light and an adaptive high beam to regulate headlights depending on the high beam switch and the detection of oncoming vehicles. The ADAS, the dataset incorporates requirements related to the primary components of a speed limiter and adaptive cruise control, which maintains the distance to the vehicle in front using the maximum speed chosen by the driver or by traffic sign detection. In addition, it provides distance warning and an emergency brake assistant that reacts to stationary and moving obstacles.

The specification consists mainly of behavioral (functional) requirements, but headers and some descriptive requirements are also part of it. Behavior requirements describe the relationships of actions. For example, they describe what exact event occurs after a specific action was performed, describe which events should never occur, or describe which properties should always hold. Many requirements have additional constraints regarding the scope in which the requirement should be met or the time of events.

B. Manual Transformation

The Daimler dataset [25] was translated into Structured English manually according to the specification patterns. The dataset was filtered as described in the following steps. The original dataset contains headers and purely descriptive statements. For example, *"The exterior lighting system contains the following user functions"*, or *"Note: The reduction of power serves the protection of the illuminant"*. These requirements do not influence any behavior of the system but are only informative. Thus, they can be omitted from the data. The translated Daimler dataset [26] contains only requirements that contain information about the system. However, there are still many requirements that are not *behavioral requirements*, for example, *"The gas pedal is mounted in the footwell area of the driver."* is neither descriptional nor a header but still does not carry information about the system behavior. Such requirements can also cause inconsistencies but are not expressable in Structured English, therefore, such requirements are also omitted. Note that such limitations can be bypassed by extending Structured English or using other languages. Some requirements consisted of multiple sentences, where some sentences were expressible in Structured English while others were not. In this case, the requirements were split into expressable sentences and inexpressible sentences. After these steps, about 75% of the remaining requirements could be expressed in Structured English.

The distribution of the resulting Structured English requirements is shown in Table I. Note that some natural language requirements were transformed into more than one Structured English requirement. Another observation was that about 79% of the transformations used the *Globally* scope, which also aligns with the results from a case study using Bosch requirements [4] in which about 80% of the Structured English requirements used this scope. On top of this, only two patterns were enough to express every transformed requirement.

TABLE I
PATTERN DISTRIBUTION

Pattern	Globally	Before	After	Between	After Until
Universality	15	0	1	0	7
Response	63	1	5	0	7

C. Automated Transformation

Due to the classification accuracy score of 90%, we assumed that it is reasonable to restrict the data for the transformation to one pattern for the evaluation of the transformation. This way we assume a perfect classification to gain insights into the actual quality of the transformation. Therefore, a subset of the translated Daimler dataset [26] was selected. Namely, each of the 38 natural language requirements was manually formalized into exactly one Structured English requirement using the response pattern. The transformation can now be performed by selecting one of the 38 requirements as a test prompt that should be transformed into Structured English and using the remaining 37 transformations as the training dataset. Then, the context is selected with the OptKATE algorithm, and the model transforms the test prompt with the provided context. We split the translation into three steps, which are exemplarily depicted in Figure 5. The first step is to create

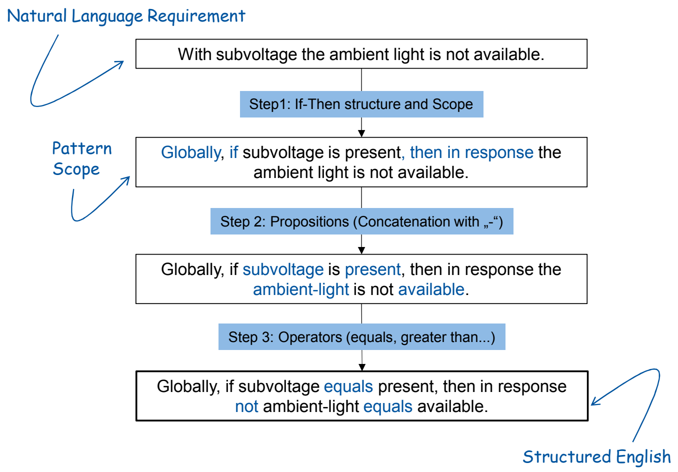

Natural Language Requirement
With subvoltage the ambient light is not available.
Step1: If-Then structure and Scope
Pattern
Scope
Globally, if subvoltage is present, then in response the ambient light is not available.
Step 2: Propositions (Concatenation with „-")
Globally, if subvoltage is present, then in response the ambient-light is not available.
Step 3: Operators (equals, greater than...)
Globally, if subvoltage equals present, then in response not ambient-light equals available.
Structured English

Fig. 5. Cascading Translation Example

the sentence structure of the target pattern. The second step is to introduce the propositions for the propositional formulas. The last step is to introduce the correct operators.

For the distance function, we tried different semantic encodings of the natural language requirements and then calculated the cosine similarity. The encodings used were Universal Sentence Encoder (USE) [29] and the SBERT Encoder [30], since both have shown good performance in the Semantic Textual Similarity benchmarks. We also used a Bag of Words Encoding [31], which sums up the one hot encodings of the words of a requirement sentence. This encoding was used since one of the main tasks during the transformation of the requirements is to introduce correct propositions. We assumed that selecting examples that use vocabulary similar to that of the test prompt might result in better transformation performance. The vector length of the Bag of Words Encodings is therefore equal to the vocabulary size used in the specification. We used two different few-shot translation approaches. The first approach (*All-In-One*) was to give all three transformation steps in the context and then let the model perform all three steps on the test prompt. The second approach (*True-Cascading*) was to let the model generate each transformation step at once, only giving the examples for the current transformation step and the test prompt (in the respective step) as context.

For transformation evaluation, we used the BLEU score [32], as well as the SE score [33], which is a manual metric defined as follows:

Class 1: The translation is both syntactically and semantically correct and fulfills the required formulation rule. No changes are required.

Class 2: The translation is semantically correct but contains one or two syntactical inaccuracies to fully implement the desired rule.

Class 3: Syntactically correct but fails to fully cover the semantics of the source requirement (e.g. by missing a quantifier or a marginal constraint).

Class 4: The translation contains one or two syntactical inaccuracies to fully implement the desired rule and the semantics are not fully covered, i.e. a combination of 2 and 3.

Class 5: The translation has grave syntactical errors or does not implement the desired rule. An identity mapping would result in this label as well (unless the input already implements the desired rule).

Class 6: The translation is semantically wrong.

We transformed the dataset with different combinations of the properties described above to find the best combination of distance function and context format. The best scores were achieved using the All-In-One format with the Bag of Words Encoding Distance (BLEU: 0.47 and SE: 3.26) and the True-Cascading format with the USE Distance (BLEU: 0.54 and SE: 3.32). To put the results in the context of a semiautomated transformation approach, the average assignment number of the SE classes of the requirements was calculated with the All-In-One format and the Bag of Words Encoding Distance. Class 1 was interpreted as a transformation that can be used without the need for any further adjustment. Class 2 to Class 4 were interpreted as classes where semantically or syntactical adjustments must be made. Class 5 and Class 6 however denote that the transformation has to be done completely manually. The average number of class assignments for the 38 response pattern requirement transformations is shown in Table II. Thus, regarding the All-In-One format with the Bag of Words Encoding Distance, on average 27.2% of the transformations could be used without adjustments, around 44.7% needed some adjustments, and around 28.1% needed a manual transformation. For the Cascading format with USE Distance 30. 7% of the requirements could be used without adjustments, around 37.7% needed adjustment, and 31.6% of the requirements had to be transformed manually.

TABLE II
AVERAGE SE-CLASS ASSIGNMENTS

Classes	Assignments (All-In-One)	Assignments (Cascading)
Class 1	10.33	11.66
Class 2	4.00	1.66
Class 3	6.33	10.66
Class 4	6.66	2.00
Class 5	4.00	0.66
Class 6	6.66	11.33

In general, it was observed that the distance function, in combination with the input format, can change the quality of transformation. Quality increased dramatically compared to the random baseline with a BLEU score that tripled with respect to 0.16 for the random baseline compared to 0.55 and 0.53 for the best transformations of the response pattern data. The average SE score also indicates that the transformation went from almost unusable, indicated by the SE score of 5.78 for the random baseline, to transformation results that can be used in a semiautomated transformation process, where some transformations need correction, but also common transformations are flawlessly by the model. However, one has to try some promising combinations such as the All-In-One format with Bag of Words Encoding Distance, or the USE Distance with the cascading format, since the evaluation has not shown a method that was superior to the other.

D. Consistency Checking

For the evaluation of the consistency checking procedure, a total of six subsets were randomly generated from the translated Daimler dataset [26], each consisting of 10 requirements. The consistency checking procedure was used on each subset, and then some inconsistencies were additionally added to the subsets, and the procedure was performed again. The added inconsistencies were either *direct inconsistencies* using the universality pattern, *postcondition inconsistencies* using the response pattern, or *temporal inconsistencies* using the universal or the response pattern. Direct inconsistencies are quite easy to find and do not rely on the scenario builder. To find the postcondition inconsistencies it is necessary that the scenario building works, thus that the preconditions of the requirements including the inconsistent postconditions are fulfilled. To check this, some requirements with inconsistent postconditions *and* preconditions were introduced. These requirements should not be returned by the consistency checker, since the preconditions cannot be fulfilled at the same time. Temporal inconsistencies were introduced to evaluate whether the transformation from TCTL time constraints to SMT assertions works properly. The evaluation results are shown in Table III.

TABLE III
CONSISTENCY CHECKING EVALUATION

Sets	Consistency Checking			
	Scenarios	Inconsistencies	Added	Found
Set 1	4	0	1	1
Set 2	3	0	1	1
Set 3	1	0	1	2
Set 4	6	1	1	2
Set 5	0	1	1	1
Set 6	1	1	3	4

In almost all cases precisely, the inconsistent requirements were found. However, in Set 3 an incorrect inconsistency was detected. The inconsistency reported is a single requirement:

> ”After speed-limit equals temporarily-deactivated until not speed-limit equals temporarily-deactivated, if gas-pedal-level less than 90%, then in response speed-limit equals activated-again.”

The requirement expresses that between deactivation and the next activation of the speed limit, if the gas pedal is below a level of 90%, the speed limit is activated again. The requirement is consistent in itself. The problem results from the precondition of the requirement that models the speed limit's deactivation and its mapping to TCTL with the universality pattern. Hence, the precondition expresses that the speed limit is deactivated all the time. The other precondition in the requirement is that the level of the gas pedal is below 90%. Since this is consistent with the deactivation of the speed limit, both preconditions are added to a scenario. Then the postcondition, which expresses the speed limit's activation, holds and is inconsistent with its precondition that the speed limit is deactivated. This results in an inconsistency reported by the consistency checker. Therefore, a solution would be to time the preconditions, thus only modeling their occurrence instead of modeling that they hold all the time. But to do this, a time point for *occurrence* has to be determined for all preconditions that should hold at the same time, without introducing inconsistencies between preconditions and requirements by accident. This task is not trivial, and due to the rare occurrence of such cases, an adjustment of the procedure is not proposed. Thus, the user must keep this case in mind. A solution to this problem is open to further research. It is recommended to check for correctness of each reported inconsistency. The results also show that none of the requirements with inconsistent pre- and postconditions were reported by the consistency checker, which shows that the scenario generator is working. Thus, the procedure found all inconsistencies in the subsets within seconds, due to an approximated scenario generation procedure and a structured satisfaction of consistent preconditions, generating test sets that reflect all possible scenarios according to the specification.

VIII. DISCUSSION AND OUTLOOK

In this paper, a complete toolchain for checking the consistency of natural language requirements was presented. By first transforming the informal language requirements into Structured English with a LLM, further formal processing of the requirements is made possible. Then, converting the requirements first into TCTL and afterward into SMT-LIB specifications prepares an easy integration of the SMT solver. The SMT solver checks translated requirements for inconsistencies and presents the results. With this toolchain, the requirement engineer can automatically check refined or new written requirements without doing the time-prone process of classification and formalization into Structured English patterns by himself.

LLMs are explicitly used only for the translation step from natural language into Structured English. These results can be read and evaluated by the requirements engineer. This evaluation is necessary because, unlike the subsequent transformation steps of the toolchain, complete correctness cannot be guaranteed when using LLM. But we think that the translation that has already taken place, in addition to making this step generally easier, will also help Structured English to be learned more quickly.

The toolchain was implemented as a prototype and evaluated on the Daimler dataset [25], [26] using the GPT-J-6B [15]. Application on more automotive datasets are planned to evaluate the overall suitability of Structured English. The extent to which knowledge of the hierarchy of requirements within specifications can further improve translation results is another research question, as we currently only rely on the requirements themselves. Comparison of results depending on the chosen LLM would benefit possible applications in an industrial context. Therefore, a comparison of the translation results is planned on larger models such as GPT-NeoX-20B [34].

In the automotive industry, specifications often consist of a large number of textual requirements. These requirements are linguistically ambiguous and written in informal language. Utilizing Structured English for requirements eliminates ambiguity, improves data quality, and supports further automated processing while maintaining readability. The recent development of large language models enables a fully automated translation approach using few-shot learning. To deal with the limited context size of large language models, an improved algorithm, OptKATE, is presented to find an ideal requirement set for few-shot learning. Structured English can be used as a basis for further formalization. This capability is key in creating an interface between natural language processing and verification, in our case, consistency analysis using the Z3 SMT solver. We implemented a grammar for translating Structured English into TCTL using the MontiCore workbench. Furthermore, since SMT-based methods currently rely on manual precondition satisfaction and do not tackle conflicting preconditions automatically, we propose a scenario generation algorithm that generates potential scenarios using the specification and checks the requirements against them. Through this approach, we can better identify and resolve conflicting preconditions, ultimately improving the consistency of requirements. Our toolchain is evaluated using an automotive requirements dataset provided by former Daimler AG.

References

[1] L. E. G. Martins and T. Gorschek, "Requirements engineering for safety-critical systems: Overview and challenges," *IEEE Software*, vol. 34, no. 4, pp. 49–57, 2017.

[2] R. Alur, C. A. Courcoubetis, and D. L. Dill, "Model-checking in dense real-time," *Information and computation*, vol. 104, pp. 2–34, 1993.

[3] S. Konrad and B. Cheng, "Real-time specification patterns," in *Proceedings - 27th International Conference on Software Engineering, ICSE05*, 06 2005, pp. 372– 381.

[4] A. Post, I. Menzel, J. Hoenicke, and A. Podelski, "Automotive behavioral requirements expressed in a specification pattern system: A case study at bosch," *Requirements Engineering*, vol. 17, pp. 19–33, 03 2012.

[5] P. Filipovikj, T. Jagerfield, M. Nyberg, G. Rodriguez-Navas, and C. Seceleanu, "Integrating pattern-based formal requirements specification in an industrial tool-chain," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2, 2016, pp. 167–173.

[6] J. S. Becker, V. Bertram, T. Bienmüller, U. Brockmeyer, H. Dörr, T. Peikenkamp, and T. Teige, "Interoperable toolchain for requirements-driven model-based development," in *ERTS 2018*, 2018.

[7] M. Dwyer, G. Avrunin, and J. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, 1999, pp. 411–420.

[8] R. Alur, C. Courcoubetis, and D. Dill, "Model-checking in dense real-time," *Information and Computation*, vol. 104, no. 1, pp. 2–34, 1993. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0890540183710242

[9] C. W. Barrett, A. Stump, and C. Tinelli, *The SMT-LIB Standard Version 2.0*. The SMT-LIB Initiative, 2010.

[10] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, p. 337–340.

[11] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," *Advances in neural information processing systems*, 2020.

[12] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[13] J. Dodge, G. Ilharco, R. Schwartz, A. Farhadi, H. Hajishirzi, and N. A. Smith, "Fine-tuning pretrained language models: Weight initializations, data orders, and early stopping," *arXiv preprint arXiv:2002.06305*, 2020.

[14] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, 2017. [Online]. Available: https://arxiv.org/abs/1706.03762

[15] B. Wang and A. Komatsuzaki, "GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model," https://github.com/kingoflolz/mesh-transformer-jax, May 2021.

[16] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, S. Presser, and C. Leahy, "The pile: An 800gb dataset of diverse text for language modeling," *CoRR*, vol. abs/2101.00027, 2021. [Online]. Available: https://arxiv.org/abs/2101.00027

[17] "Colaboratory," Google Research. [Online]. Available: https://colab.research.google.com/

[18] M. Autili, L. Grunske, M. Lumpe, P. Pelliccione, and A. Tang, "Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar," *IEEE Transactions on Software Engineering*, vol. 41, no. 7, pp. 620–638, 2015.

[19] J. Liu, D. Shen, Y. Zhang, B. Dolan, L. Carin, and W. Chen, "What makes good in-context examples for gpt-3?" in *Proceedings of Deep Learning Inside Out (DeeLIO 2022): The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures*, 2022.

[20] A. Butting, K. Hölldobler, B. Rumpe, and A. Wortmann, "Compositional Modelling Languages with Analytics and Construction Infrastructures Based on Object-Oriented Techniques - The MontiCore Approach," in *Composing Model-Based Analysis Tools*, Heinrich, Robert and Duran, Francisco and Talcott, Carolyn and Zschaler, Steffen, Ed. Springer, July 2021, pp. 217–234. [Online]. Available: http://www.se-rwth.de/publications/Compositional-Modelling-Languages-with-Analytics-and-Construction-Infrastructures-Based-on-Object-Oriented-Techniques-The-MontiCore-Approach.pdf

[21] P. Filipovikj, "Model-checking-based vs. smt-based consistency analysis of industrial embedded systems requirements: Application and experience," *Electronic Communications of the EASST*, vol. Volume 75: 43rd International Conference on Current Trends in Theory and Practice of Computer Science - Student Research Forum, p. 2017 (SOFSEM SRF 2017), 2018. [Online]. Available: https://journal.ub.tu-berlin.de/eceasst/article/view/1054/1033

[22] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "cvc5: A versatile and industrial-strength SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS*, ser. Lecture Notes in Computer Science, D. Fisman and G. Rosu, Eds., vol. 13243. Springer, 2022, pp. 415–442. [Online]. Available: https://doi.org/10.1007/978-3-030-99524-9_24

[23] J. Christ, J. Hoenicke, and A. Nutz, "SMTInterpol: An interpolating SMT solver," in *Model Checking Software*. Springer Berlin Heidelberg, 2012, pp. 248–254. [Online]. Available: https://doi.org/10.1007/978-3-642-31759-0_19

[24] P. Filipovikj, G. Rodriguez-Navas, M. Nyberg, and C. Seceleanu, "Automated smt-based consistency checking of industrial critical requirements," *ACM SIGAPP Applied Computing Review*, vol. 17, no. 4, p. 15–28, jan 2018. [Online]. Available: https://doi.org/10.1145/3183628.3183630

[25] V. Bertram, S. Maoz, J. O. Ringert, B. Rumpe, and M. von Wenckstern, "Component and Connector Views in Practice: An Experience Report," in *Conference on Model Driven Engineering Languages and Systems (MODELS'17)*. IEEE, September 2017, pp. 167–177. [Online]. Available: http://www.se-rwth.de/publications/Component-and-Connector-Views-in-Practice-An-Experience-Report.pdf

[26] V. Bertram, M. Boß, E. Kusmenko, I. H. Nachmann, B. Rumpe, D. Trotta, and L. Wachtmeister, "Technical report on neural language models and few-shot learning for systematic requirements processing in mdse," November 2022. [Online]. Available: http://www.se-rwth.de/publications/Technical-Report-on-Neural-Language-Models-and-Few-Shot-Learning-for-Systematic-Requirements-Processing-in-MDSE.pdf

[27] R. Malouf, "Maximal consistent subsets," *Computational Linguistics*, vol. 33, pp. 153–160, 06 2007.

[28] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[29] D. Cer, Y. Yang, S. Kong, N. Hua, N. Limtiaco, R. S. John, N. Constant, M. Guajardo-Cespedes, S. Yuan, C. Tar, Y. Sung, B. Strope, and R. Kurzweil, "Universal sentence encoder," *CoRR*, vol. abs/1803.11175, 2018. [Online]. Available: http://arxiv.org/abs/1803.11175

[30] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," *CoRR*, vol. abs/1908.10084, 2019. [Online]. Available: http://arxiv.org/abs/1908.10084

[31] W. A. Qader, M. M. Ameen, and B. I. Ahmed, "An overview of bag of words;importance, implementation, applications, and challenges," in *2019 International Engineering Conference (IEC)*, 2019, pp. 200–204.

[32] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, Jul. 2002, pp. 311–318. [Online]. Available: https://aclanthology.org/P02-1040

[33] V. Bertram, M. Boß, E. Kusmenko, I. Nachmann, B. Rumpe, D. Trotta, and L. Wachtmeister, "Neural Language Models and Few-Shot Learning for Systematic Requirements Processing in MDSE," in *International Conference on Software Language Engineering (SLE'22)*. ACM, December 2022, pp. 260–265. [Online]. Available: http://www.se-rwth.de/publications/Neural-Language-Models-and-Few-Shot-Learning-for-Systematic-Requirements-Processing-in-MDSE.pdf

[34] A. Andonian, S. Biderman, S. Black, P. Gali, L. Gao, E. Hallahan, J. Levy-Kramer, C. Leahy, L. Nestler, K. Parker, M. Pieler, S. Purohit, T. Songz, W. Phil, and S. Weinbach, "GPT-NeoX: Large Scale Autoregressive Language Modeling in PyTorch," Mar. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7714278