Formal Methods in Computer-Aided Design 2024

Translating Natural Language to Temporal Logics with Large Language Models and Model Checkers

Daniel Mendoza
Stanford University
Stanford, CA, USA
dmendo@stanford.edu

Christopher Hahn†
X, the moonshot factory
Mountain View, CA, USA
chrishahn@google.com

Caroline Trippel
Stanford University
Stanford, CA, USA
trippel@stanford.edu

***Abstract*—Automating translation from natural language (NL) to temporal logic (TL) specifications offers to democratize verification of hardware systems. However, this vision is challenged by (i) inherent ambiguity in NL, which can create multiple plausible translation possibilities, and (ii) the need to validate translation output, when the translator can make mistakes. To address these challenges, we propose SynthTL, an interactive approach and tool, which uses large language models (LLMs), model checkers, and oracle (human) guidance, to translate an NL specification of a hardware design's intended behavior into a TL specification that reflects the NL and holds (formally) on the design.**

SynthTL performs *structured translation*, whereby it first decomposes a complex unstructured NL specification into a logical combination of simple NL sub-specifications. Then, it produces TL translations of the simple NL sub-specifications, called *sub-translations*, and mechanically combines these sub-translations to yield a TL translation of the complex NL specification. This approach significantly reduces the oracle effort required to validate the translation output, since only sub-translations and the logical relationships between them need to be inspected. Plus, it enables the design of automated model checker utilities, which efficiently guide the search for a translation that holds on the design, despite inherent NL ambiguity. With SynthTL, we conduct the largest LLM-assisted NL to TL specification translation case study to date, producing a correct formalization of the Arm AMBA AHB bus protocol.

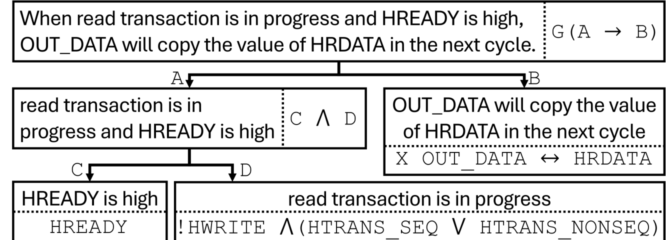

Fig. 1: *Sub-translation tree* that mechanically composes to produce LTL specification: $G((\texttt{HREADY} \wedge !\texttt{HWRITE} \wedge (\texttt{HTRANS_SEQ} \vee \texttt{HTRANS_NONSEQ})) \rightarrow X(\texttt{OUT_DATA} \leftrightarrow \texttt{HRDATA}))$. Nodes (*sub-translations*) consist of an *NL sub-specification* and a corresponding *TL sub-specification*. Directed edges from a parent node to its children denote a decomposition.

I. Introduction

Hardware verification involves checking that a *Design Under Test* (DUT) upholds desired properties through simulation, formal model checking, and/or runtime verification [1]. These properties, or *specifications*, are typically expressed as *Temporal Logic* (TL) formulas (e.g., using Linear Temporal Logic (LTL) [2], SystemVerilog Assertions (SVAs) [3], or Property Specification Language (PSL) [4] syntax), which formally define sets of allowed execution traces on the DUT. Today, verification experts *manually* derive TL specifications for a DUT from (hardware designer-friendly) *natural language* (NL) specifications. Consequently, the time and resources required to produce thorough design-specific TL specifications limits the application of verification in practice [5].

The NL to TL specification translation bottleneck is a natural target for *natural language processing* (NLP) approaches (e.g., *Large Language Models*, or LLMs), which have recently been deployed to resolve it [6], [7], [8], [9], [10], [11], [12], [13]. Yet, NLP is not a panacea. First, producing TL specifications that capture the intent of NL is challenged by its inherent ambiguity; a single NL phrase may represent *many* plausible TL formulae. Second, NLP approaches are susceptible to producing blatantly incorrect outputs.

For these reasons, NLP-based NL to TL specification translation can produce an abundance of candidate outputs, especially when specifications are complex. For instance, GPT4 [14] generates over 100K unique TL specification possibilities when translating an NL specification of the *arbiter* component of the Arm AMBA AHB bus protocol [15], [16] to TL in our case study (§V). Existing NLP-based TL formalization approaches thus rely on a human user to manually validate final generated outputs, i.e., *full* TL specifications [6], [7], [8], [9], [10], [11], [12], [13]. Unfortunately though, validating full output specifications can be just as difficult as writing them from scratch.

A. This Paper

We propose SynthTL, an approach and tool that uses LLMs, model checkers, and oracle (human) guidance, to translate an NL specification of some DUT's desired behavior into a TL specification that reflects the NL and holds (formally) on the DUT. SynthTL leverages LLMs to perform *structured translation of unstructured NL*, whereby it first decomposes a complex unstructured input *NL specification* (i.e., an NL specification that does not conform to a predefined

†Work done while at Stanford University.

https://doi.org/

This article is licensed under a Creative Commons Attribution 4.0 International License

grammar) into a logical combination of simple *NL sub-specifications*. Then, it translates each NL sub-specification to a *TL sub-specification* and mechanically combines all TL sub-specifications according to their logical structure to yield a complete output *TL specification*.

SYNTHTL's structured translation procedure can be visualized as a *sub-translation tree* (Figure 1). Nodes represent *sub-translations*, consisting of an NL sub-specification and a TL sub-specification, and labeled directed edges denote a decomposition of a parent sub-translation into a logical combination of simpler child sub-translations. The NL sub-specification of a sub-translation tree's root node is the (full) NL specification input to SYNTHTL, from which the tree is recursively generated using LLMs (with optional oracle input) as follows. First, SYNTHTL introduces zero or more fresh symbols to represent (serve as "placeholders" for) unique strict substrings in a parent node's NL sub-specification. Then, it generates the parent node's TL sub-specification as a TL formula over these symbols. Finally, for each symbol, it instantiates a child node, whose incoming edge is labeled with the symbol and whose NL sub-specification is the substring that the symbol represents.

SYNTHTL expects the oracle to inspect each node of a complete sub-translation tree to validate that its TL sub-specification is a reasonable translation of its NL sub-specification and that its child nodes exhibit a reasonable decomposition. Inaccuracies are corrected by the oracle, possibly with the help of LLMs or other external tooling. Upon oracle sign-off, SYNTHTL uses model checkers to evaluate whether the full tree's corresponding (mechanically generated) TL specification holds on the DUT. If it does, the NL specification, TL specification, and DUT are all deemed correct, since oracle sign-off (earlier) confirms the TL specification is consistent with the NL specification. If it does not, there is a bug in the (consistent) NL/TL specifications and/or the DUT, warranting further oracle investigation.

Our primary insight is that SYNTHTL's structured translation approach drastically reduces overall oracle effort. First, when validating a sub-translation tree, an oracle need only inspect simple sub-translations (nodes) and their immediate decompositions (children) and never the end-to-end translation (i.e., full TL specification). Second, model checkers can leverage this tree structure to guide the oracle towards an NL/TL specification or DUT fix when oracle tree validation or model checker TL specification evaluation fails.

SYNTHTL offers two such model checker-guided utilities: *culprit identification* and *translation search*. When an output TL specification does not hold on the DUT, culprit identification uses model checkers to find sub-translations (from its sub-translation tree) that logically contribute to the failing output specification. The oracle can prioritize fixing these sub-translations (i.e., NL/TL sub-specifications), their decompositions, or the DUT, as appropriate. To account for inherent ambiguity in NL and/or inaccurate sub-translations, translation search enables the oracle to provide (manually or using an LLM) multiple decomposition options and/or TL sub-specification options per tree node, yielding *set* of possible sub-translation trees (syntactically unique TL specifications) to be systematically checked against the DUT.

Unsurprisingly, SYNTHTL's translation search utility often produces a set of unique sub-translation trees that is too large to model check exhaustively. However, **our secondary insight** is that SYNTHTL's structured translation can be exploited to make this analysis practical. First, SYNTHTL's translation search utility is designed to detect inconsistent sub-trees, which cannot be extended to a full TL specification that holds on the DUT; its *sub-tree pruning* optimization discards all sub-translations trees that contain these sub-trees to avoid redundantly checking them on the DUT. Second, translation search uses a *batch model checking* optimization, which identifies all unique conjunctive clauses among the remaining (not pruned) full translations that must be checked on the DUT and queries a model checker at most once for each.

Overall, this paper significantly eases the burden of NL to TL specification translation via the following contributions:

- **SYNTHTL Approach & Tool:** We propose SYNTHTL to translate NL to TL specifications that both represent the intent of the NL and hold on some target DUT, using LLMs, model checkers, and oracle guidance.
- **Structured Translation:** SYNTHTL decomposes a (complex) NL to TL translation problem into a set of simpler, mechanically-composable sub-problems, organized as a sub-translation tree, that are easier to solve and validate.
- **Model Checker-Guided Translation:** SYNTHTL uses model checkers to guide an oracle towards a correct NL to TL translation by flagging potential culprit sub-translations when an output TL specification does not hold on the DUT and efficiently uncovering a correct translation out of a large space of possible options.
- **Case Study:** We use SYNTHTL to translate three real-world NL specifications—comprising the Arm AMBA AHB bus protocol [15], [16]—to LTL. Our evaluation features much larger NL specifications (maximum/average of 643/449 words) and TL specifications (maximum/average of 490/434 symbols, i.e., LTL operators and variables) than prior work, which focuses on individual NL properties within a larger specification. For comparison, each NL specification in the recent `nl2spec` dataset [6] contains a maximum/average 21/10 words; TL specifications contain a maximum/average of 37/10 symbols. Among 7.26e16 LLM-generated candidate TL specifications for an NL specification of the *controller* component of the AMBA AHB protocol, SYNTHTL converges to a correct one, consisting of 56 sub-translations. In doing so, SYNTHTL queries an oracle to validate 96 sub-translations and to fix 18/11 TL sub-specification/decompositions, where the average TL formula fixed by the oracle is 3% the size of the full correct TL specification.

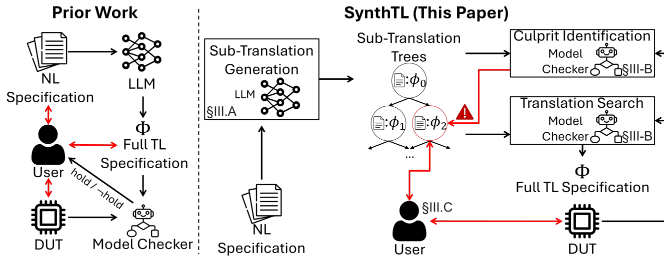

Fig. 2: Existing NL to TL specification translation approaches (left) rely on manual validation of full TL specifications. SYNTHTL (right) reduces manual effort using sub-translation trees and model checker guidance.

II. Related Work and Motivation

In this section, we give an overview of prior work on translating structured (§II-A) and unstructured (§II-B) NL to TL specifications in order to motivate SYNTHTL (§II-C), which translates unstructured NL to TL.

A. Structured Translation of Structured NL

Early work on NL to TL specification translation requires *structured* NL as input, i.e., NL that conforms to a pre-defined grammar [17], [18], [19], [20]. By *mechanically* deriving an output TL specification from a structured NL input, these approaches conduct *structured translation*, similar to SYNTHTL once it has produced a sub-translation tree. The output TL specification is guaranteed to be correct if the input NL specification is. However, supplying structured NL inputs can be burdensome.

B. Unstructured Translation of Unstructured NL

Recent NLP advances have inspired numerous efforts to translate *unstructured* NL (i.e., no grammar restrictions) to TL specifications, e.g., using LLMs [6], [8], [9], [11], [12], [10] and other NLP methods [7], [13]. Unlike SYNTHTL, these approaches conduct *unstructured translation*, meaning that an LLM or NLP algorithm ultimately constructs the final TL specification output in one shot (Figure 2, left).

Translating unstructured NL is challenged by its inherent ambiguity, e.g., the phrase "signal READY holds" can be formulated in various plausible ways, like the following in LTL: `READY`, X`READY`, G`READY`, `READY` $\leftrightarrow$ X`READY`. Plus, NLP is prone to making translation mistakes. To resolve both issues, prior work [6], [7], [8], [9], [10], [11], [12], [13] expects a user to manually validate the final formalization output, confirming that it captures the intent of the NL input.

Unfortunately, this validation step can be just as difficult as manually performing the entire NL to TL specification translation task. Moreover, when a translated TL specification is deemed inconsistent with the DUT (e.g., by a model checker), localizing bugs in the input NL specification, output TL specification, and/or DUT is difficult, as is fixing them.

We categorize prior work on unstructured translation of NL to TL specifications based on whether translation is conducted *end-to-end* (§II-B1) or *interactively* (§II-B2).

1) End-to-end Translation: End-to-end approaches [10], [11], [8], [12], [13], [9] deploy specialized prompting or training schemes to elicit accurate NL to TL specification translations from an NLP model without soliciting user input beyond the NL specification itself.

2) Interactive Translation: Interactive approaches, like LTLtalk [7] and `nl2spec` [6], query the user to validate/fix candidate translations before the final output is produced.C

LTLtalk [7] queries a user to accept or reject sample traces from full candidate TL specifications. But, manually analyzing traces of a complex TL specification is challenging.

Similar to SYNTHTL, `nl2spec` [6] decomposes the translation task into easier sub-translations and queries a human oracle to fix sub-translations. However, since no structure is enforced among sub-translations, `nl2spec` requires an LLM to compose them into a full TL specification in one shot. Plus, this lack of structure means that formal model checking cannot be applied to automatically localize errors or fix particular sub-translations.

C. Our Approach: Structured Translation of Unstructured NL

We propose *structured translation of unstructured NL* to TL specifications with SYNTHTL (Figure 2, right).

Without loss of generality, we focus our discussion on translating NL to LTL [2], which extends propositional logic with temporal operators including U (until), X (next), F (eventually), and G (globally). Figure 1 illustrates a property from the AMBA AHB bus protocol specification [15], [16] formalized in LTL. Operators can be nested (e.g., $GF\phi$ means ϕ occurs infinitely often).

SYNTHTL uses LLMs, model checkers, and oracle guidance to iteratively transform an input unstructured NL specification into a logical combination of sub-translations, which are mechanically combined to yield an output TL specification. Thus, the oracle (user) need only manually validate (simpler) sub-translations and their logical organization, but never full TL specifications. Plus, this logical organization enables SYNTHTL to localize inconsistencies between an output TL specification and the DUT to sub-translations and efficiently guide the translation procedure towards bug fixes.

III. SynthTL Approach and Tool: Structured Translation of Unstructured Natural Language to Temporal Logics

We present the SYNTHTL approach and tool in this section and detail its model checker utilities in §IV. Figure 3 illustrates how SYNTHTL translates NL phrases from the AMBA AHB bus protocol specification [15], [16] into an LTL specification.

A. Interactive TL Specification Generation

Given an input NL specification, SYNTHTL performs automated sub-translation tree generation using LLMs and queries an oracle to validate the resulting tree(s). A *sub-translation*

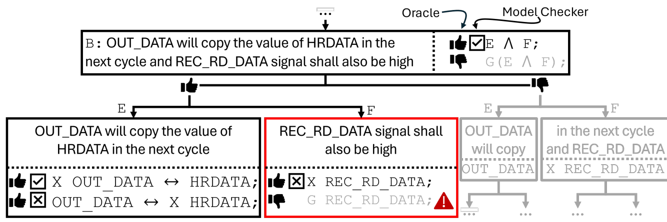

Fig. 3: For each NL sub-specification, SYNTHTL generates multiple possible decompositions and TL sub-specifications. An oracle may accept/reject (thumbs up/down) any of them. To determine if a sub-translation tree's TL specification holds on the DUT, SYNTHTL checks if each of its TL specification holds or not (checks or "x"s). If no tree holds, SYNTHTL flags sub-translations as potential root causes (red nodes).

tree (e.g., Figure 1) represents a single complete translation from an input NL specification to an output TL specification. Each node is a sub-translation, which consists of an NL sub-specification and a corresponding TL sub-specification. Labeled edges denote a decomposition of a parent sub-translation into a logical combination of simpler child sub-translations.

1) Sub-Translation Tree Generation and Validation: In sub-translation tree generation, SYNTHTL first instantiates a sub-translation tree root node whose NL sub-specification is populated with the input NL specification. It then uses LLMs (with optional user guidance) to generate a complete tree by recursively decomposing and translating nodes, starting with the root.

Decomposing a parent node involves first introducing zero or more fresh symbols, each of which serves as a representative for a unique strict substring in the parent's NL sub-specification. For each symbol, a child node is instantiated, with the symbol labeling the directed edge from parent to child. For example, in Figure 1, A labels the edge from the root node to its left child and represents the substring "read transaction is in progress and HREADY is high." The parent node's TL sub-specification is then produced by translating its NL sub-specification under the mapping of its substrings to their representative symbols.

In practice, when supplied with an input NL specification, SYNTHTL generates from it a *set* of sub-translation trees. In particular, for each parent node, it generates up to D unique decomposition options and K unique TL sub-specification options per decomposition; K and D are hyperparameters, which specify the number of LLM queries. While SYNTHTL deploys LLMs to conduct this step, it discards decompositions either where the symbols do not represent unique strict substrings within the parent's NL sub-specification or where one symbol represents a substring of another symbol (i.e., redundant symbols). SYNTHTL also discards TL sub-specifications that: are not well-formed, are trivial (i.e., $\top$ or $\bot$), do not use all symbols that label edges to its node's children, or use symbols beyond those that represent its node's children and variables defined in the DUT.

Following each node decomposition and TL sub-specification generation step during sub-translation tree generation, SYNTHTL queries the oracle to accept/reject each decomposition and TL sub-specification option (thumbs up/down in Figure 3). If ever the oracle rejects all decompositions or TL sub-specifications for a node, SYNTHTL asks the oracle to provide acceptable ones. For each accepted decomposition, fresh decomposition, translation, and validation steps are initiated for its newly-instantiated children.

Recursion terminates when encountering a node with no decomposition (i.e., zero symbols/child nodes). A node cannot be decomposed if the only strict substring that can be derived from its NL sub-specification is the empty string. In practice, LLM-based decomposition may terminate before this condition is reached. For example, in the sub-translation tree in Figure 1, the NL sub-specification "HREADY is high" has no decomposition.

2) Structured Translation of Sub-Translation Trees to TL Specifications: Sub-translation tree generation and validation produces a set of candidate sub-translation trees, which can be obtained by performing a cross product among all validated decomposition and TL sub-specification options per node. The TL specifications implied by these candidate sub-translation trees can be derived recursively as follows. Starting at their root nodes, replace placeholder symbols in each parent node's TL sub-specification with the TL sub-specifications of the symbols' corresponding children. For example, in Figure 1, `C ∧ D` will be transformed into `HREADY` ∧ (`!HWRITE` ∧ (`HTRANS_SEQ` ∨ `HTRANS_NONSEQ`)).

Note that given a sub-translation tree with N nodes, D decomposition options per node, K TL sub-specification options per node, the number of candidate sub-translation trees is $O((KD)^N)$. However, validating all generated trees only requires the oracle to validate decompositions and TL sub-specifications for each node, and thus the number of times the oracle validates a decomposition or TL sub-specification is $O(KDN)$ (significantly lower than all possible trees).

3) LLM Prompts for Sub-Translation Tree Generation: SYNTHTL queries an LLM during sub-translation tree generation to decompose NL sub-specifications and to perform NL to TL sub-specification translation. Our prototype implementation of SYNTHTL relies on *in-context* learning [21] with distinct prompting strategies for each task.

For NL sub-specification decomposition, an LLM is prompted to output a JSON dictionary, which maps variables to substrings of the input NL sub-specification. The prompt includes a description of the decomposition task and examples of correct decompositions.

For NL to TL sub-specification translation (i.e., to produce a sub-translation), SYNTHTL prompts an LLM to generate a TL sub-specification that uses the variables introduced in the decomposition of its corresponding NL sub-specification. The prompt presents examples of correct sub-translations to the LLM. To encourage the LLM to generate TL sub-specifications that use variables in the DUT, SYNTHTL in-

cludes a list of all DUT variable names in the prompt. We also observe that NL context required to correctly translate an individual NL sub-specification is often scattered across the large input NL specification that contains it (as a substring). To handle such situations, SYNTHTL uses *retrieval augmented generation* (RAG) [22] to extract relevant context for an NL sub-specification within an input NL specification using a retrieval model. This context is prepended to the TL sub-specification generation prompt.

4) Sub-Translation Tree Expressiveness: Note that SYNTHTL extracts structure from unstructured NL to make translation easier, but it retains full expressiveness of the unstructured input NL and structured output TL. A sub-translation tree can be understood as overlaying the inductive structure of a well-formed TL formula on top of an NL specification. If no structure can be extracted from the input NL specification, the result is a tree with just the root node (i.e., no decomposition), where the node's corresponding TL sub-specification is the full TL specification. However, we did not encounter such a non-decomposable NL specification in our evaluation (§V). Since TL sub-specifications are TL sub-formulae, and sub-translation trees recursively define a top-level TL formula as logical combination of TL sub-specifications using TL operators, SYNTHTL retains the full expressiveness of the TL.

B. Searching for Translations that Hold on DUT

After sub-translation tree generation and validation (§III-A1), SYNTHTL deploys a translation search procedure to find a translation, or more concretely an output TL specification, that holds on the DUT out of an exponential number of candidates (§III-A2). If some TL specification holds on the DUT, the DUT upholds the NL specification's requirements, since the oracle previously confirmed that the TL specification was a reasonable interpretation of the NL specification (by validating all sub-translations and decompositions, §III-A1). If some TL specification does not hold on the DUT, one of two things could be true: the NL and TL specifications are consistent (due to oracle sign-off), and there is a bug in the DUT; or the NL and TL specifications are inconsistent (due to ambiguity that lead to an errant oracle sign-off). In the latter case, there may or may not be a bug DUT as well.

Given a set of candidate sub-translation trees and their TL specifications, one could query a model checker to determine which hold on the DUT. However, exhaustively checking $O((KD)^N)$ TL specifications (§III-A2) is computationally expensive (and likely infeasible). SYNTHTL's model checker-assisted *translation search* utility (§IV-A) addresses this issue in two ways. First, it leverages *sub-tree pruning* (§IV-A1) to discard many candidate sub-translation trees, which cannot hold on the DUT due to inconsistent sub-trees. Second, when checking DUT adherence to the TL specifications that are not pruned, translation search leverages *batch model checking* (§IV-A2) to ensure that common conjunctive clauses among these specifications are checked at most once. Note that if multiple TL specifications hold on the DUT, SYNTHTL returns only the most constrained (i.e., most restrictive) ones

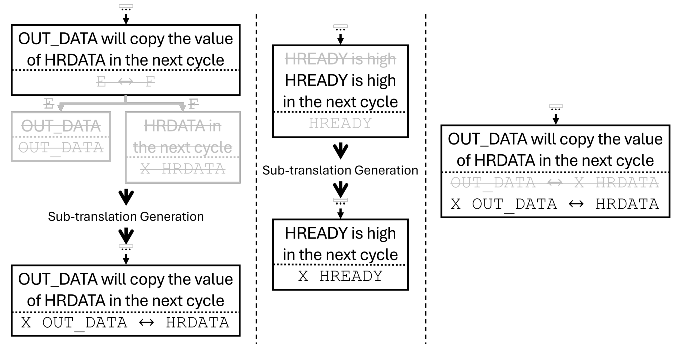

Fig. 4: Example of how SYNTHTL's users may fix a culprit node's decompositions (left), NL sub-specification (middle), or TL sub-specifications (right). After editing a decomposition or NL sub-specification, SYNTHTL returns to its sub-translation generation and validation (§III-A1) step to generate new decompositions and TL sub-specifications for the modified node. This is followed by translation search (and culprit identification if needed) (§III-B). After editing a TL sub-specification or the DUT, SYNTHTL returns to its translation search step.

by default; however, it can be configured to return all TL specifications that hold.

If no candidate sub-translation tree produces a TL specification that holds on the DUT, SYNTHTL's model checker-assisted *culprit identification* utility (§IV-B) outputs a set of the least constrained trees/specifications among them. Rather than require the user to manually analyze each failing specification, culprit identification flags a subset of the nodes in their corresponding sub-translation trees, which may be relevant for the inconsistency (red nodes in Figure 3).

C. Fixing Culprit Sub-Translations or the DUT

Once culprit nodes have been identified (§III-B), the oracle can elect to fix the DUT, the NL/TL sub-specifications within culprit nodes, and/or decompositions of culprit nodes. After applying a fix, the SYNTHTL procedure returns back to and repeats an earlier analysis phase as follows. If the oracle modifies an NL sub-specification or decomposition of a culprit node, SYNTHTL goes back to its sub-translation and validation step (§III-A1) to interactively (with the oracle) propagate the these changes by re-generating TL sub-specifications and decompositions for edited nodes. Figure 4 demonstrates examples for which a decomposition is modified (left) and NL sub-specification is modified (middle). If the oracle modifies the DUT or a TL sub-specification, SYNTHTL returns back to its translation search step (§III-B) to check if the edits give rise to a full TL specification that holds on the DUT. Figure 4 illustrates an example in which a TL sub-specification is modified (right).

Algorithm 1 Translation Search Algorithm

1: **function** TSEARCH(n, S_{IN}, mode)
2: res = {}
3: **for** decomposition in getNodeDcmps(n, S_{IN}) **do**
4: n.setChildren(decomposition)
5: **if** mode = LC **then** ▷ get TL sub-specifications for n
6: subTLSet = getLCNodeSubTL(n, S_{IN}) ▷ get LC TL for n
7: **else**
8: subTLSet = getAllNodeSubTL(n, S_{IN}) ▷ get all TL for n
9: **for** subTLSpec in subTLSet **do**
10: n.setSubTL(subTLSpec)
11: **if** mode $\neq$ LC **then** ▷ Check least constrained trees first
12: $C_n = \{T \mid T \in S_{IN} \wedge t_n \in T\}$ ▷ trees that contain t_n
13: C = TSEARCH(Root(t_n), C_n, LC) ▷ get LC trees
14: $C_{\text{DUT}} = \{T \mid T \in C \wedge T \models \text{DUT}\}$ ▷ check LC trees
15: **if** mode = LC $\vee$ $C_{\text{DUT}} \neq \emptyset$ **then** ▷ Recurse to children of n
16: cList = [TSEARCH(child, S_{IN}, mode) for child in n]
17: **for** $c1, c2, \ldots$ in crossProduct(cList) **do**
18: n' = copy(n).setChildren($c1, c2, \ldots$)
19: res.add(n')
20: **return** res

Algorithm 2 Batch Model Checking Algorithm

1: **function** BATCHMC($tSet$, DUT)
2: H_{clause} = hashTable() ▷ Initialize hash table, mapping clause to trees
3: **for** T **in** $tSet$ **do**
4: **for** clause **in** getCNF(T) **do**
5: H_{clause}[clause].add(T) ▷ Add all CNF clauses to hash table
6: **while** $|H_{\text{clause}}| > 0$ **do** ▷ Loop while clauses left to check
7: $D_{\text{clause}} = \{\}$ ▷ Clauses checked in current iteration
8: $D_T = \{\}$ ▷ Trees that do not hold on DUT in current iteration
9: **for** clause **in** H_{clause} **do**
10: D_{clause}.add(clause)
11: **if** clause $\not\models$ DUT **then** ▷ Check if clause holds on DUT
12: $D_T = H_{\text{clause}}$[clause] ▷ found inconsistent clause
13: **break**
14: **for** clause **in** H_{clause} **do** ▷ Update trees in hash table
15: H_{clause}[clause] = H_{clause}[clause] $- D_T$
16: **if** $|H_{\text{clause}}\text{[clause]}| == 0$ **then**
17: D_{clause}.add(clause) ▷ Remove clause if all trees not hold
18: $tSet = tSet - D_T$ ▷ Remove trees inconsistent with DUT
19: $H_{\text{clause}} = H_{\text{clause}} - D_{\text{clause}}$ ▷ Remove clauses already checked
20: **return** $tSet$

IV. SYNTHTL UTILITIES: MODEL CHECKER-GUIDED TRANSLATION SEARCH AND CULPRIT IDENTIFICATION

We now provide more details on SYNTHTL's model checker-guided utilities: translation search (§IV-A), which features sub-tree pruning (§IV-A1) and batch model checking (§IV-A2), and culprit identification (§IV-B).

A. Translation Search Utility

SYNTHTL's translation search procedure takes as input a set of sub-translation trees. Let $S_{IN} = \{T_1, T_2, ...\}$ denote the input set of trees, each of which represents a TL specification (i.e., a TL formula). Given S_{IN}, translation search outputs the set $S_{OUT} \subseteq S_{IN}$. If one or more input TL specifications hold on the DUT, S_{OUT} contains the *most constrained* TL specifications that hold on the DUT. If none hold (i.e., S_{OUT} is empty), SYNTHTL employs culprit identification (§IV-B). Note that there can be multiple most/least constrained TL specifications, because in general, TL specifications may not be strict supersets or subsets of one another.

Concretely, translation search returns:

$S_{\text{OUT}} = \{T \mid T \in S_{IN} \wedge T \models \text{DUT} \wedge \forall i, T_i \in S_{IN} \wedge (T \neq T_i \rightarrow T \not\models T_i)\}$

1) Sub-Tree Pruning in Translation Search: SYNTHTL's translation search coordinates the instantiation of candidate sub-translation trees that result from sub-translation tree generation and validation (§III-A1). To obtain the set of all candidate sub-translation trees, one could naively take the cross product of all possible decompositions and TL sub-specifications (§III-A2). However, translation search avoids exhaustively constructing and checking all TL specifications by incrementally constructing these candidate trees (i.e., incrementally constructing the cross product) node by node starting at the root and preemptively pruning candidate trees before they are fully generated. The pseudocode for this procedure, called TSEARCH, is shown in Algorithm 1 and detailed below. Translation search uses TSEARCH to obtain a pruned set of possible TL specifications, and then checks them against the DUT to obtain S_{OUT}.

TSEARCH starts from a root node n, input tree set S_{IN}, and mode $\neq LC$; the mode variable indicates whether TSEARCH should output all sub-translation trees that may hold on the DUT (mode $\neq LC$), or the least constrained (LC) set of trees among all possible (mode $= LC$). TSEARCH constructs all decompositions (line 3) and all TL sub-specifications (line 9) of node n and conditionally recurses to each of its child nodes (line 15). Given the recursion condition is true, the set of all possible trees that contain node n is obtained through a cross product of each possible child sub-tree of node n (§III-A2, line 16 - 19).

Suppose node n has just been instantiated, and let t_n denote a partially constructed sub-tree up to and including n. TSEARCH only recurses to n's child nodes along a particular decomposition if it is possible to extend sub-tree t_n to a TL specification that holds on the DUT (lines 11 - 14). TSEARCH determines the recursion condition by constructing all of the least constrained (mode $= LC$) TL specifications that extend sub-tree t_n (lines 12 - 13), and model checking them against the DUT (line 14). If none of these least constrained TL specifications hold on the DUT, then no other trees that extend sub-tree t_n can hold, so TSEARCH can prune them from consideration (and avoid explicitly checking them against the DUT) by terminating recursion.

Note that constructing the set of least constrained TL specifications with sub-tree t_n (lines 12 - 13) does not require exhaustively constructing all trees with sub-tree t_n since TSEARCH with mode $= LC$ (line 13) incrementally constructs least constrained trees through only considering the least constrained TL sub-specifications for each node (line 6). Also, our implementation of Algorithm 1 makes use of memoizing results to avoid redundant recursive calls (not shown).

2) Batch Model Checking: After obtaining the pruned set of sub-translation trees, SYNTHTL deploys a novel batch model checking utility to improve performance of identifying TL

Algorithm 3 Conjunctive Clause Extraction Algorithm

```
1: function EXTRACTCLAUSES(T, n)
2:     n.setChildren({})
3:     n.setSubTL(id)
4:     Φ_T = GetTLSpec(T)
5:     conjSet = getCNF(Φ_T)
6:     return {c | c ∈ conjSet if id ∈ c}
```

specifications that hold on the DUT. Batch model checking, as shown in Algorithm 2, takes as input a DUT, and a set of sub-translation trees $tSet$, and efficiently obtains the subset of these sub-translation trees that hold on the DUT. Batch model checking exploits common conjunctive clauses among the set of TL specifications, so that each conjunctive clause is checked at most once across all trees.

First, conjunctive clauses for each sub-translation tree are discovered by transforming its corresponding TL specification into a conjunctive form (line 4). Second, to identify common conjunctive clauses among the trees in $tSet$, a hash table is used to map clauses to the sub-translation trees that contain them (lines 2 to 5). Third, as long as the hash table is non-empty, the algorithm iteratively selects clauses to check against the DUT with a model checker (lines 6 to 19). Whenever a clause is found to not hold on the DUT (line 11), all of its associated sub-translation trees and clauses are pruned from the hash table (line 14 to 19).

B. Culprit Identification Utility

Suppose that translation search (§III-B) determines that no TL specifications—from all those produced during sub-translation tree generation and validation (§III-A1)—hold on the DUT. At this point, SYNTHTL deploys culprit identification to identify nodes within the least constrained TL specifications (among all generated TL specifications) that are possible culprits for (i.e., possibly contributing to) their inconsistencies with the DUT. Culprit identification obtains the set of least constrained TL specification using TSEARCH (Algorithm 1, §IV-A) with mode $= LC$ starting from the root node, given the input set of all possible trees from sub-translation tree generation and validation. A node is a possible culprit if it contributes to at least one conjunctive clause in a full TL specification that does not hold on the DUT.

To identify all possible culprit nodes, within a sub-translation tree T that does not hold on the DUT, SYNTHTL first extracts for each node n of T the set of conjunctive clauses in the full TL specification that it contributes to using Algorithm 3. First, the decomposition of node n is set to empty (line 2) and its TL sub-specification is set to a special identifier id (line 3). Then, from this new variant of T, a full TL specification is constructed (line 4) and transformed into a conjunctive form (line 5). Finally, the clauses which contain id are returned (line 6). SYNTHTL then checks if each of the clauses returned by Algorithm 3 hold on the DUT.

This approach ensures that all nodes that are responsible for a TL specification's inconsistency with the DUT are flagged as possible culprits. However, not all nodes flagged as possible culprits are true positives, since non-culprits can contribute to failing clauses. To reduce false positives in culprit identification, SYNTHTL applies a heuristic filter to prioritize nodes which are more likely to be true culprits. Instead of flagging a node as a culprit if it contributes to at least one failing clause, the heuristic filter only marks a node as a culprit if all clauses it contributes to fail. Our evaluation demonstrates this heuristic significantly improves the precision of culprit identification (fewer false positives) and identifies true culprits while retaining high accuracy (few false negatives, §V-D).

V. CASE STUDY: TRANSLATING AN INDUSTRIAL NL SPECIFICATION TO TL WITH SYNTHTL

As a case study, we use SYNTHTL to translate three real-world NL specifications comprising the Arm AMBA AHB bus protocol [15], [16] to TL. In doing so, we evaluate SYNTHTL's sub-translation tree generation and validation (§III-A1), translation search (§IV-A), and culprit identification (§IV-B) components. Moreover, we answer the following questions: How successful are LLMs in generating sub-translation trees and to what degree do sub-translation trees reduce manual effort in validating TL specifications compared to existing methods (§V-A)? How efficient is translation search compared to exhaustive search (§V-B)? To what degree does culprit identification localize inconsistencies with the DUT to particular sub-translations (§V-D)?

Prototype Implementation Our SYNTHTL prototype consists of ∼3K lines of Python code and queries the open-source LTL model checker, Spot [23]. Although our prototype implementation of SYNTHTL deals with LTL, the SYNTHTL approach can be used to generate formulas in other TLs (e.g., SVA [3], PSL [4], STL [24], and so on). We have published our code (including LLM prompts), the AMBA AHB benchmarks (taken from prior work [16]), and example generated outputs in a public repository.[1]

Benchmarks We evaluate SYNTHTL on three real-world NL hardware specifications, taken from the AMBA AHB bus protocol [15], [16], corresponding to its *arbiter*, *controller*, and *worker* modules. We select these NL specifications for our experiments for two reasons. First, AMBA AHB is an industrial protocol that is widely deployed in modern SoCs, including Arm Cortex-M based designs [25], [26]. Second, it has already been manually formalized in prior work [16], giving us "ground truth" TL specifications to use in qualitatively assessing SYNTHTL's outputs. The original TL specification is written in PSL [16], and we manually rewrite it in LTL since our prototype implementation of SYNTHTL uses LTL model checking.

For each of the three AMBA AHB hardware modules we consider, Table I gives the sizes of their published NL specifications and corresponding ground truth TL specifications [16]. These TL specifications serve the role of "golden" DUTs in our evaluation.

Large Language Models We conduct our evaluation with two state-of-the-art LLMs: GPT3.5 and GPT4 [14].

[1] https://github.com/dmmendo/SynthTL

Module	Controller	Worker	Arbiter
NL Size (words)	436	268	643
TL Size (symbols)	460	351	490

TABLE I: Specification sizes used in SYNTHTL's evaluation.

A. Evaluation of TL Generation and Validation

We compare SYNTHTL's sub-translation tree generation and validation (§III-A1) to nl2spec [6], the existing state-of-the-art approach, in terms of manual effort required to produce the correct TL specification.

Setup In this experiment, We query the LLM $D = K = 3$ times to generate decompositions and TL sub-specifications for each node with SYNTHTL. The oracle selects one correct TL sub-specification and one correct decomposition for each sub-translation with SYNTHTL and provides a correct TL sub-specification/decomposition if none of the LLM-generated options are correct.

Baseline Given an input NL specification, nl2spec [6] decomposes it into sub-translations and produces an output TL specification from the sub-translations, all using an LLM. The user can iteratively edit sub-translations and request that nl2spec re-generate (with an LLM) the output TL specification from them until the user decides the TL specification is correct.

Unlike SYNTHTL, nl2spec's sub-translations are *unstructured* (i.e., they are not organized as a sub-translation tree), and so they cannot be mechanically composed into a full TL specification. Instead, nl2spec uses an LLM to perform this composition in one shot. However, this approach renders nl2spec susceptible to generating inaccurate TL specifications, even if the user has decided that all sub-translations are correct.

In terms of oracle effort, both nl2spec and SYNTHTL require sub-translation validation. However, nl2spec additionally requires oracle validation of full output TL specifications, while SYNTHTL additionally asks the oracle to validate (comparatively much simpler) sub-translation decompositions.

We consider an *ideal* nl2spec as our baseline, which is initially (in its first iteration) given a correct set of sub-translations taken from the leaf nodes of a correct "reference" SYNTHTL sub-translation tree. Thus, nl2spec need only compose these sub-translations to produce a correct full TL specification using an LLM. If the full TL specification generated by nl2spec in an iteration is incorrect, the oracle provides new sub-translations in the next iteration that compose the current sub-translations by instantiating them within the context of their parents' TL sub-specifications in the reference sub-translation tree. For example, consider a reference sub-translation tree representing TL specification $(A \wedge B) \rightarrow (C \wedge D)$ with two leaf nodes whose TL sub-specifications are $A \wedge B$ and $C \wedge D$. The TL sub-specification of the root node defines the implication ($\rightarrow$) between the leaf nodes. In the first iteration, nl2spec is given the leaf nodes. If nl2spec does not output the correct TL specification in the first iteration, the oracle provides $(A \wedge B) \rightarrow (C \wedge D)$ in the second iteration. Thus, the sub-translations provided by the oracle become fewer and more-complex/coarse-grained in each iteration of nl2spec, approaching the full reference TL specification. nl2spec continues iterating either until it generates the correct full TL specification by composing sub-translations with an LLM or until the oracle supplies the full TL specification as input (by composing sub-translations from the previous failed iteration). The LLM is given three tries (i.e., is queried three times) in each iteration. Incorrect sub-translations are discarded between iterations.

Metrics For both SYNTHTL and our ideal nl2spec baseline, we record the following:

- Number of unique generated full TL specifications (**Space Size**).
- Number of TL sub-specifications and decompositions inspected by the oracle (**Inspections**)
- Number of TL sub-specifications and decompositions edited by the oracle (**Trns Edit** and **Dcmp Edit**). For each node, the oracle only edits a decomposition or TL sub-specification if the LLM does not generate one that is deemed valid by the oracle.
- Number of nodes in final sub-translation tree (**Tree Size**).
- Size of formulas inspected (**Inspect Size**) and edited (**Edit Size**) by the oracle in number of symbols. A symbol is an LTL operator or variable. We normalize this quantity by the size of the ground truth TL specification (from the handwritten TL specification in prior work [16]).

Results The results are shown in Table II. Across all six SYNTHTL experiments (three AMBA AHB modules and two LLM options), SYNTHTL generates up to 9.33e17 and as few as 8.29e4 unique TL specifications. This result demonstrates that the input NL specifications for modules of the Arm AMBA AHB bus protocol are highly ambiguous, despite being carefully crafted so as to be amenable to formalization [16].

SYNTHTL produces a correct TL specification, with the oracle inspecting and editing TL sub-specifications that are on average 2.43% (up to 20.9%) and 2.95% (up to 8.9%) the size of the ground truth TL specification, respectively. Further, the oracle edits on average 12.9% (up to 20.6%) and 37.5% (up to 44.8%) the number of decompositions and TL sub-specifications, respectively, that exist in the final sub-translation tree. That is, most decompositions and sub-translations in the final correct sub-translation tree are automatically generates by the LLM. These results demonstrate that SYNTHTL requires significantly less manual effort than both manual end-to-end NL to TL translation and full TL specification validation.

In all six nl2spec experiments, our ideal nl2spec baseline fails to produce the correct TL specification. It incorrectly composes TL specifications despite being given correct the sub-translations in every iteration. In all cases, the size of the inspected and edited sub-specifications is significantly smaller with SYNTHTL compared to nl2spec. This result demonstrates that SYNTHTL enables LLM-based NL to TL translation to handle large and complex NL specifications for the first time. The maximum, average, and total size of

Arm AMBA AHB Module	Controller				Worker				Arbiter			
Large Language Model	GPT3.5		GPT4		GPT3.5		GPT4		GPT3.5		GPT4	
Translation Approach	SYNTHTL	nl2spec	SYNTHTL	nl2spec	SYNTHTL	nl2spec	SYNTHTL	nl2spec	SYNTHTL	nl2spec	SYNTHTL	nl2spec
Space Size	7.26e16	5	8.29e4	5	3.05e10	6	2.99e6	6	9.33e17	6	5.97e8	6
Inspections	96	125	92	77	94	90	81	66	107	90	117	94
Trns Edit	18	18	17	18	21	20	16	21	26	22	25	23
Dcmp Edit	11	18	12	18	2	20	0	21	8	22	12	23
Tree Size	56		58		48		47		58		61	
Sum Inspect Size	1.515	9.124	1.254	6.898	1.558	11.538	1.197	8.077	2.506	5.543	2.157	8.682
Avg Inspect Size	0.021	0.073	0.020	0.090	0.027	0.128	0.023	0.122	0.028	0.062	0.027	0.092
Max Inspect Size	0.209	1.020	0.150	1	0.142	1.302	0.202	1	0.127	1	0.120	1
Sum Edited Size	0.541	2.585	0.546	2.585	0.627	2.823	0.595	3	0.606	2.773	0.629	2.804
Avg Edited Size	0.030	0.144	0.032	0.144	0.030	0.141	0.037	0.143	0.023	0.126	0.025	0.122
Max Edited Size	0.089	1	0.089	1	0.177	1	0.177	1	0.084	1	0.084	1
Gen Correct?	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗

TABLE II: Generating and validating the AMBA AHB TL specification with SYNTHTL versus nl2spec. Formula size is normalized by the size of the full ground truth specification.

Module	Worker		Controller		Arbiter	
	K=2	K=3	K=2	K=3	K=2	K=3
Exhaustive	4096	5832	4096	3888	4096	2916
SYNTHTL	505	376	2052	804	548	184
% Pruned	87.7	93.6	49.9	79.3	86.6	93.7

TABLE III: Exhaustive vs. SYNTHTL's translation search

Module	Arbiter	Controller	Worker
Formulas	128	128	10000
Clauses	640	62	1512
Variables	19	29	24
Exhaustive (s)	28075.69	10025.26	462.38
SYNTHTL (s)	7622.44	2975.77	13.09
Speedup	3.68	3.37	35.32

TABLE IV: Batch vs. Exhaustive Model Checking

inspected TL sub-specifications is on average 5.32×, 3.94×, 6.97× smaller with SYNTHTL compared to nl2spec, respectively. The maximum, average, and total size of edited TL sub-specifications is on average 4.68×, 4.68×, 9.61× smaller with SYNTHTL compared to nl2spec, respectively. These results show that SYNTHTL's sub-translation trees significantly reduce the manual effort required to inspect and fix sub-translations compared to nl2spec.

Takeaway SYNTHTL significantly reduces manual effort required to fix and validate LLM-generated TL specifications compared to prior approaches and enables automatic generation of large and complex TL specifications.

B. Evaluation of Translation Search

Next, we evaluate the efficiency of translation search (§IV-A) in discovering a correct TL specification among a set of sub-translation trees, given a correctly implemented DUT (i.e., the ground truth TL specification produced in prior work [16]). To generate a set of sub-translation trees, we direct SYNTHTL to conduct sub-translation tree generation and validation (§III-A) and limit the oracle to specifying one correct decomposition and $K = 2, 3$ TL sub-specifications per node, where at least one TL sub-specification is correct. Note that querying the LLM multiple times for TL sub-specifications/decompositions may produce equivalent TL sub-specifications/decompositions and, in such situations, duplicates are discarded. To evaluate under multiple settings, we also limit the oracle to only provide multiple TL sub-specifications for a node if the tree space size would be less than 2^{13}.

Results Table III shows the number of TL specifications generated and checked on the DUT with SYNTHTL's translation search utility (SYNTHTL row) versus exhaustively searching all trees in the input set (Exhaustive row), i.e., the maximum number of TL specifications to check on the DUT. In all cases, translation search explores significantly fewer sub-translation trees than the exhaustive approach to find the correct TL specification. SYNTHTL prunes as many as 93.6%, 79.3%, 94.7% of the search space for the arbiter, controller, and worker, respectively.

Takeaway SYNTHTL's translation search greatly improves the scalability of model checking many translation possibilities for a given NL specification through effective sub-tree pruning.

C. Evaluation of Batch Model Checking

We now evaluate the efficacy of SYNTHTL's batch model checking optimization (§IV-A2) in accelerating model checking many TL specifications against a DUT.

Results Table IV shows runtimes of exhaustive and batch model checking for a set of sub-translation trees generated by an LLM with SYNTHTL's sub-translation generation. Batch model checking is 3.68×, 3.37×, 35.32× faster compared to exhaustive model checking for the arbiter, controller, and worker, respectively.

Takeaway Batch model checking significantly accelerates model checking large sets of TL specifications.

D. Evaluation of Culprit Identification

We now evaluate the effectiveness of SYNTHTL's culprit identification in localizing inconsistencies with the DUT to particular sub-translations (§IV-B) when a sub-translation tree contains an incorrect sub-translation (§V-D1), and when the DUT is incorrectly implemented (§V-D2). Note that AC refers to culprit identification with no heuristic filters, and FC refers to the approach with the heuristic filter (§IV-B).

Module	Controller						Worker						Arbiter					
Bug	G		X		$\neg$		G		X		$\neg$		G		X		$\neg$	
Ex.	13		35		28		16		41		53		11		23		30	
App.	AC	FC	AC	FC	AC	FC	AC	FC	AC	FC	AC	FC	AC	FC	AC	FC	AC	FC
% Clpt	57.0	34.5	67.0	45.2	58.0	32.0	68.9	47.4	70.7	50.1	65.7	40.4	52.23	22.1	54.7	24.2	58.3	31.1
Recall	1	0.92	1	0.97	1	1	1	1	1	1	1	0.96	1	0.91	1	0.91	1	1

TABLE V: Culprit Identification given an incorrect AMBA AHB TL specification

Module	Controller						Worker						Arbiter					
Bug	G		X		$\neg$		G		X		$\neg$		G		X		$\neg$	
Ex.	3		8		10		16		42		50		2		9		10	
App.	AC	FC	AC	FC	AC	FC	AC	FC	AC	FC	AC	FC	AC	FC	AC	FC	AC	FC
% Clpt	42.5	6.3	41.4	5.2	53.1	24.1	59.7	30.7	61.2	33.8	67.4	44.4	42.9	6.4	48.7	16.1	53.8	24.4
Recall	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

TABLE VI: Culprit Identification given an incorrect AMBA AHB DUT

1) Culprit Identification Given an Incorrect TL Specification and a Correct DUT: To emulate a sub-translation tree that contains an incorrect sub-translation that causes the full TL specification to not hold on a correctly implemented DUT, we insert bugs into a correct sub-translation tree. Given a correct sub-translation tree, we randomly select a node, and change the sub-specification by adding an operator (i.e., either G, X, or $\neg$). If the perturbation causes the TL specification to not hold on the DUT, then we run culprit identification and record the percentage of nodes flagged as possible culprits and if the set of possible culprits contains the true culprit.

Results Table V shows the number examples evaluated per perturbation, the average percentage of nodes that are marked as possible culprits per perturbation, and the fraction of examples where the possible culprit set contained the true culprit (recall). AC flags on average 56.8% of the nodes, and the identified possible culprit set always contains the true culprit (i.e., recall is always 1). FC flags 38.5% of the nodes (fewer than AC), however, due to its heuristic nature, catches the true culprit in 97.1% of all cases (fewer than AC).

In Figure 5, we show the percentage of nodes in the sub-translation tree that are marked as possible culprits versus the percentage of true culprits in the sub-translation tree when perturbing randomly selected sub-translations with incorrect LLM-generated sub-specifications. Note that the percentage of true culprits does not go to 100%, because the LLM-generated sub-specifications for the remaining set of nodes do not cause inconsistencies with the DUT. In all cases both AC and FC find all the true culprits. The percentage of nodes flagged as possible culprits increases with the percentage of true culprits in the sub-translation tree. AC flagged as few as 49.1%, 37.9%, 38.1% and as high as 90.6%, 96.6%, 92.1% for the controller, worker, and arbiter, respectively. FC flagged as few as 15.1%, 5.2%, 4.8% and as high as 90.6%, 96.6%, 92.1% for the arbiter, controller and worker, respectively.

2) Culprit Identification Given a Correct TL Specification and an Incorrect DUT: We evaluate the efficacy of culprit identification in localizing inconsistencies with the DUT to particular sub-translations, given a correct TL specification that does not hold on a buggy DUT. To emulate bugs in the DUT, we first construct a correct sub-translation tree for each module by manually transforming the ground truth LTL specification [16] into one. Then, we add an operator (i.e., either G, X, or $\neg$) in a randomly selected a sub-translation and use the perturbed TL specification as the DUT.

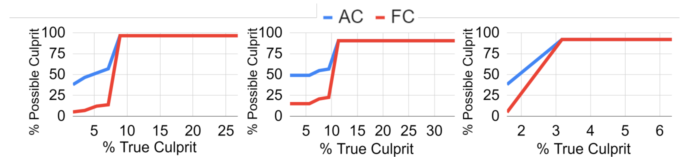

Fig. 5: Percent flagged possible culprits vs. true culprits of an AMBA AHB controller (left), worker (middle), arbiter (right).

Results Table VI shows what percent of nodes in the sub-translation tree are marked as possible culprits. Both AC and FC find all true culprits in all examples. AC/FC flag on average 59.7%/32.2% of nodes as possible culprits, respectively.

Takeaway Culprit identification significantly reduces manual effort in localizing inconsistencies with the DUT to particular parts of an NL specification and TL specification.

VI. Conclusion

Typical translation of unstructured NL to TL is *unstructured*, requiring users to manually inspect/correct complex TL outputs. Instead, SynthTL conducts *structured translation* of unstructured NL to TL, which enables users to exclusively validate simple TL sub-specifications and decompositions that mechanically compose to produce a TL output. Plus, structured translation enables LLMs, model checkers, and human users to meaningfully collaborate on an NL to TL translation task.

Acknowledgment

We thank Mohammad Rahmani Fadiheh, Haoze Wu, and the anonymous reviewers for their constructive comments and feedback. This work was supported in part by the National Science Foundation (NSF), under awards 2153936 and 2236855 (CAREER); the Defense Advanced Research Projects Agency (DARPA) under contract W912CG-23-C-0025 and subcontract from Galois, Inc.; and by the German Federal Ministry of Education and Research (BMBF), through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 16KIS1138).

References

[1] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*. Cambridge, MA, USA: MIT Press, 2000.

[2] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pp. 46–57, 1977.

[3] S. Vijayaraghavan and M. Ramanathan, "A practical guide for system verilog assertions," 2005.

[4] "Ieee standard for property specification language (psl)," *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)*, pp. 1–182, 2010.

[5] H. Foster, "Part 1: The 2022 wilson research group functional verification study," Jan 2023.

[6] M. Cosler, C. Hahn, D. Mendoza, F. Schmitt, and C. Trippel, "nl2spec: Interactively translating unstructured natural language to temporal logics with large language models," in *Computer Aided Verification* (C. Enea and A. Lal, eds.), (Cham), pp. 383–396, Springer Nature Switzerland, 2023.

[7] I. Gavran, E. Darulova, and R. Majumdar, "Interactive synthesis of temporal specifications from examples and natural language," *Proc. ACM Program. Lang.*, vol. 4, nov 2020.

[8] F. Fuggitti and T. Chakraborti, "Nl2ltl - a python package for converting natural language (nl) instructions to linear temporal logic (ltl) formulas," in *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'23/IAAI'23/EAAI'23, AAAI Press, 2023.

[9] Y. Chen, R. Gandhi, Y. Zhang, and C. Fan, "NL2TL: Transforming natural languages to temporal logics using large language models," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Dec. 2023.

[10] J. He, E. Bartocci, D. Ničković, H. Isakovic, and R. Grosu, "Deepstl: from english requirements to signal temporal logic," in *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, (New York, NY, USA), p. 610–622, Association for Computing Machinery, 2022.

[11] C. Hahn, F. Schmitt, J. J. Tillman, N. Metzger, J. Siber, and B. Finkbeiner, "Formal specifications from natural language," 2022.

[12] J. X. Liu, Z. Yang, B. Schornstein, S. Liang, I. Idrees, S. Tellex, and A. Shah, "Lang2LTL: Translating natural language commands to temporal specification with large language models," in *Workshop on Language and Robotics at CoRL 2022*, 2022.

[13] C. Wang, C. Ross, Y.-L. Kuo, B. Katz, and A. Barbu, "Learning a natural-language to ltl executable semantic parser for grounded robotics," in *Proceedings of the 2020 Conference on Robot Learning* (J. Kober, F. Ramos, and C. Tomlin, eds.), vol. 155 of *Proceedings of Machine Learning Research*, pp. 1706–1718, PMLR, 16–18 Nov 2021.

[14] OpenAI, J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, R. Avila, I. Babuschkin, S. Balaji, V. Balcom, P. Baltescu, H. Bao, M. Bavarian, J. Belgum, I. Bello, J. Berdine, G. Bernadett-Shapiro, C. Berner, L. Bogdonoff, O. Boiko, M. Boyd, A.-L. Brakman, G. Brockman, T. Brooks, M. Brundage, K. Button, T. Cai, R. Campbell, A. Cann, B. Carey, C. Carlson, R. Carmichael, B. Chan, C. Chang, F. Chantzis, D. Chen, S. Chen, R. Chen, J. Chen, M. Chen, B. Chess, C. Cho, C. Chu, H. W. Chung, D. Cummings, J. Currier, Y. Dai, C. Decareaux, T. Degry, N. Deutsch, D. Deville, A. Dhar, D. Dohan, S. Dowling, S. Dunning, A. Ecoffet, A. Eleti, T. Eloundou, D. Farhi, L. Fedus, N. Felix, S. P. Fishman, J. Forte, I. Fulford, L. Gao, E. Georges, C. Gibson, V. Goel, T. Gogineni, G. Goh, R. Gontijo-Lopes, J. Gordon, M. Grafstein, S. Gray, R. Greene, J. Gross, S. S. Gu, Y. Guo, C. Hallacy, J. Han, J. Harris, Y. He, M. Heaton, J. Heidecke, C. Hesse, A. Hickey, W. Hickey, P. Hoeschele, B. Houghton, K. Hsu, S. Hu, X. Hu, J. Huizinga, S. Jain, S. Jain, J. Jang, A. Jiang, R. Jiang, H. Jin, D. Jin, S. Jomoto, B. Jonn, H. Jun, T. Kaftan, Łukasz Kaiser, A. Kamali, I. Kanitscheider, N. S. Keskar, T. Khan, L. Kilpatrick, J. W. Kim, C. Kim, Y. Kim, J. H. Kirchner, J. Kiros, M. Knight, D. Kokotajlo, Łukasz Kondraciuk, A. Kondrich, A. Konstantinidis, K. Kosic, G. Krueger, V. Kuo, M. Lampe, I. Lan, T. Lee, J. Leike, J. Leung, D. Levy, C. M. Li, R. Lim, M. Lin, S. Lin, M. Litwin, T. Lopez, R. Lowe, P. Lue, A. Makanju, K. Malfacini, S. Manning, T. Markov, Y. Markovski, B. Martin, K. Mayer, A. Mayne, B. McGrew, S. M. McKinney, C. McLeavey, P. McMillan, J. McNeil, D. Medina, A. Mehta, J. Menick, L. Metz, A. Mishchenko, P. Mishkin, V. Monaco, E. Morikawa, D. Mossing, T. Mu, M. Murati, O. Murk, D. Mély, A. Nair, R. Nakano, R. Nayak, A. Neelakantan, R. Ngo, H. Noh, L. Ouyang, C. O'Keefe, J. Pachocki, A. Paino, J. Palermo, A. Pantuliano, G. Parascandolo, J. Parish, E. Parparita, A. Passos, M. Pavlov, A. Peng, A. Perelman, F. de Avila Belbute Peres, M. Petrov, H. P. de Oliveira Pinto, Michael, Pokorny, M. Pokrass, V. H. Pong, T. Powell, A. Power, B. Power, E. Proehl, R. Puri, A. Radford, J. Rae, A. Ramesh, C. Raymond, F. Real, K. Rimbach, C. Ross, B. Rotsted, H. Roussez, N. Ryder, M. Saltarelli, T. Sanders, S. Santurkar, G. Sastry, H. Schmidt, D. Schnurr, J. Schulman, D. Selsam, K. Sheppard, T. Sherbakov, J. Shieh, S. Shoker, P. Shyam, S. Sidor, E. Sigler, M. Simens, J. Sitkin, K. Slama, I. Sohl, B. Sokolowsky, Y. Song, N. Staudacher, F. P. Such, N. Summers, I. Sutskever, J. Tang, N. Tezak, M. B. Thompson, P. Tillet, A. Tootoonchian, E. Tseng, P. Tuggle, N. Turley, J. Tworek, J. F. C. Uribe, A. Vallone, A. Vijayvergiya, C. Voss, C. Wainwright, J. J. Wang, A. Wang, B. Wang, J. Ward, J. Wei, C. Weinmann, A. Welihinda, P. Welinder, J. Weng, L. Weng, M. Wiethoff, D. Willner, C. Winter, S. Wolrich, H. Wong, L. Workman, S. Wu, J. Wu, M. Wu, K. Xiao, T. Xu, S. Yoo, K. Yu, Q. Yuan, W. Zaremba, R. Zellers, C. Zhang, M. Zhang, S. Zhao, T. Zheng, J. Zhuang, W. Zhuk, and B. Zoph, "Gpt-4 technical report," 2024.

[15] Arm Ltd., *AMBA AHB Protocol Specification*, 2021.

[16] Y. Godhal, K. Chatterjee, and T. A. Henzinger, "Synthesis of amba ahb from formal specification: a case study," *International Journal on Software Tools for Technology Transfer*, vol. 15, pp. 585 – 601, 2011.

[17] E. Conrad, L. Titolo, D. Giannakopoulou, T. Pressburger, and A. Dutle, "A compositional proof framework for fretish requirements," in *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2022, (New York, NY, USA), p. 68–81, Association for Computing Machinery, 2022.

[18] S. Konrad and B. H. C. Cheng, "Real-time specification patterns," *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pp. 372–381, 2005.

[19] L. Grunske, "Specification patterns for probabilistic quality properties," in *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, (New York, NY, USA), p. 31–40, Association for Computing Machinery, 2008.

[20] A. Brunello, A. Montanari, and M. Reynolds, "Synthesis of LTL formulas from natural language texts: State of the art and research directions," in *26th International Symposium on Temporal Representation and Reasoning, TIME 2019, October 16-19, 2019, Málaga, Spain* (J. Gamper, S. Pinchinat, and G. Sciavicco, eds.), vol. 147 of *LIPIcs*, pp. 17:1–17:19, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[21] 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," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, (Red Hook, NY, USA), Curran Associates Inc., 2020.

[22] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive nlp tasks," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, (Red Hook, NY, USA), Curran Associates Inc., 2020.

[23] A. Duret-Lutz, E. Renault, M. Colange, F. Renkin, A. G. Aisse, P. Schlehuber-Caissier, T. Medioni, A. Martin, J. Dubois, C. Gillard, and H. Lauko, "From Spot 2.0 to Spot 2.10: What's new?," in *Proceedings of the 34th International Conference on Computer Aided Verification (CAV'22)*, vol. 13372 of *Lecture Notes in Computer Science*, pp. 174–187, Springer, Aug. 2022.

[24] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems* (Y. Lakhnech and S. Yovine, eds.), (Berlin, Heidelberg), pp. 152–166, Springer Berlin Heidelberg, 2004.

[25] B. Walshe, "What is amba?," 2014.

[26] Arm Ltd., "Learn the architecture - an introduction to amba axi," 2022.