Automated Generation of LTL Specifications For Smart Home IoT Using Natural Language

Shiyu Zhang*, Juan Zhai*†, Lei Bu*(✉), Mingsong Chen‡, Linzhang Wang*, Xuandong Li*
*State Key Laboratory of Novel Software Technology, Department of Computer Science and Technology, Nanjing University
†Department of Computer Science, Rutgers University
‡Shanghai Key Lab of Trustworthy Computing, East China Normal University

***Abstract*—Ordinary users can build their smart home automation system easily nowadays, but such user-customized systems could be error-prone. Using formal verification to prove the correctness of such systems is necessary. However, to conduct formal proof, formal specifications such as Linear Temporal Logic (LTL) formulas have to be provided, but ordinary users cannot author LTL formulas but only natural language.**

To address this problem, this paper presents a novel approach that can automatically generate formal LTL specifications from natural language requirements based on domain knowledge and our proposed ambiguity refining techniques. Experimental results show that our approach can achieve a high correctness rate of 95.4% in converting natural language sentences into LTL formulas from 481 requirements of real examples.

I. Introduction

With the rapid development of Internet of Things (IoT), trigger-action programming (TAP) in the form of "If a trigger occurs, THEN an action will be performed" rules, is becoming increasingly popular. Although many IoT service providers build easy-to-use platforms for ordinary inexperienced users, they also pose safety and security issues, due to conflicts or complicate chained reactions among rules. Recently, it is an emerging trend to use the model checking technique to model and verify the correctness of user-customized IoT systems [1]–[5]. However, most of these works focus on how to model and verify the behavior of the system. For specifications, they either provide a shortlist of predefined routine specifications or assume users would provide such specifications by themselves.

Nevertheless, mastery of formal specifications requires a significant amount of training and expertise. For ordinary IoT users, writing formal specifications is way beyond their ability. They tend to express their requirements using natural language. Hence, it is necessary to automatically translate the natural language specifications into formal specifications to make the checking of the certain system possible.

Many research works have been devoted to the natural language based formal specification generation problem [6]–[8]. Most of the works give a strict limitation of the grammar of the input natural language to facilitate the translation [6], [7]. However, the expressiveness of such fixed grammar is very limited, and users need to put in extra efforts to learn it. There are also works trying to generate formal specifications from arbitrary natural languages [8], [9], but few of them give a satisfactory answer to the ambiguities problem, as the input sentences given by users are often not well-written.

To address the above challenges, we propose a new approach to translate arbitrary natural language specifications into LTL formulas in this paper. We utilize home automation (HA)-IoT domain knowledge and design specific ambiguity refining techniques to fix and adapt the error-prone parser tree of the original sentence. Then, we extract the semantic information from the parser tree, and build mappings between the tree and LTL formula accordingly. To the best of our knowledge, our work is the first to automatically translate natural language specifications from IoT users into LTL specifications.

II. Background

This paper attempts to fill the gap between natural language specifications with rigorous formal formulas in the domain of HA-IoT. The formal specification logic we use is linear temporal logic (LTL) [10], which is a modal temporal logic and widely used in various mature model checkers.

LTL formulas are built up from a set of atomic propositions (AP), the logical operators $\neg$ (negation), $\vee$ (disjunction) and temporal operators G (always), X (next), U (until), F (eventually). Formally, the set of LTL formulas used in our study are inductively defined according to the following grammar:

$$\phi ::= p|\neg p|\phi_1 \vee \phi_2|G\ \phi|X\ \phi|\phi_1\ U\ \phi_2|F\ \phi,\ \ p \in AP$$

Other logic connectives such as $\wedge$ (conjunction), $\rightarrow$ (implication) can be defined using the listed connectives. An LTL formula is satisfiable by a model if there exists a sequence of states in the model such that the formula is true from the initial state.

Considering the complexity of LTL, it is unreasonable to ask ordinary users, who may have no background in computer science, to master and author LTL formulas to express their requirements.

III. Motivating Example

Here, we use a motivating example to show the issues that this work focuses on. Imagine a home scenario consisting of 5 devices and sensors: alarm, gas sensor, temperature sensor, heater and cooler. These devices are connected by 3 user-authored TAP rules:

- R1: **IF** `GAS_CONCENTRATION > 10` **THEN** `ALARM.TurnOn`, which warns the owner when the gas leaks.

- R2: **IF** `COOLER.state = ON` **THEN** `HEATER.TurnOff`, which is designed for energy efficiency.
- R3: **IF** `TEMPERATURE < 15` **THEN** `HEATER.TurnOn`, which is designed to keep the room warm.

Additionally, we give the corresponding requirements/specifications that the homeowner hopes the HA-IoT system should satisfy:

Req-1 When the gas concentration exceeds 10%, the alarm should be activated immediately until the gas concentration is below 8%.

Req-2 The cooler in the bedroom and the heater in the living room should not be on simultaneously.

For normal HA-IoT users, it is not easy to tell directly whether these specifications can be satisfied by the rules. After we translate the requirements into LTL formulas manually, we utilize one of the HA-IoT correctness checker MenShen [1] to check the correctness of these systems, and find both the requirements are violated. **Req-2** is violated because rule R3 may turn on the heater even if it has been turned off by rule R2. The violation of **Req-1** is tricky that the activation of the **action** part may not be very prompt under different implementation of the HA-IoT infrastructure[1]. In this case, the system cannot guarantee to activate the alarm immediately when the gas concentration exceeds 10%.

If we do not ask users to express the specific requirements they want their system to satisfy, then do the corresponding checking, users may not notice such violation and the corresponding HA-IoT system may run improperly. However, it is impossible to ask ordinary HA-IoT users to write formal specifications for each requirement manually. Therefore, an automatic translation from natural language specification to LTL formula is a must in the domain of HA-IoT verification.

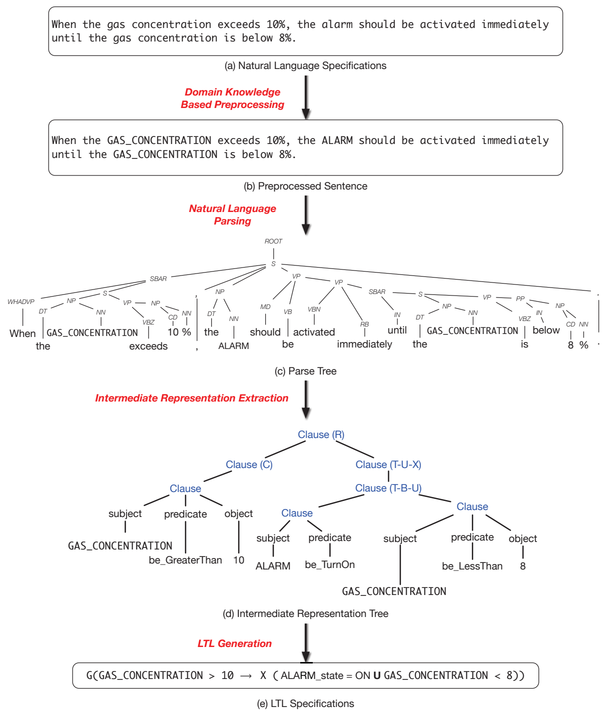
IV. Approach

A. Overview

In this section, we present our grammar based method which translates a natural language specification into a formal LTL formula. Fig. 1 gives an overview of the workflow. We take **Req-1** as an example to demonstrate our idea:

- Firstly, the original sentence is preprocessed based on the domain knowledge to facilitate the later processing.
- Secondly, an off-the-shelf NLP parser is leveraged to generate a preliminary parse tree of a sentence. Then, we present domain-related optimizations to eliminate the ambiguity and refine the tree.
- Semantic key clauses are identified in the parse tree to produce a tree-like intermediate representation (IR).
- Semantic templates are leveraged to generate an atomic proposition for each clause in the IR. Then we design an algorithm to combine the atomic propositions to a complete LTL formula. Finally, the LTL specification generated for **Req-1** is shown in Fig. 1(e).

[1]As reported in IFTTT.com, the time delay of the activation of the action could be 15 minutes in the worst case.



```
When the gas concentration exceeds 10%, the alarm should be activated immediately
until the gas concentration is below 8%.
```

```
When the GAS_CONCENTRATION exceeds 10%, the ALARM should be activated immediately
until the GAS_CONCENTRATION is below 8%.
```

```
G(GAS_CONCENTRATION > 10 → X ( ALARM_state = ON U GAS_CONCENTRATION < 8))
```

Fig. 1. Workflow Overview

B. Domain Knowledge Based Preprocessing

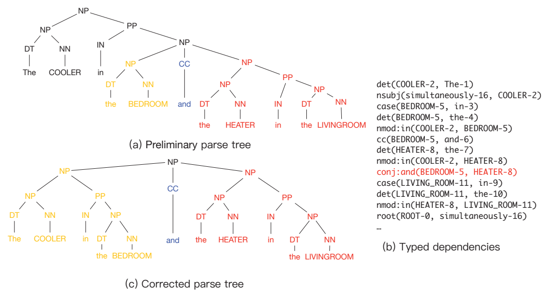
The specifications written in natural language use phrases or synonyms of domain specific terms, but traditional parser may give incorrect Part-Of-Speech (POS) tags or complex syntactic analysis results. In order to improve the accuracy of the parser, we first built a domain specific glossary that defines terms commonly used in IoT domain and LTL semantic expressions, as well as synonyms caught from WordNet [11].

Then we take a natural language sentence as input, replaces the phrases and synonyms with pre-defined domain specific terms (e.g. “gas concentration” → “GAS_CONCENTRATION”), and generates a preprocessed sentence (as shown in Fig. 1(b)).

This procedure can help our method to handle more flexible natural language expressions, eliminate word-sense ambiguity and make semantic understanding and mapping easier.

C. Natural Language Parsing

After the preprocessing step, we leverage the state-of-art well-known Stanford Parser [12], [13] to generate a preliminary parse tree of the sentence. However, it is widely recognized that the Stanford Parser may encounter two main classes of ambiguities including “attachment ambiguity” and “coordination ambiguity”. Clearly, a wrong parse tree will finally lead to a wrong LTL formula. To handle this problem, we make an investigation of typed dependencies generated by the Stanford Parser to locate the error and utilize the pre-defined domain related device information (e.g. device name, type, system variables) to fix the tree.



```
det(COOLER-2, The-1)
nsubj(simultaneously-16, COOLER-2)
case(BEDROOM-5, in-3)
det(BEDROOM-5, the-4)
nmod:in(COOLER-2, BEDROOM-5)
cc(BEDROOM-5, and-6)
det(HEATER-8, the-7)
nmod:in(COOLER-2, HEATER-8)
conj:and(BEDROOM-5, HEATER-8)
case(LIVING_ROOM-11, in-9)
det(LIVING_ROOM-11, the-10)
nmod:in(HEATER-8, LIVING_ROOM-11)
root(ROOT-0, simultaneously-16)
…
```

(a) Preliminary parse tree
(b) Typed dependencies
(c) Corrected parse tree
The COOLER in the BEDROOM and the HEATER in the LIVINGROOM

Fig. 2. Coordination ambiguity correction.

We take “coordination ambiguity” as an example to demonstrate our approach. In coordination ambiguity, different sets of phrases can be conjoined by a conjunction. We take **Req-2** as an example. Fig. 2 shows part of the preliminary analysis result where “BEDROOM” is wrongly conjoined with “HEATER”. However, from the device information, we can tell “HEATER” is a “Device”, but “BEDROOM” is a “Location”, they can not be conjoined to together. Therefore we find the nearest suitable “Device”, “COOLER”, and lift up the coordination node to connect “COOLER” and “HEATER”.

D. *Intermediate Representation Extracting*

Now, we introduce an intermediate representation (IR) extracting procedure to extract necessary semantic information form the parse tree for the later LTL generation.

We first identify the clauses in the parse tree. Then we extract semantic elements and match them to the domain specific terms in each clause. The task is performed by POS tags and lexical matching.

More specifically, a requirement always contains a main clause, which we call *effect*, and often have *condition*s which can trigger the *effect*. We introduce a special class: Clause (C) to indicate *condition* in the IR.

Temporal information is an important part of temporal logic. We classify the temporal descriptions in natural language specifications into two categories, the unary operator for a single atomic proposition, such as Eventually or Next, and the other for describing the relationship between two or more APs, such as Until, Before or After.

For example, Fig. 1(d) shows the ultimate IR tree after information extraction. We can see that “the gas concentration is below 8%” is correctly marked by class Clause(T-B-U), which indicates it is a temporal clause with binary operator: Until. The complete subsentence of the activation of the alarm is marked by class Clause (T-U-X), which indicates it is a temporal clause with unary operator: Next[2].

E. *LTL Formula Generation*

Finally, we generate an LTL specification by traversing the IR tree. The procedure contains two steps: atomic propositions generation and formula generation.

First, we generate an AP for each simple clause. The task is performed by looking for an exact match of $< subj, predicate, obj >$ with a set of pre-defined templates. For example, the template `subj be_GreaterThan obj` will be translated into $subj > obj$.

Then, we traverse the IR tree with a recursive algorithm to connect the APs with logic operators and generate a complete LTL specification. For each tree node, we expand the specification with corresponding *proposition* basing on the label. For a normal clause, we will generate an AP as described in the last paragraph. For a *condition* or an unary temporal operator, the algorithm recursively generates the subformula of the children nodes and compose it with logic and temporal operators. For a binary temporal operator, we write such binary clauses into LTL subformulas[3].

After visiting all the nodes, the specification is enhanced with operator G (globally), because it should be satisfied all the time. Finally, we get an LTL specification from the IR tree.

V. EVALUATION

The approach proposed in this paper has been implemented in a prototype tool in Java. The library used in our tool includes Stanford Parser [12], [13] for parse tree generation, Tregex [15] for pattern identification, and Tsurgeon [15] for parse tree modification.

A. *Experiment Setup*

In the absence of a sufficiently large corpus of natural language requirements in the field of smart homes, we design a user study to collect specifications from real users. We design a house scenario, which contains 29 sensors and devices that are popular in smart homes, including temperature sensor, air-conditioner, smart door and so on. Participants are asked to author 5-10 TAP rules and 5-10 natural language specifications they want the HA-IoT system to satisfy.

In total, 52 participants join this user study. 445 rules and 481 natural language specifications are collected. We group the rules and specifications authored by the same participant as a scenario. 2 participants only provide rules without specifications and these cases are removed. Finally, we get 50 distinct scenarios. Based on these data sets, we start to do the evaluation of our tool.

The evaluation is done on a workstation with Intel (R) Core (TM) i5-6300HQ CPU @ 2.30GHz, 4GB RAM, Ubuntu 14.04.5 LTS 64-bit.

B. *Effectiveness and Correctness of Translation*

We apply our tool on all the 481 natural language specifications to generate the corresponding LTL formulas. For each sentence, the processing time ranges from 17 milliseconds to 657 milliseconds, with an average of 55.6 milliseconds.

Then, we ask a group of 10 experts in the area of formal verification to go through the correctness of the generated LTL

[2]The binary temporal operator which is currently handled in this work including: Until, Before, After. The unary operator supported including Next, Future, and Global. This list can be expanded easily in the future work.

[3]There are many mature works about how to write such binary clauses into LTL subformulas. In this function, we implement the rules given in [14]. Due to the space limitation, please refer to [14] for detail.

specifications manually. The results are shown in Table I. Out of the 481 natural language specifications, 435 of them are translated correctly, the correctness rate is as high as 90.4%.

TABLE I
CORRECTNESS OF TRANSLATION.

#	Correct	Wrong	Total	Accuracy
Original Set	435	46	481	90.4%
Without Invalid Sentences	435	21	456	95.4%

We manually examine all the incorrect cases, and summarize the reasons that they fail into the following three classes:

- 16 specifications are wrong because these natural language specifications contain incomplete, wrong, or fuzzy information. For example, specifications like "The Bedroom should be comfortable" is difficult to formalize as "comfortable" is too subjective to translate.
- 9 specifications are wrong because the devices authored in the specifications do not exist in the system. In this case, our tool fails to find certain devices in the domain knowledge database and stop the translation.
- 21 specifications are wrong because the domain-related implicit semantic is not translated correctly. Take the specification "...the camera should take a photo and send the photo to my mobile phone..." as an example. Although these two actions are connected by "and", they don't happen at the same time in the specific context, "take" photo should occur in front of "send" photo.

Fairly speaking, the first two classes of errors cannot be treated as failure of our tool actually. If we treat these sentences as invalid, we can see from Table I, the accuracy of our method reaches 435/456=95.4%. Definitely, our approach can handle arbitrary cases. However, special cases, like the above photo case, can be addressed by introducing domain-related knowledge and rules into our framework in the future.

VI. Related Work

Translating natural language specifications to LTL has been investigated in many works. As there is no mature corpus for the specific problem, the widely used deep learning method in the natural NLP community can not be used here. The most relevant works with our goals are those of [6]–[8]. The first two works used structured natural language and restricted the grammar to a specific subset of English. This makes semantic parsing easier, but limits the expressiveness of the specification. Compared to them, we make no special restriction of the grammar of the specifications.

Study [8] translated stylized natural language into various formalisms from an intermediate representation. However, this approach relied entirely on the results of parsing and did not handle the wrong syntactic analysis caused by ambiguity. Our approach corrects and adapts the original parse tree, and resolves two kinds of ambiguities.

There are also works exploring artifact generation from natural language in the wide domain of software engineering [9], [16]–[19]. Compare to these works, we target a different goal in a different domain. To the best of our knowledge, we are the first to automatically translate the natural language specifications from HA-IoT users into LTL specifications.

VII. Conclusion and Future Work

In this paper, we present a novel grammar-based framework for automatically translating natural language specifications to LTL specifications in the HA-IoT domain, where we utilize the domain-related knowledge to help with ambiguity elimination and semantic extraction. We conduct a user study to collect 481 natural language specifications from real users. Our evaluation results show that our method has reached a high correctness rate of 95.4% in the formula generation.

Acknowledgment

The authors thank the anonymous reviewers for constructive reviews and Dr. Shujian Huang for his fruitful discussion. The authors in Nanjing University are supported in part by the National Key Research and Development Plan (No. 2017YFA0700604), and the National Natural Science Foundation of China (No.61632015, No.61572249, No.61561146394, No.61802166). Mingsong Chen is supported by the National Natural Science Foundation of China (No.61872147).

References

[1] L. Bu, *et al.*, "Systematically ensuring the confidence of real-time home automation iot systems," *TCPS*, vol. 2, no. 3, pp. 22:1–22:23, 2018.
[2] C. M. Liang, *et al.*, "Systematically debugging iot control system correctness for building automation," in *Proc. BuildSys@SenSys*, 2016, pp. 133–142.
[3] L. Zhang, *et al.*, "Autotap: synthesizing and repairing trigger-action programs using LTL properties," in *Proc. ICSE*, 2019, pp. 281–291.
[4] J. Croft, *et al.*, "Systematically exploring the behavior of control programs," in *Proc. USENIX ATC.*, 2015, pp. 165–176.
[5] K. Hsu, *et al.*, "Safechain: Securing trigger-action programming from attack chains," *IEEE TIFS*, vol. 14, no. 10, pp. 2607–2622, 2019.
[6] H. Kress-Gazit, *et al.*, "Translating structured english to robot controllers," *Advanced Robotics*, vol. 22, no. 12, pp. 1343–1359, 2008.
[7] R. Yan, *et al.*, "Formal consistency checking over specifications in natural languages," in *Proc. DATE*, 2015, pp. 1677–1682.
[8] S. Ghosh, *et al.*, "ARSENAL: automatic requirements specification extraction from natural language," in *NASA NFM*, 2016, pp. 41–46.
[9] A. Chhabra, *et al.*, "Formalizing and verifying natural language system requirements using petri nets and context based reasoning," in *Proc. MRC*, 2018, pp. 64–71.
[10] A. Pnueli, "The temporal logic of programs," in *IEEE FOCS, Providence*, 1977, pp. 46–57.
[11] Princeton University, "About wordnet." https://wordnet.princeton.edu/citing-wordnet, 2010.
[12] D. Klein and C. D. Manning, "Accurate unlexicalized parsing," in *Proc. ACL, 2003.*, 2003, pp. 423–430.
[13] M. de Marneffe, *et al.*, "Generating typed dependency parses from phrase structure parses," in *Proc. LREC*, 2006, pp. 449–454.
[14] M. B. Dwyer, *et al.*, "Patterns in property specifications for finite-state verification," in *Proc. ICSE*, 1999, pp. 411–420.
[15] R. Levy and G. Andrew, "Tregex and tsurgeon: tools for querying and manipulating tree data structures," in *Proc. LREC*, 2006, pp. 2231–2234.
[16] I. S. Bajwa, *et al.*, "NL2 alloy: A tool to generate alloy from NL constraints," *JDIM*, vol. 10, no. 6, pp. 365–372, 2012.
[17] D. K. Deeptimahanti and M. A. Babar, "An automated tool for generating UML models from natural language requirements," in *Proc. ASE*, 2009, pp. 680–682.
[18] C. Wang, *et al.*, "Automatic generation of system test cases from use case specifications," in *Proc. ISSTA*, 2015, pp. 385–396.
[19] J. Zhai, *et al.*, "Automatic model generation from documentation for java API functions," in *Proc. ICSE*, 2016, pp. 380–391.