Towards Automated and Accurate Understanding of ARINC Standard in Heterogeneous Data Formats

Cuifeng Gao[1,2], Wenzhang Yang[3,*], Xianchang Luo[1,2], and Yinxing Xue[1,2,3]
[1]School of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui, China
[2]Suzhou Institute for Advanced Research, University of Science and Technology of China, Suzhou, Jiangsu, China
[3]Institute of AI for Industries, Nanjing, Jiangsu, China
gcf20225162@mail.ustc.edu.cn, yywwzz@mail.ustc.edu.cn, bigluo@mail.ustc.edu.cn, yxxue@ustc.edu.cn
*corresponding author

2025 25th International Conference on Software Quality, Reliability and Security (QRS) | 978-1-6654-7771-0/25/$31.00 ©2025 IEEE | DOI: 10.1109/QRS65678.2025.00037

Abstract—Accuracy and rigor are vital indicators of the specification document, especially for the ARINC653 aviation industry standard. A high-quality standard or specification should clearly depict the system behaviors yet leave no fatal vulnerability. Formal verification could definitely help achieve this goal, but it requires intensive professional domain knowledge and overwhelming manpower. Recently, fast-growing natural language processing (NLP) techniques do well in harvesting knowledge extraction for the downstream tasks. However, since knowledge about an entity is scattered over heterogeneous contents (plain text, pseudocode, XML, etc.) for almost all such standard documents, a single content or not all contents cannot account for the entire knowledge. To this end, we propose a novel and practical approach to construct the Ontology of ARINC653 and extract the logical guards. Technically, we combine the NLP techniques with domain-specific naming and lexical rules for entity recognition in Ontology and then apply information extraction and relation formalization for relation extraction (in terms of guards). We evaluate the quality of our Ontology against that induced by the domain professor. We further apply this approach to the historical ARINC653 standards and evaluate the performance. Results show that our approach indeed helps construct knowledge integration and aid for specification understanding.

Keywords–ARINC653; NLP; heterogeneous knowledge integration; inconsistency detection

1. Introduction

A real-time operating system (RTOS) is an operating system (OS) that can process data as it comes in and make rapid responses to external events within a specified time. The standard or specification of an RTOS should be satisfied, especially for the industrial aviation ARINC653 O/S [1] which has a much stricter need for timeliness. Drafting ARINC653-like standards/specifications are highly expert-required, but still error-prone. During the evolution of the ARINC653 standards, hundreds of action items were proposed and taken into the standards; nevertheless, errors appeared in each release of the standards. For example, Zhao *et al.* found some critical errors in the ARINC653 standard and confirmed their existence in some ARINC653 OSes in the work [2]. Hence, to make sure the ARINC653 OS to be highly trusted, the corresponding standard is desired to be logically explicit and flawless.

Documented standards have been formalized and verified to ensure their correctness and improve the assurance of critical systems based on them. The ARINC653 standard has been fully formalized and verified by Event-B [2]. The Vellvm project provides a mechanized formal semantics of LLVM IR language, which is a standard for compilers' intermediate representation, using the Coq interactive theorem prover [3]. Numerous approaches have been proposed to formally specify and verify service compositions [4], such as WS-BPEL [5], a standard for business process execution language for web services. Existing works manually translate the standards into formal models and conduct verification by theorem proving or model checking. After all, these methods are promising, but they face two major challenges in practice. *First, verification-based methods are in general costly and time-consuming*. Though automated formal verification has been studied in recent years [6], automated creation of formal models (or specifications) from informal and semi-structured standards are still challenging. *Second, most existing methods focus only on code or text description for analysis and verification*. Thus, they fail to consider the content scatted in heterogeneous data formats in ARINC653 standard (see § 2-B).

In this study, the motivation is to bridge the above gap by leveraging the natural language processing (NLP) techniques for knowledge understanding. Important knowledge in ARINC653 standard is scattered in heterogeneous contents (e.g., plain text, pseudocode, and XML) so that it is heavily manpower-required to integrate them. On the other hand, knowledge and further Ontology of standards can help improve the consistency of the standard document and the linking between semi-formal and formal content of the standard document for safety-critical avionics systems [7]. To resolve this concern, we propose a novel approach based on NLP tools to reach a comparable performance with minimum required domain expertise.

Technically, given an extra IT glossary and some potential entities provided by human experts (see § 3), our approach can parse the ARINC653 document and conduct three key steps of processing, including named entity recognition (NER), relation extraction (RE) and relation formalization. Finally, it outputs a refined Ontology model (showing entities, entity attributes and entity relations) and the formalized entity relations in logical guards (first order logic). Besides, it yields an analysis report

2693-9177/25/$31.00 ©2025 IEEE
DOI 10.1109/QRS65678.2025.00037

about the quality of ARINC653 specification, such as typos, confusing alias, inconsistent type definition, etc.

To our best knowledge, we make the first attempt to apply the NLP tools to extract knowledge for ease of understanding and formalizing the documented standards like ARINC653. We conduct an evaluation on three versions of ARINC653 (see § 5). Results confirm the quality of our Ontology model, prove the accuracy of the analysis approach, and show the usefulness of the formalized relations in logical guards.

To sum up, we make these contributions in this study:

- Implementing *automated* knowledge extraction from heterogeneous data formats in the ARINC653 standard.
- Combining NLP techniques for *accurate* entity recognition and relation extraction from heterogeneous data formats.
- Yielding *useful* Ontology model and logical guard for understanding the ARINC653 standard.
- Pointing out 8 typos and 21 inconsistent issues legacy in ARINC653 standard. We form them as an errata report to ARINC Industry Activities and get their acceptance.

2. Background

2.1 ARINC653 Standard

ARINC653 [8] aims to provide a standardized interface between a given Partitioned operating systems (POS) and application software, as well as a set of functionalities to improve the safety and certification process of safety-critical systems. The latest version of ARINC653 published in 2019[1] is organized in six parts. Part 1 specifies the required services; hence, ARINC653 compliant POSs are mandated to implement this part. Other parts are overview, extended services, conformity test, subset services, and required capabilities. The last three versions of ARINC653 Part 1 are P1-3 (Part 1 Version 3) published in 2010, P1-4 in 2015 and P1-5 in 2019. Part 1 defines the system functionality of POSs by natural language descriptions and service requirements as APplication EXecutive (APEX) interface by the APEX service specification grammar which is in semi-formal as a combination of natural and structural languages. ARINC653 P1-5 defines six types of system functionalities, i.e. Partition Management, Process Management, Time Management, Memory Management, Inter-partition Communication, Intra-partition Communication and Health Monitoring. In particular, the scheduling specified in ARINC653 is a real-time and two-level scheduling, including partition scheduling and process scheduling. The deadline time (a delay time or a specified budget time) is usually utilized in scheduling model. Notably, in this study, we mainly focus on part 1 (the content of 300 pages in version 5) of the ARINC653 standard.

2.2 ARINC653 Part1 Representation

To provide more details for better guidance, the specification organizes its contents in heterogeneous formats such as plain text, pseudocode, XML-schema, etc. Without any exception, the Partition Management part has relevant content in three

[1]https://www.aviation-ia.com/file/3925/download?token=B0Z8FoMg

VARIABLE ATTRIBUTES (defined and controlled during run-time)
1. Lock Level – denotes the current lock level of the partition.
2. Operating Mode – denotes the partition's execution state.
3. Start Condition – denotes the reason the partition is started.

(a) VARIABLE Attributes (from p.15)

IDLE: In this mode, no application defined partitions are executing within the partition's allocated partition windows. The partition is not initialized (e.g., none of the ports associated to the partition are initialized), no processes are executing, but the partition windows allocated to the partition are preserved.

(b) One state of Operating Mode: IDLE (from p.18)

Figure 1: Plain text: describing attributes of Partition Management (class attributes in OOP implementation)

```
type PARTITION_STATUS_TYPE is record
    IDENTIFIER          :  PARTITION_ID_TYPE;
    PERIOD              :  SYSTEM_TIME_TYPE;
    DURATION            :  SYSTEM_TIME_TYPE;
    LOCK_LEVEL          :  LOCK_LEVEL_TYPE;
    OPERATING_MODE      :  OPERATING_MODE_TYPE;
    START_CONDITION     :  START_CONDITION_TYPE;
end record;


for PARTITION_STATUS_TYPE use record at mod 8;
    PERIOD              at  0   range 0..63;
    DURATION            at  8   range 0..63;
    IDENTIFIER          at  16  range 0..31;
    LOCK_LEVEL          at  20  range 0..31;
    OPERATING MODE      at  24  range 0..31;
    START_CONDITION     at  28  range 0..31;
end record;
```

(a) PARTITION_STATUS_TYPE (from p.137)

```
procedure SET_PARTITION_MODE
 (OPERATING_MODE    : in  OPERATING_MODE_TYPE;
  RETURN_CODE       : out RETURN_CODE_TYPE) is
error
 when (OPERATING_MODE does not represent an existing mode) =>
    RETURN_CODE := IVALID_PARAM;
 /*  omit several lines  */
normal
 set current partition's operating mode := OPERATING_MODE
 if (OPERATING_MODE is IDLE) then
    shut down the partition;
 if (OPERATING_MODE is WARM_START or OPERATING_MODE is COLDE_START) then
    inhibit process scheduling and switch back to initialization mode;
 if (OPERATING_MODE is NORMAL) then
    inhibit the partition;
    activate the process scheduling;
 RETURN_CODE := NO_ERROR;
end SET_PARTITION_MODE;
```

(b) SET_PARTITION_MODE (from p.53)

Figure 2: Pseudocode: describing the behaviors of Partition Management (class methods in OOP implementation)

heterogeneous formats — plain text in Figure 1 to describe the attributes (class attributes in OOP [9] implementation), pseudocode in Figure 2 to describe its behaviors (methods and code in OOP implementation), and XML-Schema files in Figure 3 to describe the structure of each part and relations among parts (analogical to class structure and their relations in OOP implementation). *As descriptions for attributes and behaviors of each part are distributed in these three heterogeneous formats, we should integrate such information together to acquire the entire knowledge.*

2.3 Challenges

In practice, we encounter the following challenges:

1) *Direct application of NLP tools is not enough to extract useful information from the heterogeneous formats.* As illustrated, Figure 1 shows an example of using plain text for attribute definitions of Partition; in Figure 2 pseudocode contains variable (or structure) declaration and the operations of pseudocode for Partition.Operating_Mode; and Figure 3 is the structural definition in XML-Schema for entity Partition

```
<xs:element name="Partition" maxOccurs="unbounded">
  <xs:attribute name="Identifier" type="IdentifierValueType"
      use="required">
    <xs:annotation>Identifier of the partition</xs:annotation>
  </xs:attribute>
  <xs:attribute name="Name" type="NameType" use="required">
    <xs:annotation>Name of the partition</xs:annotation>
  </xs:attribute>
  /*omit similar definitions for Duration and Period*/
  <xs:element name="MemoryRegions">
    <xs:annotation>Memory Region Mapped into the
        partition</xs:annotation>
    <xs:complexType>
      <xs:element name="MemoryRegion"
          type="A653_MemoryRegionType" maxOccurs="unbounded"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="PartitionPorts">
    <xs:annotation>Ports of the partition</xs:annotation>
    <xs:complexType>
      <xs:element name="PartitionPort" type="PortType"
          maxOccurs="unbounded"/>
    </xs:complexType>
  </xs:element>
</xs:element>
```

Figure 3: XML-Schema (from p.207)

and its related entities. However, the traceability of the same entity (e.g., Partition) among heterogeneous formats is not explicitly marked. Different name conventions (or alias) could be applied in heterogeneous formats, which aggravates this issue. How to find information of the same entity across different formats is a challenging task.

2) *There are many document quality issues, such as inconsistent usage of special symbols (e.g., ':', '–', '-', etc.), typos, improper alias, ambiguous references or pronouns.* We find two types of alias among these entity names: acronym/abbreviation and alias due to others (case sensitive or hyphen, etc.). Besides, inconsistent phrasing of the same entity will confuse the audience and hinder the readability. For example, the term Error Process, which literally means a Process with error, is an inappropriate presentation — because its real meaning is the handler for Error Process (thus a better term should be Error Handler Process). Obviously, directly applying named entity recognition (NER) tools without proper pre-processing and normalization may not yield ideal results.
3) *Due to the domain specificity of the ARINC653 standard, even with entities identified, the entity relations could be hard to extract by existing relation extraction (RE) tools.* Terms in the ARINC653 standard often differ from the meanings they have in daily-life documents such as news or novel. For example, the entity Blackboard here indicates an area that all Process within the same Partition can read Message from or write Message to, rather than the large dark board in daily life attached to a wall and used for writing. The existing RE tools mainly work on the daily-life documents (such as biomedicine, news reports or encyclopedia). As there lacks a large dataset of documented standards (related to ARINC653) with entity-relation knowledge well-labeled, it is extremely difficult to train a domain-specific RE model on supervision learning. Hence, directly or indirectly applying RE tools may not help. Finding alternative tools, which work at a more general level, poses a technical challenge on accurate relation extraction in domain-specific documents.

To sum up, towards an accurate and automated analysis of the ARINC653 standard, we propose our novel NLP-based approach to address these challenges in § 3 and § 4.

3. Approach Overview

In this section, we depict the basic steps of our approach towards acquiring knowledge in heterogeneous formats.

3.1 Overall Workflow

Figure 4 illustrates the workflow of our approach. It needs three types of input: the ARINC653 standard document, an external dictionary for IT glossary [10] (e.g., containing acronym and abbreviation) and a list of potential entities provided by human experts. The output of our approach includes two parts: the refined Ontology model (with entities and their relations extracted), the analysis report showing the grammar issues (e.g., typos) and logic flaws (e.g., inconsistent type definition or constraint) in the ARINC653 standard. Overall, our approach consists of the following processing: ① heterogeneous content processing (step 1-2, see § 4-A), ② entity recognition based on alias clustering (step 3, see § 4-B), ③ relation extraction based on information extraction (step 4, see § 4-C), and ④ relation formalization (step 5, see § 4-D). We briefly introduce the five steps in Figure 4 as below.

- **Step 1**: parsing document, in which we follow common documentation practices (e.g., headings, bullet lists, indention, etc.) to identify the corresponding layout information. This step could be used for many ARINC653-like standards that conceptually follow a similar pattern.
- **Step 2**: integrating content, in which we put together the contents parsed from heterogeneous formats (e.g., attribute definition in plain text; variable declaration in pseudocode and XML-Schema types definition). Besides, we design and implement data structures (similar to *Class* in OOP[9]) to store the content no matter from which format it is extracted.
- **Step 3**: entity recognition, in which we use the IT glossary, acronyms in the appendix of the ARINC653, and the potential entity list to cluster alias in the ARINC653 via the ClusterAlias algorithm (see § 4-B). This algorithm handles similar entity/attribute names due to case insensitivity, acronym, abbreviation and abuse of hyphens, typos, and highly similar (calculated by the Glove word embedding [11] method) noun-chunk.
- **Step 4**: relation extraction, in which we parse the input pseudocode and the entity description, and output the subject–verb–object (SVO) list via the aid of the information extraction (IE) tool OpenIE4 [12]. Due to repetitions and redundancy of the information extraction, the SVO list generated by OpenIE4 also has duplication. So we propose the IEMerger algorithm (see § 4-C) to identify the similar or redundant SVOs, simplify and merge them.
- **Step 5**: relation formalization, in which we convert the relations in form of SVO into formal formula (i.e., guard clauses in first order logic). Converting SVO into logic guards is not a trivial task, for which we propose the FormalRelation algorithm (see § 4-D) that considers different SVO structures and outputs corresponding guards.

With the outputs of step 3 and step 5, we could get a list of entity names as well as a list of merged and refined

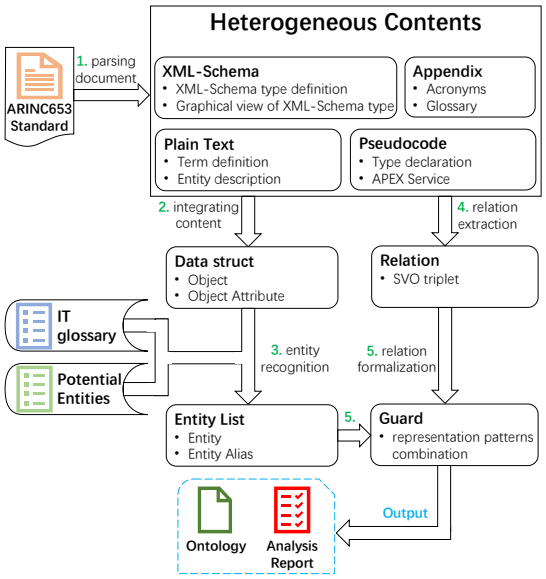

Figure 4: Overview of Our Approach

relations that machines can understand among the entities that machines also can understand. Based on that, we build the Ontology model in OWL [13] format via the aid of PROTÉGÉ [14]. During the parsing and processing of the standard, we also identify typos, ambiguous references, misuse of special symbols, unclear coreference in text, and inconsistent relations across various formats, and finally yield the analysis report.

3.2 Solutions to Challenges

Notably, to address the challenge 1 due to heterogeneous formats, during step 1, we utilize Python libraries PDFMINER and DOCX to handle the file format issue of the standard (in *.pdf* [15] and in *.docx* [16] respectively). For example, we handle terms in heading, terms in bold font, terms in underscores, possible relation in bullet lists, etc. Last, the terms in Figure 2 and type definitions in XML-schema in Figure 3 follow certain patterns, which we can accurately parse.

To address the challenge 2 due to document quality issues, during steps 2–3, we propose several methods to carefully validate the content in different formats. To handle the typo issues, we adopt the Levenshtein editing distance [17] and word embedding [11] similarity to identify and correct the misspelled terms or typos. To handle the entity name differences due to acronym, abbreviation and misuse of connection symbol, we resort to the external dictionary of IT glossary and the NLP tools (i.e., STANFORD CORENLP [18] and SPACY [19]), and etc.

Last, to address the challenge 3 due to the domain specificity of the document, we design the steps 4–5 for accurate relation extraction. As aforementioned, instead of directly using existing RE tools (e.g., OPENNRE [20]), we apply the NLP tools (i.e., STANFORD CORENLP and SPACY) and the IE tool (i.e., OPENIE4 [12]). The solution idea is not to apply the general RE tools directly, but to first locate the entities in the content and then identify the relation via the aid of IE tool.

4. METHODOLOGY

In this section, we elaborate on each step of our approach.

4.1 Integrating Content from Heterogeneous Formats

To make the analysis as complete (no missing content) as possible, as mentioned in § 2-B, we consider content in three formats (see Figure 1 – Figure 3). Furthermore, our analysis includes some important content in the appendix of the standard (e.g., acronym list). As shown in Table 1, useful knowledge from various formats can be extracted and categorized into three types: entity, entity attributes, and entity relation.

Knowledge Collection. Information about entities can be identified according to the content layout, term naming and lexical rules inside different formats. Thus, we summarize the following three guidelines for identifying them:

- *Plain Text.* In this part, we find the most useful knowledge is about the description of entity names and relations between entities. For instance, the entity name Partition is explained as follows: "*A partition is therefore a program unit of the application designed to satisfy these partitioning constraints*". Further, the entity relation between Partition and Process is identified from the following text: "*a partition comprises one or more processes that combine dynamically to provide ...*".
- *Pseudocode.* In pseudocode, the most useful knowledge is about entity attributes and entity relations. The pseudocode follows an Ada-like or a C-like syntax for variable declaration or type definition — in fact, pseudocode defines the basic data structure for the implementation class of the RTOS. For example, the three conditional statements in 2(b) are about attribute Partition.Operating_Mode and the corresponding enumerated value is one member of the set {IDLE, WARM_SATRT, COLD_START, NORMAL}. Meanwhile, plenty of entity relations are defined in pseudocode. For the pseudocode code "*if (there are any processes waiting for that event) then ...*", in § 4-C, we can identify the entity relation in SVO: ['any processes', 'waiting', 'for that event']. Naturally, the pseudocode is used to exhibit the data structure (attributes or variables) and actions (methods or state transitions), but lacks detailed natural language descriptions.
- *XML-Schema Files.* As shown in Table 1, XML-Schema files contain few entity descriptions (annotation content), but a few entity relations and plenty of entity attributes. Take the XML-Schema file in Figure 3 as an example, the entity Partition has not only the basic attributes Partition.Identifier and Partition.Duration, and also the *has* relation for MemoryRegions and PartitionsPorts. Then in § 4-C, we extract such relations in SVO: ['partition', 'has', 'memory regions'] and ['partition', 'has', 'partition ports'].

Knowledge Integration. Based on Table 1, the desired knowledge is scattered over three heterogeneous formats of content and only from one or two parts of them cannot acquire the entire knowledge. Note that the content in the appendix (see

Table 1: Knowledge Distribution

Contents \ Knowledge	Entity	Entity Attributes	Entity Relation
Plain Text	◕	◒	◕
Pseudocode	◔	◒	◕
XML-Schema	◔	◕	◒

[1] Note that ◕ means "plenty of"; ◒ means "a few"; ◔ means "few"; and the descriptions of Entity Relation in Plain Text are mostly contained in Pseudocode.

Figure 4) does not directly provide useful knowledge on an entity, but it supplies a list of acronyms (not exhaustive). Besides, we encounter some cases where a single term (e.g., an entity or an attribute) has different names (called alias), for example, Memory Requirements in plain text and MemoryRegions in XML-Schema all indicate the memory bounds of the Partition. Thus, at this step, the task is to collect the knowledge relevant to the same entity (e.g., MemoryRegions) together from various formats. The more accurate analysis such as clustering alias, merging extracted relations and formalizing refined relations will be done in the following steps. So till now, we have collected useful knowledge from the content in different formats, and integrate them together in three categories: entity name (along with entity descriptions), entity attribute (along with definition and declaration) and entity relation (sentences describing entity relations, not in SVO form yet).

4.2 Handling Document Quality Issues

As mentioned in § 2-C, the standard document suffers from some quality issues: typos, misuse of special symbols, alias due to case sensitivity, acronym, abbreviation or hyphenation.

Typos. Though ARINC653 standard has gone through a history of 15 years, with a series of five versions, there still exist some typos. For example, "*DecOrHexValueType*" is misspelled as "*DexOrHexValueType*" (see Table 3) in XML-Schema type definition part. To address this issue, we propose two methods: the Levenshtein distance and the word embedding technique.

Special Symbols. The existence of some special symbols (e.g., ':', '–', '(', ')' and '-'), which disturbs the syntactic structure, impairs the performance of sentence splitter tools. For example, due to the parentheses, SPACY mistakenly splits the sentence "*The creation of processes (e.g., names used, number of processes, etc.) for one partition ...*" into "*The creation of processes (e.g., names used, number of processes, etc.)*" and "*for one partition ...*". Such bad phenomena will definitely cause severe information loss for downstream works. Considering such issue, we adopt two methods to resolve parentheses: (1) when the content in parentheses is used as a complement, we retain the content whose starting word PoS (Part of Speech) is one of the following types: {preposition or subordinating conjunction, coordinating conjunction, wh-adverb}. Otherwise, the parentheses part will be deleted. (2) When the content before and in current parentheses can constitute a full name/acronym pair (see Figure 5), we will replace the full name with its acronym.

Acronym and Abbreviation. The usage of acronyms and abbreviations for entity names or attributes is observed across

> The primary objective of this Specification is to define a general-purpose APEX (APplication/EXecutive) interface between the Operating System (O/S) of an avionics computer resource and application software.

Figure 5: Example of full name/acronym pair (from p.1)

Algorithm 1: ClusterAlias

Input: $initialDict$: a dictionary for IT glossary
Input: $entityList$: a list of entity/attribute names
Input: $sents$: a list of sentences describing entity/attribute
Output: $aliasDict$: the dictionary for entity/attribute names and their alias

```
1  for (full name/acronym) pair P ∈ sents do
2      initialDict ← initialDict ∪ P;
3  aliasDict ← entityList ∪ initialDict.values;
4  repeat
5      foreach s ∈ sents, t ∈ aliasDict.items do
6          // Regular Expression Matching
7          if re.matched(s, t) then
8              aliasDict ← aliasDict ∪ matched alia;
9          // Levenshtein Distance
10         else if find_near_matches(t, s, K(t)) then
11             aliasDict ← aliasDict ∪ matched alia;
12         // Bag of Words Embedding Cosine Similarity
13         else if CosSimilarity(candidate alia ∈ s, t) ≥ θ1 then
14             aliasDict ← aliasDict ∪ candidate alia;
15 until no new alias clustered;
16 return aliasDict
```

the three heterogeneous formats (see § 4-A). In total, an entity or attribute name in ARINC653 standard usually has more than one alias, owing to the existence of case insensitivity, full name/acronym, and abuse of hyphens, etc. Take the acronym "FIFO" as an example, members of its alias set are {FIFO, fifo, First In/First Out, first-in/first-out, First In First Out, first-in-first out}. To avoid ambiguity and confusion, we need to cluster the alias of the same entity or attribute.

Algo. 1 shows how we deal with the issues in entity recognition. The input of Algo. 1 includes the term dictionary $initialDict$ initialized with Appendix B and an IT glossary, a list of entities and their attribute names $entityList$ (output of § 4-A Knowledge Collection part), and a list of splitted sentences $sents$ from the content of all three formats. The output of Algo. 1 is a list of normalized alias dictionary $aliasDict$ (the key is a normalized entity name/attribute, the value is the set of its corresponding alias, including its acronym, abbreviation, names with special symbols and typos). The idea of Algo. 1 is to cluster alias like an iterative snowballing method [21] until no new alias is recognized.

Specifically, lines 1-2 depict the processing of full name/acronym pair in parentheses for the example in Figure 5. Lines 4-14 iteratively check whether the words in a given sentence s from $sents$ satisfy one of the following conditions: (1) the candidate alia is matched with an element t in $aliasDict$ using regular expression (this matching pattern is case insensitive); (2) return *True* from fuzzy search function $find\_near\_matches$[2] which is based on Levenshtein distance [17]. The threshold function $K(t)$ is defined in Equation 1

[2]fuzzysearch is a Python package useful for finding approximate subsequence matches, https://github.com/taleinat/fuzzysearch

where the scalar α is assigned 0.3 after some convergence judgment experiments. In addition, we can identify some typos with the aid of Levenshtein distance. (3) we propose the cosine similarity between the word embedding candidate alia in each sentence s and each element t in $aliasDict$. When the cosine similarity (the word embedding is from [11]) between a candidate alia parsed by SPACY and the element t in $aliasDict$ greater or equal to θ_1 (empirically assigned 0.96 in this study), the candidate alia is qualified (such as Process_State and "*the state of the process*"). Algo. 1 will terminate until no new alias is found after scanning the entire list of sentences $sents$ as a whole. Notably, given a known entity/attribute name t, the threshold $K(t)$ for Levenshtein distance is proposed in the Chen et al's study [22], and defined as below:

$$K(t) = min(5,\ len(t) * \alpha) \tag{1}$$

Consequently, after the alias dictionary is established, the analysis of entity/attribute names could be more accurate. After on rule-based named entity recognition on the $aliasDict$ using spaCy, we aim to solve the issues arising from relation extraction.

4.3 Handling NLP analysis issues

As mentioned in § 2-C, due to the domain specificity of the ARINC653 standard, it is hard to directly apply the existing RE tools for effective relation extraction. Besides, towards accurate analysis, we need to handle coreference resolution.

Coreference Resolution. As an RTOS requirement and design document, the ARINC653 standard still has many parts of natural language descriptions. One challenging issue for NLP tasks is coreference resolution, e.g., the pronouns "*it*", "*which*", "*this service*". Such pronouns would not only hinder the readability for the human audience but also pose challenges for automated extraction of relation. For the simple pronouns like "*it*" and "*which*", we can adopt the coreference resolution technique (implemented by SPACY or STANFORD CORENLP). But for the coreference (e.g., "*this material*", "*this service*") that its corresponding mention does not appear in the surrounding context, the existing coreference resolution tools fail. We address such coreference in a declaration-based method [23]. For example, "*this service is called by the error handler process*" is resolved into "*RAISE_APPLICATION_ERROR service is called by the error handler process*".

Relation Extraction. As explained in Table 1, the information of entity relation is scattered over plain text, pseudocode and XML-schema. We follow different strategies to extract relation knowledge from these formats.

- In plain text, it is common practice to use the RE tools to get the relation of entity-pair, such as the tool OPENNRE [20] that should provide a plain sentence and the relevant entity-pair indexes. Besides, other tools [24], [25], [26], [27] can output relation triplets with only raw text input. As in § 4-B, we have identified the list of entity names and their attributes. Furthermore, we do proper coreference resolution and extract all possible relations between these entities. For example, from "*Each partition consists of one or more concurrently executing processes*" we can get the relation ['each partition', 'consists of', 'one or more processes'].
- In XML-schema, for the example in Figure 3, the entity name, their attributes and their relations could be explicitly identified. At line 1, the tag *<xs:element name= "Partition" Occurs="unbounded" >* defines the entity name Partition; at line 2, the tag *<xs:attribute name= "Identifier" ...>* defines the entity attribute Partition.Identifier. At lines 15–20, the tag *<xs:element name= " PartitionPorts" >* is defined in the body of <xs:element name= "Partition"> (line 1) and *<xs:element name= " PartitionPort" >* (line 18) is defined in the body of *<xs:element name= " PartitionPorts" >* (line 15) via the tag *<xs:complexType>* — this means entity Partition *has* or *owns* entity PartitionPorts (aggregation relation), and PartitionPort is *part of* PartitionPorts (composition relation). Similarly, we can extract all the relations among the entities defined in the XML-schema.
- In pseudocode, for Figure 6, the capability of the existing RE tools is limited in extracting relation from pseudocode. The reason is twofold: (1) these tools can only generate such simple pre-defined relations (e.g., relation ['Person', 'is founder of', 'ORG'] of study [28] and relation ['Person', 'born in', 'LOC'] of study [29], etc) or are only available to extract partial relation; (2) existing RE tools pay much more attention to relation type extraction and ignore the modifier (e.g., adjective, prepositional phrase or subordinate clause, etc.) which directly impairs the information integrity. Hence, based on the IE tools that can extract more than one triplets to

Algorithm 2: IEMerger

Input: $sents$: a list of sentences describing entity relations (containing entities)
Output: $relations$: the list of SVO (subject-verb-object) triplet

```
1  relations ← [];
2  for s ∈ sents do
       /* information extractions */
3      IEs ← OpenIE4(s);
4      if complement exists in IEs then
5          concat relevant IEs;
6      if sub-relation exist in IEs then
7          delete sub-relations of IEs;
8      if to-infinitive exists in IEs then
9          concat infinitive of IEs;
10     if IEs are completely separated by connectives of s then
11         relation ← IEs with connectives;
12     else
13         if connectives exist in IEs then
14             split relevant IEs;
15             relation ← IEs with connectives;
16         else
17             relation ← IEs;
18     relations.append(relation);
19 return relations
```

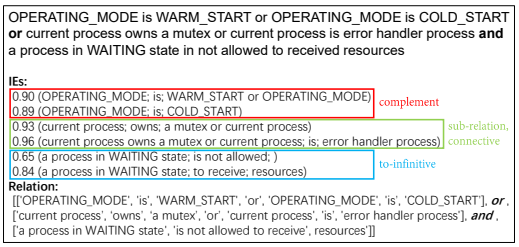

Figure 6: An example for the relation extraction in pseudocode

effectively avoid information loss, we propose Algo. 2 to extract relations from the sentences, especially in pseudocode.

The input of Algo. 2 includes the list of sentences $sents$ containing entity relations. The output of Algo. 2 is the list of SVO triplets $relations$. Specifically, lines 2–3 of Algo. 2 iterates each sentence inside $sents$ and applies the IE tool OPENIE4 respectively. Lines 4–5 are to concatenate the complements to assure the completeness of relation extraction, i.e., OPERATING_MODE could be COLD_START or WARM_START (see the red box in Figure 6). Lines 6–7 are to identify the sub-relation *(current process; owns; a mutex or current process)* is part of *(current process owns a mutex or current process; is; error handler process)* and then remove the sub-relation (see the green box in Figure 6). Lines 8–9 are to handle the relation to be extracted from the to-infinitive, for which the sentence "*a process in WAITING state is not allowed to received resources*" will be parsed into two relations *(a process in WAITING state; is not allowed)* and *(a process in WAITING state; to receive; resources)* (see the blue box in Figure 6). Lines 10–17 are to handle the cases that use the connectives *and*, *or* (like logical operators && and || in code). Notably, lines 13–15 handle the case *(current process owns a mutex or current process; is; error handler process)* where *or* is found in the IE (also the green box in Figure 6).

Finally, from the three heterogeneous formats, we extract the entity relations and collect them together into a SVO triplet list, named $relations$, for the following step of relation formalization.

4.4 Handling Relation Formalization

Till now, we can get the relations in form of SVO that are suitable for human reading and comprehending, but not for machine processing or verification still. Hence, we propose Algo. 3 to formalize the entity relations obtained in § 4-C.

As illustrated in Algo. 3 FormalRelation, its inputs include: $aliasDict$, the output of Algo. 1 that is an alias dictionary; $entityStructure$, the integrated entity/attribute knowledge obtained in § 4-A; $funcParameters$, functions and their corresponding input parameters in pseudocode; $relations$, the output of Algo. 2. The output of Algo. 3 is the logical guards in first order logic (FOL). Before the relation formalization, we adopt SPACY.ENTITYRULER to construct a rule-based NER using the alias dictionary $aliasDict$ (line 1). Then as shown in lines 3–10, we will subdivide each relation's components such as subject, verb, object and object complement (if exist) into

Algorithm 3: FormalRelation

Input: $aliasDict$: the output of entity recognition in §4-B
Input: $entityStructure$: the entity/attribute knowledge in §4-A
Input: $funcParameters$: functions and their input parameters in pesudocode
Input: $relations$: the output of Algo. 2
Output: $guards$: the list of FOL-like expressions

```
1  NER ← spaCy.EntityRuler(aliasDict);
2  guards ← [];
3  for rel ∈ relations do
4      tupleList ← [];
5      // subdivide rel parts using NER
6      for part ∈ rel do
7          if part is Subject or Object then
8              tupleList.append(parseNoun(part));
9          else if part is Verb or Verb Complement then
10             tupleList.append(parseVerb(part));
11     // transform tupleList into guard with the
           assistance of funcParameters and entityStructure
12     guard ← PredicatedTransform(tupleList);
13     guards.append(guard)
14 return guards
```

$tupleList$. Finally, at lines 11–12 we translate such $tupleList$ into a $guard$ through transformation rules for various representation patterns of the sentences (see examples in Table 6).

Take the example shown in Figure 7, before formalizing the relation ['the semaphore\'s current value', 'equals', 'its maximum value'] extracted from Algo. 2, we leverage the rule-based NER to subdivide all relation parts. For details, (1) subdividing the noun phrase (subject or object) into a quintuple [*Flag*, *DT*, *Adj*, *Noun*, *Complement*], where the *Flag* represents whether this noun phrase is affirmative, *DT* and *Adj* is article and adjective respectively, and *Complement* means a prepositional phrase (e.g., 'of Partition' and 'for Buffer', etc.) or a subordinate clause; (2) subdividing verb phrase (verb or verb complement) into a tuple [*Flag*, *Verb*]. Finally, based on our predicated transformation, we translate "*equals*" into "==", then formalize the attribute "*current value*" and "*maximum value*" as "SEMAPHORE(SEMAPHORE_ID).CURRENT_VALUE" and "SEMAPHORE(SEMAPHORE_ID).MAXIMUM_VALUE" respectively with the assistance of the input $funcParameters$ item SIGNAL_SEMAPHORE: (SEMAPHORE_ID, RETURN_CODE), where the key is a function name and value corresponds to its parameters (That is, given function name SIGNAL_SEMAPHORE, we can easily get the Semaphore key SEMAPHORE_ID).

Finally, we have gained the guards that machines can process and are applicable to auto-verification to check whether there are any statements with possible inconsistency in system entity/attribute knowledge.

5. EVALUATION

In this section, we conduct experiments to evaluate the quality of the Ontology outputted by our approach, the analysis accuracy and the practical usefulness of our approach.

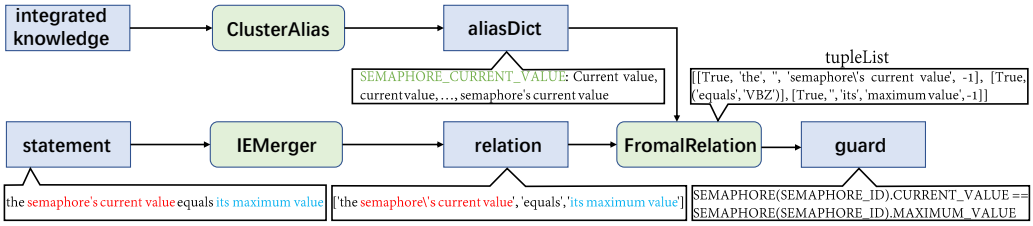

Figure 7: The input, output and examples of relation formalization

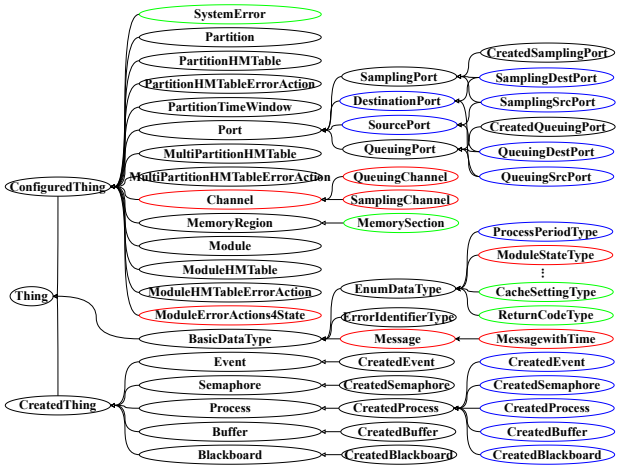

Figure 8: The overlaps and differences of the two models $Ontology_{NLP}$ and $Ontology_{HE}$. Ellipse nodes (representing entities) in black color denote the overlapping part; nodes in green color denote the unique necessary parts of $Ontology_{NLP}$; nodes in red color (and blue) denote the unique necessary (and unnecessary) parts of $Ontology_{HE}$, respectively.

5.1 Research Questions

We aim to answer these two research questions (RQs):

1) What is the *quality* of the Ontology model extracted with the aid of NLP techniques, compared with the one manually constructed by human experts?
2) What is the *accuracy* of our approach in analyzing among the three different versions (P1-3 to P1-5) of the ARINC653 standard? What are the analysis results?

To address RQ1, we use PROTÉGÉ [14] to generate our Ontology model (called $Ontology_{NLP}$) from the knowledge on entities and their relations, then compare it with the Ontology model summarized by human experts for version 3 in the study [2] (called $Ontology_{HE}$). The metrics to measure Ontology quality are the completeness of two models. To address RQ2, we measure the precision and recall of the two algorithms in relation extraction and formalization on three versions of part one of the ARINC653 standard (ARINC653 P1). Besides, we provide details of the analysis results on the anonymous website [30]: https://sites.google.com/view/nlp-for-arinc653/home.

5.2 Answer to RQ1: Quality of the Ontology Model

Ontology Quality. In Figure 8, we illustrate the overlaps and differences of the two models $Ontology_{NLP}$ and $Ontology_{HE}$, and Table 2 lists the statistics of the two models. In total, they share a similarity of 86.27% in terms of entities and relations (i.e., the class hierarchy and objective properties in Ontology). The unique part of $Ontology_{NLP}$ is about the 4 entities (associated with their attributes and relations, see Figure 8 green ellipse nodes) SystemError, MemorySection, CacheSettingType and ReturnCodeType. After contacting the authors of Zhao et al. [2], it is confirmed that these four entities are valid and should be considered into the Ontology model. On the other side, $Ontology_{HE}$ contains 7 unique and necessary entities (see Figure 8 red ellipse nodes) include Channel, ModuleErrorActions4State, ModuleStateType, Message as well as their sub-entities (if any). After we inspect the ARINC653 standard, we find these entities just appear with names, and not enough content in plain text or pseudocode is provided for NLP to extract any useful knowledge. The authors of Zhao et al. [2] include these 7 entities mainly based on their domain knowledge. Besides, the 15 (see Figure 8 blue ellipse nodes, and the rest 3 entities are not listed) remaining unnecessary entities of $Ontology_{HE}$ all are *enum* instances, which are actually considered in the SVO guards we extract — whether to consider them in Ontology is a modeling decision problem.Details of the two Ontology models could be found at [30].

Table 2: Statistics about $Ontology_{HE}$ and $Ontology_{NLP}$

Terms / Ontology Item	Entity (Classes Hierarchy)	Attributes (Data Properties)	Relations (Object Properties)
#Unique for $Ontology_{NLP}$	4	10	12
#Unique for $Ontology_{HE}$	7	8	28
#Intersection	**44**	**53**	**52**

Typo Fixing. During the construction of $Ontology_{NLP}$, we find that Algo. 1 almost has no errors in clustering alias — all the typos, acronyms, abbreviations for exciting entity/attribute names have been correctly identified as aliases. Hence, we apply our approach (see § 4-B) and find 9 typos in version 3 of ARINC653, listed in Table 3. In the latest version 5 of ARINC653, six typos are fixed and the remaining three typos still exist. Take a typical example, the entity name Error Process, which literally means a Process with error, is a typo. Its name should be Error Handler Process, as it actually refers to the handler for Error Process — this typo could be very misleading. This proves that our approach can indeed improve the standard quality at some easily ignored places in an automated way.

Answer to RQ1: Results show that $Ontology_{NLP}$ and $Ontology_{HE}$ have an overlapping of over 86.27%. $Ontology_{NLP}$ has knowledge that exists in the standard but is missed by $Ontology_{HE}$; $Ontology_{HE}$ has knowledge not detailed in the standard but added by human experts.

Table 3: Typos found in version 3 of ARINC653 standard

Typos	Term	Page No. of Version 3	Fixed in Version 5
process sate	process state	26	YES
error process	error handler process	53, 183	YES
DexOrHexValueType	DecOrHexValueType	179	NO
MultiPartitionHM-TableHameRef	MultiPartitionHM-TableNameRef	183	YES
NameTypes	NameType	186	NO
ARINX	ARINC	199	YES
PartitionType	PartitionsType	200	YES
memorySection	memorySectionType	211	YES
CacheSetting	cacheSetting	211	NO

Table 4: ACCURACY OF ALGORITHM 2 FOR THREE ARINC653 VERSIONS

ARINC653 P1	Version 3	Version 4	Version 5
#Total Relation	201	237	236
#Correct SVO	193	226	226
#Wrong SVO	3	6	5
#Missed SVO	5	5	5
Precision[1](%)	98.47	97.41	97.83
Recall[2](%)	97.47	97.83	97.83

[1] Precision=#Correct SVO/(#Correct SVO + #Wrong SVO)
[2] Recall=#Correct SVO/#Total Relation

5.3 Answer to RQ2: Analysis Accuracy and Results

Accuracy of Algo. 2. In Table 4, we show the accuracy of Algo. 2 in relation extraction for versions 3–5. Experiment results show that the precision is quite good, for three versions, no more than 6 relations are extracted into the wrong SVOs. These failed cases are attributed to the internal issues of the adopted tool OPENIE4, which cannot well handle relations in complex sentences and subordinate clauses. In each version, there are 5 missed cases that the corresponding SVOs are failed to get their relations. After inspection, we find that OPENIE4 will omit the object complement when encountering to-infinitive, for example, "*the specified port is not configured to operate as a source*" will be parsed into (*the specified port*; *is not configured*;) and (*the specified port*; *to operate*;) without its object complement "*as a source*" (source is an enumeration value). In total, except for the few false positives or false negatives due to OPENIE4, Algo. 2 exhibits an excellent accuracy for extracting relations from sentences.

Accuracy of Algo. 3. In Table 5, we show the accuracy of Algo. 3 in relation formalization from SVO to logical guards. For version 5, among the 226 correct SVO triplets from the previous step, Algo. 3 can correctly extract 216 guards but fail for the left 10 (226-216) SVO triplets. 6 out of 10 ones extract wrong guards due to complex sentence structures in SVO—multiple prepositional phrases, and causative sentence (e.g., "*no process has preemption locked*"). Meanwhile, for the other 4 SVO triples, no proper guards are generated due to the issues in subdividing parts by the NER tool. For example, when the SVO triplet is divided as ['PROCESS_ID', 'is', 'a process that owns a mutex or is waiting on a mutex\'s queue'], without domain knowledge, it is hard for Algo. 3 to extract the correct guard like human experts. Overall, Algo. 3 also shows an excellent accuracy in relation formalization, and only fails when the sentence structure in SVO is very complex or the part subdivision by the NER tool has some issues.

Table 5: ACCURACY OF ALGORITHM 3 FOR THREE ARINC653 VERSIONS

ARINC653 P1	Version 3	Version 4	Version 5
#Correct SVO	193	226	226
#Correct Guard	186	217	216
#Wrong Guard	4	5	6
#Missed Guard	3	4	4
Precision[1](%)	97.87	97.75	97.30
Recall[2](%)	98.41	98.19	98.18

[1] Precision=#Correct Guard/(#Correct Guard + #Wrong Guard)
[2] Recall=#Correct Guard/#Correct SVO

```
when (current process is periodic and new deadline will exceed next release point)
    RETURN_CODE := INVALID_MODE
```

Figure 9: Attribute deadline mistakenly used for deadline_time in pseudocode

Quality of Logical Guards. The logical guards formalized from the SVO triplets are formulas of first order logic (FOL). Hence, these guards are not necessary to be atomic propositions, but compound propositions (see Table 6 column "Example Guard"). During relation formalization, we indeed find some atomic representation patterns in pseudocode, which is listed in Table 6 column "Representation Pattern". Each logical guard is basically a combination of logic propositions behind one or more atomic representation patterns. The last column "#Frq" of Table 6 lists the number of guards (inside 216) which has the corresponding representation pattern as part. For example, the representation pattern (3) (judging on an attribute of some entity) is the most frequent pattern found in logical guards, contained in 155 out of 216 guards.

Inconsistencies Found in Real-time Requirements. After getting the logic guards, we solve them and check the results. Interestingly, the solving results show these guards cannot be satisfied as a whole (indicating possible logical issues). After inspection, we find the issues root from the inconsistent term usage in the real-time requirements across different formats. *The attribute deadline is mistakenly used, when the attribute deadline_time is actually needed.* We found there exist more than 21 instances of inconsistent term usage on deadline and deadline_time across different formats in ARINC653 standard (see example in Figure 9)—deadline is an *enum* attribute and deadline_time is the actual time attribute. As a key concept in real-time system standards, the mistaken usage of deadline could cause critical issues in system implementation. Hence, we have reported such issues to the ARINC653 standard committee and get their acceptance.

Answer to RQ2: Results show that Algo. 2 and Algo. 3 exhibit excellent precisions and recalls for versions 3-5. The errors in the analysis are mainly due to 1) very complex sentence structure 2) the internal issues of used IE and NER tools. Our automated analysis also helps find 9 typos and more than 21 instances of requirement inconsistency.

5.4 Discussion

Aid in Automated Verification. In this study, we aim to bridge the gap between NLP techniques and automated formal verification. The successful construction of Ontology

Table 6: Representation Patterns and Their Usage in Guards

No.Representation Pattern	Relation in SVO	Example Guard	#Frq[1]
(1) **ENT(ENT_NAME/ENT_ID)**: the entity identified by the ENT_NAME or ENT_ID	['PROCESS_ID', 'does not identify', 'an existing process']	PROCESS(PROCESS_ID) ∉ Set(PROCESS)	96
(2) **this→ENT**: current ENT	['current process', 'is', 'error handler process']	this → PROCESS == ERROR_HANDLER_PROCESS	140
(3) **ENT.ATTR**: the attribute ATTR of ENT	['partition's lock level', 'is', 'greater or equal to MAX_LOCK_LEVEL']	this → PARTITION.LOCK_LEVEL ≥ MAX_LOCK_LEVEL	155
(4) **ATTR**: an enumeration or a parameter	['DELAY_TIME', 'is', 'infinite']	isInfinite(DEADY_TIME)	70
(5) **isAdj(ENT)**: ENT is adj	['identified process', 'is not', 'a suspended process']	isSuspended(PROCESS(PROCESS_ID))	44
(6) **Adj(ENT)**: adj ENT	['previous process', 'is not stopped']	¬ isStopped(Previous(this → PROCESS))	1
(7) **isVerbPrep(ENT1, ENT2)**: ENT1 is verb prep ENT2	['any processes', 'waiting', 'for that event']	∃ process ∈ Set(PROCESS) | isWaitingFor(process, EVENT(EVENT_ID))	45
(8) **Set(ENT)**: ENT set of current partition	['no current partition process', 'named', 'PROCESS_NAME']	PROCESS(PROCESS_NAME) ∈ Set(this→PARTITION.PROCESS)	94

[1] #Frq means the number of guards containing the corresponding representation form.

models and guards extraction from the ARINC653 standard has demonstrated the feasibility — construction of the high-quality formal model, as pain points in standard (or protocol) verification, could be facilitated by modern NLP techniques. Besides, we have conducted simple verification by solving the logical guards formalized from the relations in SVO triplets. Towards the goal of fully automated verification, in the future, we plan to automatically convert the Ontology model and logical guards into the Event-B notation [31]. Based on the Event-B model, we could conduct the verification on entity attributes and relations.

Efficiency of Our Approach. According to the authors of Zhao et al.[2], it takes them around 3 months' manual effort to construct $Ontology_{HE}$, even they have done the manual analysis for similar specification documents before. In this study, given the list of potential entities and some IT glossary (external dictionary, see § 3-A), through the aid of our NLP-based approach, it only takes around 10 minutes to automatically construct the model $Ontology_{NLP}$ and formalize the complementing entity relations in guards for any version of the ARINC653 standard. Moreover, as Figure 8 shown, we can conclude 4 unique entities omitted by the human expert.

Generality of Our Approach. The studied standard ARINC653 in this paper is a typical and well-structured document for software. It combines plain text, semi-formal pseudocode, and formal data structure in XML-Schema. This kind of documents is strongly recommended for a high *evaluation assurance level* in safety and security certification, e.g. Common Criteria (CC), DO-178C and EN61508. For instance, in CC certification, developers are requested to provide functional specification (FSP) and TOE design specification (TDS) documents similar to ARINC653 standards. The approach in this paper is directly applicable or extensible for such documents.

Threats to Validity. The internal threats to validity are threefold. First, our NLP-based approach heavily relies on the modern NLP tools such as STANFORD CORENLP, SPACY, and OPENIE4, etc. The issues of these tools affect the results of our approach. Second, some key parameters, e.g., the similarity threshold $K(t)$ for Levenshtein distance in § 4-B, are empirically determined in related literature [22]. Third, the current methodology operates under the assumption of well-structured, clean standard documents, which could impair its performance when applied to disorganized industrial specifications. Additionally, the ARINC653-specific rule adaptations carry inherent overfitting potential. In future, more rigorous experiments could be done to address this issue. The external threats to validity arise from the input quality (the entity list and the IT glossary) for our approach. In this study, we use the common terms in OS domain and the IT glossary [10] as input, and our approach does not require that the entity list or the IT glossary must be of high quality and complete.

6. RELATED WORK

The following lines of work are related to our study.

6.1 Formalization Verification

The purpose of verification is aimed to find any violation or vulnerability. Based on this starting point, [32] directly analyzes the Proverif model extracted from its JavaScript code to verify the protocol core. Besides symbolical proof Proverif, [33] also applies pi-calculus and uses computational proof CryptoVerif in the hope of finding deeper vulnerabilities. Recently, the verification of ARINC653 specification is accomplished by Circus language [34], AADL (Architecture Analysis and Design Language) [35], [36], [37], and PROMELA of the SPIN model checker [38], however, in which only a small part of services are modeled. In [2], the system functionalities and all service requirements of ARINC653 have been formalized in Event-B, and some errors have been found in the standard. All of the above formal specifications are developed manually, whilst this paper focuses on the automated creation of the ARINC653 standard.

6.2 Relation Extraction

Methods based on bootstrapping and supervised learning are proposed to extract relations.

Bootstrapping is also called snowballing which can extract the same relation given entity-pair and initial few relation seeds. It iteratively calculates the similarities between instances and existing relation patterns to cluster the same relation instances. [21] shows book-author relation extraction among instances whose named entities are already tagged. [29] recently reveals a novel bootstrapping method called Neural SnowBall that can address new relation issues by transfer learning with only a few instances. However, our approach can further retain the important modifier (e.g., adjective, prepositional phrase or subordinate clause, etc.) to ensure information integrity.

Moreover, a supervised relation extraction usually can extract more high-quality relations. An RNN model is presented by [39] for relation classification based on the corresponding word sequence features. [40] even applies a convolutional DNN for relation classification with the input of lexical and sentence-level features. To reach a better performance, [41] adds a Feature-rich Compositional Embedding Model (FCM) for subsequent relation extraction. Currently, popular RE tools also possess a high degree of automation that even with simple input examples, the desired relationships can be extracted. REVERB [24] only needs raw text input and quickly outputs subject-verb-object SVO triplets and their corresponding confidence. As for OPENNRE [20], entity-pair indexes are required additionally. Our approach can efficiently extract relations from the domain specification ARINC653 which has no comparable scale with the corpus (e.g., biomedicine [42], news reports, etc.) applied in the above models or tools.

6.3 Documentation Content Extraction

A document always contains the target content that will be parsed or verified. [43] translates the UML Statechart Diagrams into PROMELA and then automatically verifies this translation with the help of the SPIN model checker. [44] further supplements the automated Linear Temporal Logic (LTL) properties derived from the sequence diagram for the verification of a protocol. Instead, [32] pays attention to an interoperable implementation of a protocol (TLS 1.0-1.3) and directly analyzes this protocol core. Different from the above methods, [2] leverages the product of manually summarized Ontology induced from ARINC653 Standard as the input to the subsequent work. Notably, our approach can make full use of much more detailed information from various formats.

7. CONCLUSION

In this paper, we propose an approach towards knowledge extraction from heterogeneous data formats in the ARINC653 standard. Our approach combines the NLP techniques with domain-specific naming (e.g., acronym and abbreviation), content layout rules (e.g., headings, bullet, indention, etc) and lexical rules (e.g., usage of special symbols) to extract knowledge from the semi-structured ARINC653 standard. Based on that, we conduct the processing of named entity recognition (NER), relation extraction (RE) and relation formalization via the aid of modern NER and IE tools. Finally, experiments confirm the quality of our Ontology model, prove the accuracy of the analysis approach, and show the usefulness of formalized relations in logical guards (we find 9 typos and 21 inconsistency issues). In the future, we plan to automatically convert our Ontology model and logical guards into the Event-B notation for automated formalization verification.

ACKNOWLEDGMENT

This work was supported in part by the National Natural Science Foundation of China under Grant 61972373, in part by the Anhui Provincial Department of Science and Technology under Grant 202103a05020009, the Basic Research Program of Jiangsu Province under Grant BK20201192.

REFERENCES

[1] *Avionics Application Software Standard Interface: ARINC Specification 653P1-3, Required Services*, https://www.aviation-ia.com/products/653p1-5-avionics-application-software-standard-interface-part-1-required-services, Aeronautical Radio, Inc. Std., 2010.

[2] Y. Zhao, D. Sanan, F. Zhang, and Y. Liu, "Formal specification and analysis of partitioning operating systems by integrating ontology and refinement," *IEEE Transactions on Industrial Informatics*, vol. 12, no. 4, pp. 1321–1331, 2016.

[3] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Formalizing the llvm intermediate representation for verified program transformations," in *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. SIGPLAN, 2012, pp. 427–440.

[4] G. M. M. Campos, N. S. Rosa, and L. F. Pires, "A survey of formalization approaches to service composition," in *2014 IEEE International Conference on Services Computing*. IEEE, 2014, pp. 179–186.

[5] E. Stachtiari and P. Katsaros, "Compositional Execution Semantics for Business Process Verification," *Journal of Systems and Software*, vol. 137, pp. 217 – 238, 2018.

[6] L. Nelson, H. Sigurbjarnarson, K. Zhang, D. Johnson, J. Bornholt, E. Torlak, and X. Wang, "Hyperkernel: Push-button verification of an os kernel," in *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP, 2017, pp. 252–269.

[7] A. D. Brucker and B. Wolff, "Using ontologies in formal developments targeting certification," in *International Conference on Integrated Formal Methods*. Springer, 2019, pp. 65–82.

[8] *ARINC Specification 653: Avionics Application Software Standard Interface, Part 0 - Overview of ARINC 653*, Aeronautical Radio, Inc., June 2013.

[9] Wikipedia contributors, "Object-oriented programming," "https://en.wikipedia.org/wiki/Object-oriented_programming", 2021.

[10] J. C. Rigdon, "Dictionary of computer and internet terms," "http://www.damanhour.edu.eg/pdf/738/dictionaries/Dictionary_of_Computer_and_Internet_Terms_Words.pdf", 2016.

[11] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. EMNLP, 2014, pp. 1532–1543.

[12] M. Schmitz, H. Pal, B. Mani, and M. Guerquin, "Openie 4," https://github.com/allenai/openie-standalone, 2017.

[13] OWL Working Group, "Web ontology language (owl)," "https://www.w3.org/2001/sw/wiki/OWL", 2012.

[14] M. A. Musen and P. Team, "The protégé project: a look back and a look forward," *Ai Matters*, vol. 1, no. 4, pp. 4–12, 2015.

[15] P. ISO, "32000-1. document management–portable document format–part 1: Pdf 1.7," *01/07*, 2008.
[16] I. Standardization, "Information technology-document description and processing languages-office open xml file formats," *ISO/IEC 29500*, 2008.
[17] G. Navarro, "A guided tour to approximate string matching," *ACM computing surveys (CSUR)*, vol. 33, no. 1, pp. 31–88, 2001.
[18] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, "The stanford corenlp natural language processing toolkit," in *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*. ACL, 2014, pp. 55–60.
[19] M. Honnibal and I. Montani, "Industrial-strength natural language processing in python," *spaCy*, 2020.
[20] X. Han, T. Gao, Y. Yao, D. Ye, Z. Liu, and M. Sun, "Opennre: An open and extensible toolkit for neural relation extraction," *arXiv*, 2019.
[21] S. Brin, *Extracting Patterns and Relations from the World Wide Web*. The World Wide Web and Databases, 1998.
[22] C. Chen, Z. Xing, and X. Wang, "Unsupervised software-specific morphological forms inference from informal discussions," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 450–461.
[23] H. Li, S. Li, J. Sun, Z. Xing, X. Peng, M. Liu, and X. Zhao, "Improving api caveats accessibility by mining api caveats knowledge graph," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 183–193.
[24] A. Fader, S. Soderland, and O. Etzioni, "Identifying relations for open information extraction," in *Proceedings of the 2011 conference on empirical methods in natural language processing*. EMNLP, 2011, pp. 1535–1545.
[25] K. Gashteovski, R. Gemulla, and L. d. Corro, "Minie: minimizing facts in open information extraction." Association for Computational Linguistics, 2017.
[26] M. Gardner, J. Grus, M. Neumann, O. Tafjord, P. Dasigi, N. Liu, M. Peters, M. Schmitz, and L. Zettlemoyer, "Allennlp: A deep semantic natural language processing platform," *arXiv preprint arXiv:1803.07640*, 2018.
[27] P. Remy, "Python wrapper for stanford openie," https://github.com/philipperemy/Stanford-OpenIE-Python, 2020.
[28] D. S. Batista, B. Martins, and M. J. Silva, "Semi-supervised bootstrapping of relationship extractors with distributional semantics," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. EMNLP, 2015, pp. 499–504.
[29] T. Gao, X. Han, R. Xie, Z. Liu, and M. Sun, "Neural snowball for few-shot relation learning," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 5, pp. 7772–7779, 2020.
[30] anonymity, "nlp_for_arinc653," "https://sites.google.com/view/nlp-for-arinc653/home", 2021.
[31] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin, "Rodin: an open toolset for modelling and reasoning in Event-B," *STTT*, vol. 12, no. 6, pp. 447–466, 2010.
[32] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified models and reference implementations for the tls 1.3 standard candidate," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 483–502.
[33] N. Kobeissi, K. Bhargavan, and B. Blanchet, "Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach," in *2017 IEEE European symposium on security and privacy (EuroS&P)*. IEEE, 2017, pp. 435–450.
[34] A. Oliveira Gomes, "Formal specification of the arinc 653 architecture using circus," Ph.D. dissertation, University of York, Heslington, York YO10 5DD, 2012.
[35] Y. Wang, D. Ma, Y. Zhao, L. Zou, and X. Zhao, "An aadl-based modeling method for arinc653-based avionics software," in *2011 IEEE 35th Annual Computer Software and Applications Conference*. IEEE, 2011, pp. 224–229.
[36] J. Delange, L. Pautet, and F. Kordon, "Modeling and validation of arinc653 architectures," in *ERTS2 2010, Embedded Real Time Software & Systems*. ERTS2, 2010, pp. 1–8.
[37] F. Singhoff and A. Plantec, "Aadl modeling and analysis of hierarchical schedulers," *ACM SIGAda Ada Letters*, vol. 27, no. 3, pp. 41–50, December 2007.
[38] P. de la Cámara, J. R. Castro, M. d. M. Gallardo, and P. Merino, "Verification support for arinc-653-based avionics software," *Software Testing, Verification and Reliability*, vol. 21, no. 4, pp. 267–298, January 2011.
[39] K. Hashimoto, M. Miwa, Y. Tsuruoka, and T. Chikayama, "Simple customization of recursive neural networks for semantic relation classification," in *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. EMNLP, 2013, pp. 1372–1376.
[40] D. Zeng, K. Liu, S. Lai, G. Zhou, and J. Zhao, "Relation classification via convolutional deep neural network," *the 25th International Conference on Computational Linguistics: Technical Papers*, pp. 2335–2344, 01 2014.
[41] M. R. Gormley, M. Yu, and M. Dredze, "Improved relation extraction with feature-rich compositional embedding models," *Computer ence*, 2015.
[42] J. Shang, L. Liu, X. Ren, X. Gu, T. Ren, and J. Han, "Learning named entity tagger using domain-specific dictionary," *arXiv preprint arXiv:1809.03599*, 2018.
[43] D. Latella, I. Majzik, and M. Massink, "Automatic verification of a behavioural subset of uml statechart diagrams using the spin model-checker," *Formal Asp. Comput.*, vol. 11, pp. 637–664, 12 1999.
[44] P. S. Kaliappan, H. Koenig, and V. K. Kaliappan, "Designing and verifying communication protocols using model driven architecture and spin model checker," *IEEE*, 2008.