Formal Consistency Checking over Specifications in Natural Languages

Rongjie Yan
State Key Laboratory of Computer Science,
Institute of Software,
Beijing, China
yrj@ios.ac.cn

Chih-Hong Cheng
Industrial Software Technologies
ABB Corporate Research,
Ladenburg, Germany
chih-hong.cheng@de.abb.com

Yesheng Chai
School of Computer Science & Technology,
Soochow University,
Suzhou, China
chaiys@ios.ac.cn

***Abstract*—Early stages of system development involve outlining desired features such as functionality, availability, or usability. Specifications are derived from these features that concretize vague ideas presented in natural languages. The challenge for the verification and validation of specifications arises from the syntax and semantic gap between different representations and the need of automatic tools. In this paper, we present a requirement-consistency maintenance framework to produce consistent representations. The first part is the automatic translation from natural languages describing functionalities to formal logic with an abstraction of time. It extends pure syntactic parsing by adding semantic reasoning and the support of partitioning input and output variables. The second part is the use of synthesis techniques to examine if the requirements are consistent in terms of realizability. When the process fails, the formulas that cause the inconsistency are reported to locate the problem.**

I. Introduction

Early stages of system development involve outlining desired features such as functionality, availability, or usability. The importance of early verification of high-level requirements can never be underestimated: Precise specifications can avoid overly frequent corrections in late developing phases, and they serve as a reference model or a test-case generator later in system and architecture design. Nevertheless, the challenge for the early verification process arises from the syntax and semantic gap between different representations. We consider a specification to have three views. Initial specifications are generated from these features that concretize vague ideas presented in natural languages. Precise specifications can be represented under the assist of logic. But for reference models or even test case generators, very often they are represented as programs [1], [2]. In many cases, one struggles to maintain the following two types of consistency.

- The *semantic* consistency between the intuitive meaning of a textual specification and its logic presentation.
- The *realizability* consistency between the intuitive meaning of a logic specification and a model or a test case generator - a logic specification should guarantee the existence of an implementation[1].

In this paper, we present a framework that endeavors to bring consistency to different representations of the high-level specification, via a *synthesis-based transformation*. The goal is to generate, from requirements in natural languages, to two other representations automatically, thereby ensuring consistency. We also call it *LTL-oriented*, as the intermediate format makes the use of linear temporal logic (LTL) [3].

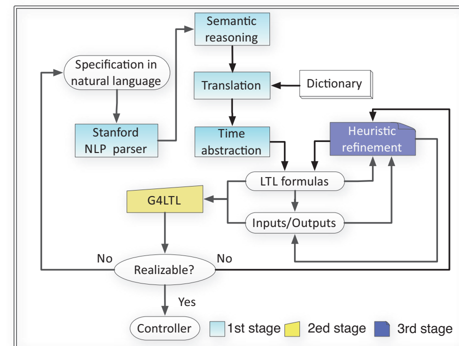

Fig. 1. The overall workflow of SpecCC.

The workflow of our framework (see Figure 1 for overview) involves a loop of three stages. The first part is a translation framework that can automatically turn requirements written in natural language into formulas formalized in LTL. This portion includes a heuristic partition of input and output variables, together with an appropriate abstraction of time. Importantly, apart from pure lexical parsing, we also introduce the use of dictionaries to infer the meaning. This allows the solver to reason status scenarios such as $\mathsf{on(light)} \equiv \neg\mathsf{off(light)}$, thereby avoiding the creation of two propositions. The second part is the use of LTL synthesis [4], [5], [6] to detect the realizability of specifications. The last part is a semi-automatic procedure that changes the physical interpretation of time and examines if the partition of input-output variables is reasonable. When changes from any of the two items arise, the specification is refined and is re-analyzed by the LTL synthesis engine.

The paper is structured as follows. After a brief summary of related work, we describe the CARA system [7] as a motivating example. We present the translation framework from natural to formal language in Section IV. Then, we summarize the consistency checking between formal language and implementation, together with the following refinement steps in Section V. Section VI explains the implementation of the prototype tool SpecCC (Specification Consistency Checking) and presents the results of applying the tool on a set of non-trivial examples. We conclude the paper in Section VII.

[1] E.g., consider the specification "*the output should always be the same as the input 3 time ticks from now*". Although it can be rewritten formally using LTL ($\mathbf{G}\,(\mathsf{output} \leftrightarrow \mathbf{XXX}\mathsf{input})$), it is unrealizable, as any implementation requires to have *clairvoyance* over the future events.

II. Related Work

Managing consistencies for requirements of different representations is essential in system modeling and subsequent implementation / refinement steps.

Translating requirements in natural languages to LTL has been investigated in many works, which is always based on a subset of natural language. There are approaches using property patterns and scopes to assist the specification of formal properties specified in LTL as well as other languages [8], [9]. Machine learning and data mining are also used to classify natural language into temporal requirements [10]. The translator implemented in [11] is one of the available tools, which translates specifications on hardware systems into LTL for model checking. In the research of consistency checking of requirements, one of the pioneering work in this domain is [12], which analyzes requirements expressed in the SCR (Software Cost Reduction) tabular notation. Li, Liu and He have generated pre and post conditions for system operations of UML requirements model to check requirement consistency based on the semantics [13]. Overall, none of above results supports the consistency checking between formal requirements and implementability.

In the community of formal methods, Uchitel, Brunet and Chechik have proposed a method to synthesize a set of safety properties in Fluent Linear Temporal Logic (FLTL) to generate partial behavior models to reduce the effort of model construction [14]. One of the closest work from ours is an application inside robotic domains [15], where the technique parses templates to generate LTL formulas and synthesizes a controller. However, the translated specification only supports a strict language subset (GR-1 [16]) that disallows the use of Until and strictly limits the use of Next to at most once. Contrarily, we allow the user to still present their requirements in English, while the synthesis engine supports full LTL and allows simple semantic reasoning. Another similar work [17] translates specifications in stylized natural language into various formalisms from an intermediate representation, and checks the realizability with synthesis analysis with GR-1 synthesis tool. As their framework allows translation to various models, their approach is not tailored for LTL synthesis. E.g., they do not propose automatic partition of input and output variables and leave the work to the end user.

III. Example: CARA System

We use the CARA (Computer-Aided Resuscitation Algorithm) infusion pump control system [7], [18] as a running example. It is a software system developed to drive a high output infusion pump used for fluid resuscitation of patients suffering from conditions that lead to hypotension, based on the collected data on blood pressure. The main functionality is to monitor and control the operations of the infusion pump, to drive resuscitating fluids into a patient's blood stream.

System Description. We focus on the three main modes (wait, manual and auto-control) of operation in CARA, and check the consistency of the specification on the three working modes.

The system is in the wait mode when the pump is off. It does not monitor the blood pressure or control the pump. The system enters the manual mode when the pump is initially turned on. In this level, the software only performs monitoring functions. If the power supply to the pump is lost, the control goes to a backup battery and triggers an alarm. To leave this mode, either the pump is turned off, or a button for the auto-control mode is pressed by the care-giver. That button is only enabled when the pump is in the normal operative mode. In the auto-control mode, CARA is responsible for monitoring the status lines from the pump and controlling the infusion rate. When the system is in the auto-control mode, it can use three sources of blood pressure data. Among the sources such as an arterial line, a pulse-wave and a cuff, the arterial line has the highest priority. That is, if the three sources are both available, the arterial line is used as the input. If the arterial line source is lost, the pulse-wave has a higher priority than the cuff.

The corresponding requirements from [7] are organized as the input of our framework. The following list enumerates some requirements as illustrating examples.[2]

Req-08 If Air Ok signal remains low, auto-control mode is terminated in 3 seconds.

Req-17 When auto-control mode is entered, eventually the cuff will be inflated.

Req-28 If a valid pressure is unavailable in 180 seconds, manual-mode should be triggered.

Req-32 If pulse_wave or arterial_line is available, and cuff is selected, corroboration is triggered.

Req-42 When auto-control mode is running, and the arterial_line or pulse_wave or cuff is lost, an alarm should sound in 60 seconds.

Req-44 If pulse_wave and arterial_line are unavailable, and cuff is selected, and blood_pressure is not valid, next manual_mode is started.

IV. Maintaining Consistencies between Natural Language and Formal Language

Maintaining consistencies between natural language and formal logic is obtained by automatic translation from natural language to LTL. In the following subsections, we review the definition of LTL, propose a restricted English grammar for syntactic parsing, and give the underlying algorithm for an extended reasoning on the semantic level and the techniques for abstracting time. Afterwards, we provide heuristics on partitioning input and output variables to apply the synthesis techniques in the second level.

A. LTL Syntax

Linear temporal logic [3] is a modal temporal logic whose modalities refer to time. It can express properties of paths in a computation tree. The formulas are built up from a set of atomic propositions (AP), logical operations and temporal modal logics. The set of supported LTL formulas are constructed from atomic propositions $p \in AP$ according to the following grammar.

$$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid \boldsymbol{X}\phi \mid \boldsymbol{F}\phi \mid \boldsymbol{G}\phi \mid \phi\boldsymbol{U}\phi \ ,$$

where $\boldsymbol{X}$ is the Next time operator, $\boldsymbol{F}$ is the Eventually operator, $\boldsymbol{G}$ is the Globally operator, and $\boldsymbol{U}$ is the Until operator. Given negation ($\neg$) and disjunction ($\vee$), we can define conjunction ($\wedge$), implication ($\rightarrow$) and equivalence ($\leftrightarrow$).

B. Grammar of the Structured English

Natural languages allow a rich diversity of sentence structures and easily cause semantic ambiguities. Hence, we do not provide a translator for the full set of natural languages. Instead, we select a more structured subset, whose constituting

[2] A detailed list of requirements is available at https://www.dropbox.com/sh/t67r3g5az94r90s/AABHAfn37O2aIE6wLTOy9JtLa?dl=0

grammar can encompass the needs of functional requirements in most cases. In this subset, we only support present, future and passive tenses with correct syntax according to the English grammar. To be concise, we only present positive form of the grammar. The negative form can be inferred accordingly. We first present the grammar of the structured English.

$sentence$ ::= $(subclause,)^*.(clauses).(,subclause)^*$
$subclause$ ::= $(subordinator).(clauses)$
$clauses$ ::= $(clause).[,(conjunction).(clause)]$
$clause$ ::= $[modifier].(subject).(predicate).[constraint]$
$subject$ ::= $substantives$
$substantives$ ::= $(substantive).[(conjunction).(substantives)]$
$substantive$ ::= $noun$
$predicates$ ::= $[modality].predicate$
$predicate$ ::= $verb \mid (be).participle \mid (be).(complement)$
$participle$ ::= $(verb)$.(ed) | $(verb)$.(ing)
$complement$ ::= $adjective \mid adverb$
$modality$ ::= shall |should |will |would |can |could |must
$subordinator$ ::= if | after | once | when | whenever | while | before | until | next
$modifier$ ::= globally | always | sometimes | eventually
$conjunction$ ::= and | or
$constraint$ ::= in t

where t is a time constant, * (star) means the presence of zero or more subcomponent, . (dot) means the composition of different components, and [ ] means the optional components. For *noun*, *verb*, *participle*, *adjective* and *adverb*, we do not decompose them any more. The proposed grammar allows extracting temporal operators according to the elements in *subordinator* and *modifier*.

According to the above definition, a sentence consists of at least one clause. A clause consists of at least one main phrase for the core meaning of the sentence, and optionally conditional or time clauses that extend the meaning. In addition to the skeleton shown in the above grammar, a main phrase may also contain filter constructions such as the and a that will be ignored in the translation. For example, for Requirement Req-17: " When auto-control mode is entered, eventually the cuff will be inflated.", the main clause is "eventually the cuff will be inflated", and "When auto-control mode is entered" is a time clause. In this requirement, "eventually" is a modifier for the temporal operator Eventually and "the" is a filter construction in the main phrase "eventually the cuff".

C. Lexical and Syntactic Parsing

A specification here is a set of sentences (requirements) in the structured English. For every sentence, first it is parsed by a natural language parser to extract all grammatical ingredients. Then according to the dependency relation extracted by the parser, the translator decomposes the sentence into clauses recursively to isolate the independent temporal units. After the decomposition, we construct a syntax tree for the ingredients of the sentence, to extract atomic propositions, and to infer the temporal relationship of individual temporal elements according to the subordinators and modifiers. Usually an atomic proposition comes from a subject and its predicate extracted by the dependency relation from the parser, i.e., in the form of *predicate_subject*, to combine a variable and its valuation. For multiple subjects connected with conjunctions, they are decomposed to generate different atomic propositions and then connected by the corresponding logic operators.

Reconsider the example used in the last subsection. Requirement Req-17 consists of two clauses. The corresponding

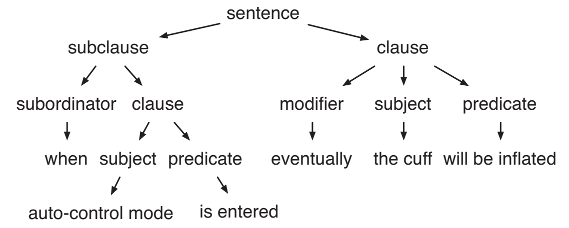

Fig. 2. The syntax tree of Requirement Req-17

syntax tree is illustrated in Figure 2. By removing tense information in the predicates, two atomic propositions, enter_auto-control_mode and inflate_cuff are extracted from the two clauses. Note that only one word will be taken as the subject in a clause in the translation process. To keep every proposition meaningful, we need to add "_" to contact relative words together for a complete subject in a requirement.

According to the existing study on the scopes and patterns, and their templates on LTL formula generation [19], we have selected some frequently used patterns (Universality and Existence), and the corresponding templates in our translator, to transform the elements in the syntax tree to the corresponding formulas. For example, according to the syntax tree for Req-17 and its keywords, modifier "eventually" is translated into unary temporal operator "$\boldsymbol{F}$", which will be joined with atomic proposition inflate_cuff. Subordinator "when" is translated into the logic operator "$\rightarrow$", to connect the two formulas from the two clauses. Accordingly, the temporal formula for Req-17 is $\boldsymbol{G}(\text{enter_auto-control_mode} \rightarrow \boldsymbol{F}\text{inflate_cuff})$.

D. Moving from Lexical Parsing towards Semantic Reasoning

To preserve semantic consistency and obtain succinct temporal formulas, additional to pure lexical parsing, we also adopt semantic reasoning over specifications in natural language. The semantic reasoning discussed here is to extract the relation between adjectives and adverbs (we call them *antonym candidates* in the rest of the paper) respectively according to their meaning in a specification, instead of semantic roles labelled by a parser. More precisely, after extracting the antonym candidates used in a specification, we group them in pairs of semantically contrasting words by looking up an antonym dictionary specified by users. With these semantically related words, we can reduce the number of atomic propositions used in the generated formulas, and avoid adding the assumptions on the mutual exclusive propositions.

We propose a two-step antonym extraction process to compute the pairs of antonyms in a specification. Intuitively, pairs of antonyms always come from the sentences with same subjects. Therefore, we first organize the antonym candidates related to the same subjects according to their dependency relation $\langle subject, dependent\rangle$ extracted by the natural language parser, where every word in the dependent set is initialized with color green. If the number of words in the dependent set for a subject is larger than one, we use an antonym dictionary to check whether semantically contrasting words exist in the same set. Otherwise, we continue to deal with other groups. The reason being that we cannot use the derived antonyms for the corresponding proposition reduction. The details on the reasoning process are depicted in Algorithm 1, where subject is to group elements depending on the same subjects, and wordset is to store the set of antonym candidates with the

Algorithm 1: Semantic reasoning

```
input : Specification S in natural language
output: pairs of antonyms in the specification
begin
1    wordset = ∅
     // extract subject dependent words, and store
        antonym candidates and the dependency relations
        in wordset and subject, respectively
2    subject = extract(S, wordset)
     for ∀s ∈ subject do
3        if |s.dep| > 1 then
             for ∀w ∈ s.dep do
                 if w.color == green then
4                    if wordset(w).antonym == ∅ then
5                        wordset(w).antonym = online(w)
6                    antonmy = s.dep ∩ wordset(w).antonym
7                    if antonmy ≠ ∅ then
                         w.color = blue
                         for w' ∈ antonym do
8                            w'.color = blue
9                            wordset(w').antonym =
                             wordset(w').antonym ∪ {w}
     return (subject, wordset)
```

extracted antonyms. To mark the status of dependent words in a subject during semantic reasoning, we use two colors, where blue for having found the words in the dependent set of the subject with contrasting meaning by consulting the dictionary, and green stands for the non-existence of antonyms in the set. These colors are indicators for our proposition reduction in the formula transformation process. That is, a word marked with green will be directly converted into atomic propositions with the corresponding subject. Otherwise, for pairs of antonyms related to the same subject, we first decide the positive and negative properties. Then we replace the words in negative meaning with the negative form of their antonyms.

In Algorithm 1, we assume that every antonym candidate can find its antonyms from an antonym dictionary. We first initialize wordset (Line 1) and extract the dependency relation from the specification (Line 2). In the *extract* function, the extracted antonym candidates are stored in wordset, where initially the sets of their antonyms are empty. The items in the dependent sets of subjects are initialized with green. Then for every extracted word w in subject s, if it is not analyzed before (Line 4), we look for its antonyms from the antonym dictionary and add the result to the antonym set of w in wordset (Line 5). If it has been processed, we check whether the intersection between its antonyms and the set of words in $s.dep$ is empty (Line 6). If the interaction is not empty, we say that there exists a pair of antonyms in $s.dep$, and mark the corresponding words in $s.dep$ with blue (Lines 7 and 8). Finally, we will complete the information for the antonyms of w (Line 9).

Consider Requirements 32 and 44 presented in Section III as an example. The subject pulse_wave has dependency relation with available, and unavailable. Without semantic reasoning, we will create two propositions as available_pulse_wave, and unavailable_pulse_wave. In our semantic reasoning process, we need to look up the antonyms for available and unavailable in the antonym dictionary. As the intersection between available and the set of antonyms for unavailable is not empty, we say that the two words constitute a pair of antonyms. Therefore, we can replace the proposition unavailable_pulse_wave by $\neg$ available_pulse_wave. Note that we cannot reason antonyms for verbs. The reason being that different actions capture different semantics and we cannot simply use their negation to replace the corresponding antonyms.

E. Time Counting and Abstraction

Requirements in natural language may contain timing constraints. In LTL, one can use Next operator to emulate discrete time. That is, decide unit time (e.g., 1 second), and define the elapse of unit time via a Next operator. For example, for Requirement Req-08:"If Air Ok signal remains low, auto-control mode is terminated in 3 seconds", its LTL formula is $\boldsymbol{G}(\neg$Air_ok_signal $\rightarrow \boldsymbol{XXX}$ terminate_auto-control_mode$)$, where one $\boldsymbol{X}$ is for one second.

Explicit enumerating time units brings a clearer mapping for requirement analysis. However, the larger the number of time units in a requirement, the longer of the translated formula with a higher complexity. To alleviate this problem, we present a rewriting technique that can abstract time with the consideration of all the requirements inside the specification. Intuitively, we can reduce the numbers of Next operators by dividing them with the *greatest common divisor (GCD)* for the lengths of successive Next operators in the requirements. The method is sound, i.e., a specification with the greatest common divisor reduction is realizable, if and only if the original specification is realizable.

For example, in Requirements Req-08, Req-28 and Req-42 in Section III, the lengths for successive Next operators are 3, 180 and 60, respectively. The greatest common divisor among the three numbers is 3. Therefore, after reduction, the lengths of the successive Next operators for the three requirements are 1, 60 and 20, respectively. So the corresponding formula for Req-08 becomes $\boldsymbol{G}(\neg$Air_ok_signal $\rightarrow$ $\boldsymbol{X}$ terminate_auto-control_mode$)$. From this example, one can observe that the reduction is quite conservative and still produces formula with huge amount of Next, thereby hindering the later synthesis process.

The complete algorithm overcomes the limitation via the introduction of *arrival errors*, borrowed from the concept of jitter[3]. Intuitively, an arrival error specifies, for an action, whether it is allowed for arriving later or earlier. This allows us to search for a better re-encoding of consecutive Next operators. With arrival errors, one can formulate the problem into an optimization problem (nonlinear with degree at most 2).

Formally, let $\Theta = \{\theta_0, \ldots, \theta_n\}$ be a set of numbers of successive Next operators in the transformed formulas from a specification where $\theta_i \neq \theta_j$ for any $i \neq j$. Let d be the devisor, θ'_i be the length of successive Next operators after the abstraction from θ_i, and Δ_i be the error introduced in the abstraction. With these notations, we obtain the following constraint system, for all $i \in \{0, \ldots, n\}$,

$$\theta_i = \theta'_i \times d + \Delta_i, -d < \Delta_i < d, d \text{ and } \theta'_i \in \mathbb{Z}^+, \text{and } \Delta_i \in \mathbb{Z}, \quad (1)$$

where $\mathbb{Z}$ is the set of integers, and $\mathbb{Z}^+$ is the set of positive integers including zero. Intuitively, $\theta_i = \theta'_i \times d + \Delta_i$ means that the algorithm rewrites θ_i consecutive Next operators into θ'_i Next operators, via a factor of d. This generates an error of

[3]Jitter is the undesired deviation from true periodicity of an assumed periodic signal in electronics and telecommunications.

Δ_i. If a Δ_i is positive, it means that the corresponding event happens earlier in the rewritten specification. Otherwise, the event arrives later than expected. In the above GCD case, Δ_i should always be 0.

Based on the above constraint system, we derive a two-objective optimization problem: Reduce the use of Next (i.e., reduce θ'_i), and minimize the controlled error (i.e., reduce $|\Delta_i|$, where $|\Delta_i|$ stands for the absolute value of Δ_i).

$$\text{minimize } \sum_{i=0}^{n} \theta'_i, \quad \text{minimize } \sum_{i=0}^{n} |\Delta_i| \tag{2}$$

As the first objective is more important, we let the user specify an allowed upper-bound B for $\sum_{i=0}^{n} |\Delta_i|$, and restrict the domain for every Δ_i such that either $\Delta_i \in [0, d]$ or $\Delta_i \in [-d, 0]$ (i.e., arrival error can be either early or late, but not both). The second objective can be rewritten as a single integer linear constraint (as one can remove the absolute sign). This reduces the two-objective optimization problem into a single-objective optimization problem, which can be efficiently solved by modern SMT solvers (e.g., Yices 2 [20]) via bit-blasting.

Back to the example and consider again Requirements Req-08, Req-28 and Req-42 in Section III. The set for the lengths of Next operators in these requirements is $\{3, 180, 60\}$, that is, $\theta_0 = 3, \theta_1 = 180$ and $\theta_2 = 60$. If we specify $\Delta_i \geq 0$ for $0 \leq i \leq 2$, and set B to 5, we could obtain an optimal result with $d = 60$, and

$$\theta'_0 = 0, \quad \theta'_1 = 3, \quad \theta'_2 = 1, \quad \Delta_0 = 3, \quad \Delta_1 = 0, \quad \Delta_2 = 0.$$

F. Input and Output Variable Partition

To check the consistencies of a specification, the translator also needs to distinguish the propositions for input and output variables in an LTL formula. Very commonly, we find that every line of a specification discusses how the system should response, when encountering a certain scenario. This can be understood as having an implication between an occurring event and the corresponding response. Sometimes, the response might need to endure until another environment event appears. Generally speaking, for left-hand parts in an implication, or for right-hand parts of the Until operator, we assume that the constituting variables are input variables. If a proposition in positive form appears in the both sides of such operators, it is assumed as an output.

We use this idea to analyze every requirement and maintain, for each requirement, two disjoint sets for input and output variables. Then when all requirements are analyzed, a unification is performed by checking consistencies among variable sets. Whenever there exists a conflict (i.e., a variable is considered to be an input variable in one requirement but an output variable in another requirement), the translator regards it as an output. If no input variable exists after the partition, we select one randomly from the set of output variables as the input. After the partition, the translator also asks the user for the assistance. For example, for the generated formula from Req-32: ***G***((available_pulse_wave || available_arterial_line) && select_cuff → trigger_corroboration), we can conclude that available_pulse_wave, available_arterial_line and select_cuff are the input variables, and the output variable is trigger_corroboration.

V. Maintaining Consistencies between Formal Language and Implementability

A. Synthesis

In real world applications, there are always a large amount of requirements describing functionality of the desired system. The inconsistency between various requirements may arise between different sequences of events (actions) defined by the semantics of the corresponding LTL formulas. As the synthesis technique is capable of generating a controller over a set of LTL formulas, the inconsistency between different LTL formulas will prevent the synthesis tool from creating a controller. Therefore, we can borrow the idea of LTL-based synthesis techniques to check the inconsistency among the requirements represented in LTL formulas.

B. Heuristic refinement

We use G4LTL [6] as our underlying LTL synthesis engine, which automatically checks consistency between LTL formulas transformed from the requirements in a specification, and reports the inconsistency between neighbored requirements. However, it cannot always automatically provide reasons causing the inconsistency, especially when the pair of requirements causing inconsistencies are not neighbored. We suggest the following manual strategy to isolate the problem.

- Locate the pair of inconsistent requirements. The process starts from a subset of consistent formulas. We can add more formulas continuously to the subset to check which one is not consistent with the subset. Once we have located the problem, we could filter out other formulas that do not contain any propositions of the located formulas.
- Adjust the existing input and output variable partition. As the input and output partition method is heuristic, we may have to adjust the partition if G4LTL reports the inconsistency. The propositions belonging to the intermediated variables in the located formulas are targets to be adjusted.
- Modify the requirements. When we cannot guarantee the consistency by adjusting the partition on input and output variables, the specification is not consistent indeed. Therefore, we analyze the semantics of the formulas to extract the reason of inconsistency, and modify the requirements to ensure the consistency of the specification.

VI. Implementation and Evaluation

We have implemented a research prototype for the above mentioned framework. For the language translation part, the implementation extends the Stanford NLP parser [21] to analyze the requirements in the structured English to extract its ingredients for LTL generation. In the consistency checking part, we integrate the synthesis library G4LTL [6].

We have selected three case studies coming from different application areas, to maintain the consistencies in the two levels, and check the specification scalability that we can handle in the second level.

CARA We first have considered Requirements 1, 7, 8, 13, 16, 17, 20, 28, 32, 34, 42, 44, 48, 49 and 54 among seventy requirements that are related to the switches between working modes and the corresponding responsibilities for the CARA system as the input of our framework. We also adopted the detailed specifications for the three components (Pump Monitor, Blood Pressure Monitor

TABLE I. EXPERIMENTAL RESULTS

Name	No.	Specification	formulas	in	out	time(s)
CARA	0	Working mode and switching	30	22	28	34
	1	Pump Monitor	20	9	14	2
	2.1.1	BPM: cuff detector	14	13	12	1
	2.1.2	BPM: AL detector	15	11	14	2
	2.1.3	BPM: pulse wave detector	14	9	12	1
	2.2.1	BPM: initial auto control	16	14	15	1
	2.2.2	BPM: first corroboration	19	11	16	29
	2.2.3	BPM: valid ctrl blood pressure	13	11	10	2
	2.2.4	BPM: cuff source handler	11	9	10	2
	2.2.5	BPM: arterial line blood pressure	16	9	13	1
	2.2.6	BPM: arterial line corroboration	12	8	13	1
	2.2.7	BPM: pulse wave handler	20	10	21	23
	3.1	(PA) Model ctrl algorithm	9	15	11	3
	3.2	(PA) Polling algorithm	56	12	20	11
TELE	1	Shopping	29	11	24	8
	2	Article processing	17	3	13	1
	3	On-line reservation	6	3	4	1
	4	Information	15	8	14	1
	5	Local bulletin board	17	7	16	1
Robot	1	A robot with 4 rooms	9	2	5	1
	2	A robot with 9 rooms	14	2	10	1
	3	Two robots with 5 rooms	25	2	11	7

(BPM), Polling Algorithms (PA)) in the CARA system for evaluation.

TELEPROMISE We have also considered the functional specification for the TELEPROMISE project demonstrator as a case study[4], which covers five generic applications: Shopping application, Article processing application, On-line reservation application, Information application and Local bulletin board application.

Rescue robot We have modified the rescue robot scenario used in [15] as the third case study. The responsibility of the robots in this scenario is to look for the injured people and take them to a medic who is in some room. Different numbers of rooms and robots have been considered here, with the constraint that two robots cannot be in the same room at the same time.

Table I lists the results for the corresponding component specifications and the scale of every specification (number of lines, and number of I/Os), as well as the time consumption (in seconds) for realizability checking of requirements in LTL formulas. One can observe that the tool allows handling problems with more than 50 formulas. The performance of G4LTL are sensitive to the number of subformulas, the number of input and output variables, and the length of a formula, e.g. the third case with different numbers of robots. Therefore, the time consumption for realizability checking of various specifications are quite different. For the consistency maintenance between natural language and formal language, the time consumption is linear to the number of requirements, which is ignored here.

In the CARA case study, the specifications for the transformation between three working modes and the three components are consistent. During consistency checking for the TELECOMPRISE case study, G4LTL failed to generate controllers for the last two specifications. The failure was caused by the classification of input and output variables. After locating the problem and modifying the input/ output variable partition, the specifications are consistent.

VII. CONCLUSION

We have proposed a framework for ensuring consistencies between different representations of specifications. The framework maintains semantic consistencies between oral and formal specifications, while ensuring the implementability using synthesis. With the framework, we propose a time extraction mechanism, input-output partition heuristics, and an extension from pure syntactic parsing of sentences to reason on the semantics level. The approach is evaluated under a rich set of examples with positive results.

For future work, we will increase the maturity level of our implementation by capturing the semantics in natural language to better understand the meaning of sentences thus avoiding manual matching between different words, and by using learning techniques for more generic natural languages.

[4]Available at: www.eng.utoledo.edu/eecs

ACKNOWLEDGMENT

This work has been partly funded by National Science Foundation of China under Grant No. 61100074, 91418204.

REFERENCES

[1] C. D. Nguyen, A. Marchetto, and P. Tonella, "Combining model-based and combinatorial testing for effective test case generation," in *ISSTA*, 2012, pp. 100–110.

[2] A. Riesco, "Test-case generation for maude functional modules," in *WADT*, 2012, pp. 287–301.

[3] A. Pnueli, "The temporal logic of programs," in *FOCS*, 1977, pp. 46–57.

[4] B. Jobstmann and R. Bloem, "Optimizations for LTL synthesis," in *FMCAD*, 2006, pp. 117–124.

[5] A. Bohy, V. Bruyre, E. Filiot, N. Jin, and J.-F. Raskin, "Acacia+, a tool for LTL synthesis," in *CAV*. Springer, 2012, pp. 652–657.

[6] C.-H. Cheng, C.-H. Huang, H. Ruess, and S. Stattlemann, "G4LTL-ST: Automatic generation of PLC programs," in *CAV*. Springer, 2014, pp. 541–549.

[7] "CARA increment 3 (CARA-Tagged-Requirements), version 1.2," March 21, 2001.

[8] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Property specification patterns for finite-state verification," in *FMSP*, 1998, pp. 7–15.

[9] O. Mondragon, A. Q. Gates, and S. Roach, "Prospec: Support for elicitation and formal specification of software properties," *Electron. Notes Theor. Comput. Sci.*, vol. 89, no. 2, pp. 67–88, 2003.

[10] A. P. Nikora and G. Balcom, "Automated identification of LTL patterns in natural language requirements," in *ISSRE*, 2009, pp. 185–194.

[11] L. Zilka, "Temporal logic for man. Bachelor's thesis," Brno, But FIT, 2010.

[12] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw, "Automated consistency checking of requirements specifications," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 3, pp. 231–261, Jul. 1996.

[13] X. Li, Z. Liu, and J. He, "Consistency checking of UML requirements," in *ICECCS*, 2005, pp. 411–420.

[14] S. Uchitel, G. Brunet, and M. Chechik, "Synthesis of partial behavior models from properties and scenarios," *IEEE Trans. Software Eng.*, vol. 35, no. 3, pp. 384–406, 2009.

[15] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas, "Translating structured english to robot controllers," *Advanced Robotics*, vol. 22, no. 12, pp. 1343–1359, 2008.

[16] N. Piterman, A. Pnueli, and Y. Saar, "Synthesis of reactive (1) designs," in *VMCAI*, 2006, pp. 364–380.

[17] S. Ghosh, D. Elenius, W. Li, P. Lincoln, N. Shankar, and W. Steiner, "ARSENAL: Automatically extracting requirements specifications from natural language," *CoRR*, vol. abs/1403.3142, 2014.

[18] A. Ray and R. Cleaveland, "Unit verification: the CARA experience," *Int. J. Softw. Tools Technol. Transf.*, vol. 5, no. 4, pp. 351–369, 2004.

[19] S. Salamah, A. Gates, S. Roach, and O. Mondragon, "Verifying pattern-generated LTL formulas: a case study," in *SPIN*, 2005, pp. 200–220.

[20] B. Dutertre, "Yices 2.2," in *CAV*. Springer, 2014, pp. 737–744.

[21] M.-C. de Marneffe, B. MacCartney, and C. D. Manning, "Generating typed dependency parses from phrase structure parses," in *SLT*, 2006.