# Synthesis of Formal Specifications From Requirements for Refinement-based Real Time Object Code Verification

Eman M. Al-Qtiemat*, Sudarshan K. Srinivasan*, Zeyad A. Al-Odat *, Mohana Asha Latha Dubasi*, Sana Shuja†
*Electrical and Computer Engineering, North Dakota State University,
Fargo, ND, USA
†Department of Electrical Engineering, COMSATS University,
Islamabad, Pakistan
Emails: *eman.alqtiemat@ndsu.edu, *sudarshan.srinivasan@ndsu.edu, *zeyad.alodat@ndsu.edu, *dubasi.asha@gmail.com,
†SanaShuja@comsats.edu.pk

*Abstract*—**Formal verification methods have been shown to be very effective in finding corner case bugs and ensuring safety of embedded software systems. The use of formal verification requires a specification, which is typically a high-level mathematical model that defines the correct behavior of the system to be verified. However, embedded software requirements are typically described in natural language. Transforming these requirements to formal specifications is currently a big gap. While there is some work in this area, this paper proposes solutions to address this gap in the context of refinement-based verification, a class of formal methods that have shown to be effective for embedded object code verification. The proposed approach also addresses both functional and timing requirements and has been demonstrated in the context of safety requirements for software control of infusion pumps.**

***Keywords–requirements analysis; safety-critical IoT embedded devices; timing specifications; timing transition systems; formal model; formal verification.***

## I. Introduction

Ensuring the correctness of control software used in safety-critical embedded devices is still an ongoing challenge [1]. For example, from 2001 to 2017, the Food and Drug Administration (FDA) has issued 54 Class-1 recalls on infusion pumps (medical devices used to deliver controlled doses of fluid medications to patients intravenously) due to software issues [2]. Class-1 recalls are applied to medical device models whose use can cause serious adverse health consequences or death. With the advent of IoT, such safety-critical embedded devices incorporate a whole slew of additional functionality to interface with the network and other components, in addition to their core control functions. These additional functions significantly exacerbate the challenge of ensuring that the core functionality of the control software is correct and intact.

Critical devices such as insulin pump still have safeness issues which need valuable software amendments to assure the reliability on design level, this can be handled by either appending new safety insurance specifications to fix existing hazards, or modifying some defined specifications that cause faulty behaviours. Since critical devices are considered as real time systems, most of their specifications have well defined timing constraints must be met else wise the system will fail. This paper works with both functional and timing specifications (called functional and timing requirements), they are basically written in natural language and need to be transformed into a formal model, then it can be tested using a formal verification method. The use of formal verification has become an industry standard when addressing software correctness of safety-critical devices. There are many success stories and commercial adoption of formal verification processes. Examples include Intel [3], Microsoft [4] and [5], and Airbus [6].

Refinement-based verification [7] is a formal verification technology that has been demonstrated to be applicable to the verification of embedded control software at the object-code level [8]. In formal verification and refinement-based verification, typically the design artifact to be verified is called the implementation and the specification is a formal model that captures the correct functionality of the implementation. The goal of refinement-based verification is to mathematically prove that the implementation behaves correctly as defined by the specification. In refinement-based verification, both the implementation and specification are modeled as transition systems and timed transition systems if timing specifications are existed.

One of the key features of refinement-based is the use of refinement maps, which are functions that map implementation states to specification states. In practice, these refinement maps have a very favorable property in that they abstract out behaviors of the implementation not relevant to the specification, but only after determining that these additional behaviors do not actually impact the behaviors of the implementation relevant to the specification. This property of refinement maps makes the refinement-based verification very suitable for the verification of control software used in IoT devices as refinement maps can be used to abstract out the additional functionality of software in IoT devices; again, only after determining that these additional functionality are not impacting the behavior of the core functionality of the implementation as defined by the specification.

One of the crucial challenges in applying refinement-based verification to commercial devices is the availability of formal specifications. For commercial devices, typically, the specification of a device is given as natural language requirements. There are many approaches towards transforming natural language requirements to formal specifications, however none targeted towards refinement-based verification. In this paper, we present methodologies for transforming natural language requirements (both functional and timing) into formal specifications that can be used in the context of refinement-based verification.

The rest of the paper is organized as follows. An overview of the background is presented in Section II. Section III details the related work. A formal model describing the synthesis procedure of functional requirements is presented in Section IV, while Section V presents a different formal model describing the synthesis procedure of timing requirements. Section VI details the case study. Section VII gives the verification results for the proposed formal model. Conclusions and direction for future work are noted in Section VIII.

## II. Background

This section explores the parsing tree, the definition of transition systems and the definition of timed transition systems as key terms related to our work.

### A. Parsing tree

A parse tree is an ordered tree that pictorially represents how words in a sentence are connected to each other. The connection between each word in the sentence gives the *syntactic categories* for the sentence. The parsing process represents the syntactic analysis of a sentence in natural language. For example, when the parsing process is applied on a simple sentence like "Adam eats banana", the parse tree categorizes the two parts of speech: N for nouns (Adam, banana) and V for the verb (eats). Here N, V are the syntactic categories. The parsing process is considered to be a preprocessing step for some applications, where natural language should be converted into other forms. Usually, the system requirements are written in natural language, which needs to be converted into a structural form that can then be used to create the transition system(s) (explained in Section II-B). Enju [9] is an English consistency-based parser, which can process very long complex sentences like system requirements using an accurate analysis (the accuracy relation is around 90 percent of news articles and bio-medical papers). Besides, Enju is a high-speed parser with less than 500 msec per sentence. The output is the resulting tree in an XML format which is considered to be one of the commonly used formats by various applications. As will be described later, the case study used to describe the proposed methodology is from the bio-medical area, Enju was the perfect tool as the natural language processing (NLP) parser.

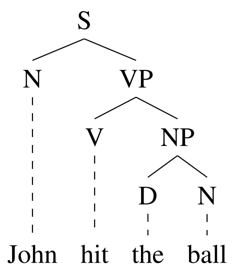


Figure 1. A simple example of a parsing tree using Enju parser [10].

Figure 1 shows a simple tree example using Enju. Here, Enju distinguishes between terminal nodes (John is a terminal node) and non-terminal nodes (VP is a verb phrase). The abbreviations of the syntactic categories of Figure 1 are: S stands for sentence (the head of the tree), N stands for noun, VP stands for verb phrase (which is a subtree), NP stands for noun phrase, V stands for verb, and finally D stands for determiner (comes with noun phrases). Using these syntactic categories, we have developed an extraction technique that would help in translating the natural language to a formal model of the requirements.

### B. Transition systems

The implementation and specification in refinement-based verification are represented using Transition Systems (TSs) [7], [8]. The definition of a TS is given below:

*Definition 1:* A TS $M = \langle S, R, L \rangle$ is a three tuple in which $S$ denotes the set of states, $R \subseteq SXS$ is the transition relation that provides the transition between states, and $L$ is a labeling function that describes what is visible at each state.

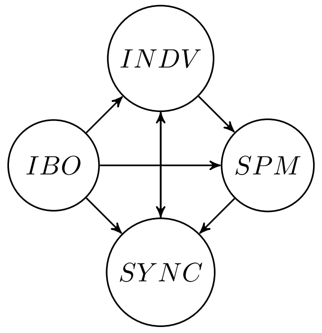


Figure 2. An example of a transition system (TS).

An Atomic Proposition (AP) is a statement that can be evaluated to be either true or false. The labeling function maps state to the APs that are true in every state. An example of a TS is shown in Figure 2. Here $S$ = {IBO, SPM, SYNC, INDV}, $R$ = {(IBO, SPM), (SPM, SYNC), (SYNC, INDV), (INDV, SYNC), (INDV, SPM), (IBO, INDV)} and, $L(SPM)$ represents the atomic propositions that are true for the SPM state. Similarly, labeling function can be applied to all the states in this TS.

### C. Timed Transition Systems

Some applications have requirements with timing conditions on the state's transitions called as timing requirements. Timing requirements explain the system behaviour under some timing constraints. Timing constraints are very important especially if we deal with a critical real time systems. As mentioned in the previous section (Sec II-B), transition systems are used to represent the implementation and specification in refinement-based verification, however they do not contain timing requirements. Hence, in the verification of real time systems that contain timing constraints, timed transition systems (TTSs) [8] are used to represent the implementation and specification.

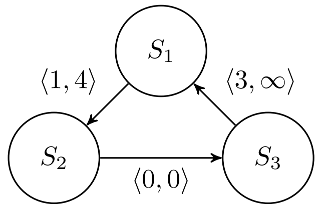


Figure 3. An example of a timed transition system (TTS)

*Definition 2:* A TTS $M_t = \langle S, R_t, L \rangle$ is a three tuple in which $S$ denotes the set of states and $L$ is a labeling function that describes what is visible at each state. The state transition $R_t$ has the form of $\langle x, y, l_t, u_t \rangle$ where $x, y \in S$ and $l_t, u_t \in N$ represents the lower and upper bounds as the timing condition for the transition.

Figure 3 shows an example of a timed transition system that consists of three states { S1, S2, S3 }, for instance; if the system is in state $S1$ it can go to state $S2$ only between 1 and 4 units of time, while going from $S2$ to $S3$ the time is zero meaning that it should happen immediately, and so on.

## III. Related Work

In the last few years, there has been a tremendous growth in finding the optimal technique of requirement transformation into a formal model. While most of them proposed system-driven models, our approach is user-driven to ensure a safe product.

Automatic Requirements Specification Extraction from Natural Language (ARSENAL) [11] is a system based framework that applies some semantic parsers in multi-level to get the grammatical relations between words in the requirement. ARSENAL transforms natural language requirements into formal and logical forms expressed in Symbolic Analysis Laboratory (SAL) (a formal language to describe concurrent systems), and Linear Temporal Logic (LTL) (a mathematical language that describes linear time properties) respectively. The LTL formulas are then used to build the SAL model. Multiple validation checks are applied on Natural Language Processing (NLP) stage and LTL formulas to check for their correctness. However, ARSENAL records some inaccuracies in NLP stage that need a user intervention.

Aceituna et al. [12] have proposed a front end framework that builds a model to exhibit the system behavior (for synchronous systems only) and help in creating temporal logic properties automatically. This framework can be used before applying the model checking technique, it exposes accidental scenarios in the requirements. The framework is designed in a manner that helps in understanding the errors in a non-technical manner for users who do not have a formal background. In contrast, our work does not need the temporal logic in defining the specifications for a model.

A semantic parser has been developed by Harris [13] to extract a formal behavioral description from natural language specifications. The proposed semantic parser was employed to extract key information describing bus transactions. The natural language descriptions are then converted to verilog (a hardware description language) tasks.

Kress-Gazit et al. [14] have proposed a human-robot interface to translate natural language specification into motions. This interface allows a user to instruct the robot using a controller. LTL formulas are employed to formalize the desired behavior requested by the user.

An approach supporting property elucidation (called PROPEL) has been introduced by Smith et al. [15], it provides templates that capture properties for creating property pattern. Natural language and finite state automation are used to represent the templates.

Two approaches have been proposed by Shimizu [16] to solve the ambiguity of natural language specifications using formal specification. The first approach simplifies the formal specification development for the popular PCI bus protocol and the Intel Itanium bus protocol. The second approach explains how formal specifications can help in automating many processes that are now done manually.

A natural language parsing technique has been used with the default reasoning, which is a requirement formalism to support requirement development, this work helps stakeholders to easily deal with requirements in a formal manner, in addition, a method has been proposed for discovering any existed requirements inconsistencies. A prototype tool called CARL was used for implementation and verification by Zowghi et al. [17].

Gervasi et al. [18] have also worked on solving the requirement's inconsistencies issues by using a well-known formalism called monotonic logic, it has been used especially for requirement's transformation. Multiple natural language processing tools [19]–[22] in additional to grammatical analysis methodologies for requirement's development have been done to get requirements in a formal manner.

Bouyer et al. [23] have recently presented a survey on timed automata and how it can be applied for model checking of real-time systems. This survey has summarized the work that has been done since the inception of timed automata in the early 1990s till now. The timing information in real-time models is expressed as temporal logic. However, the survey does not specify gathering timing information from natural language requirements, which has been the focus of our paper.

Knorreck et al. [24] have presented a graphical tool called AVATAR-TEPE (Automated Verification of reAl Time softwARe - TEmporal Property Expression Language), in which the logical and temporal properties are expressed in formal language. This tool can perform all tasks from requirement capture to verification in one language and in one environment. However, the tool requires the knowledge of logical and temporal properties to verify the application. The tool is heavily based on property modeling.

A standardized testing method for distributed real-time cyber-physical systems (CPS) has been proposed by Shrivastava et al. [25]. Temporal properties have been used to express the timing constraints. Peters et al. [26] have proposed a new language that considers timing requirement and checks for errors in the description of the timing constraints. Kang et al. [27] have presented a model-driven approach to verify the timing requirements for automotive systems at the design level. However, in all these works, gathering the timing constraints from natural language requirements, which has been the focus of this paper, has not been addressed.

Carvalho et al. [28] have proposed a symbolic model for translating natural language requirements to a formal model which consider time. Model-based testing techniques are then applied to these formal models. Hassine [29] has presented a formal framework to describe, simulate and analyze real-time systems. This framework considers timing requirements. However, this proposed framework is yet to be applied on large scale industrial projects. In these works, even though the timing requirements are considered, none of the these works are targeted at refinement based verification.

The main advantages of our work over prior algorithms in requirements engineering is its ability to generate a full formal model directly from natural language requirements by an expert supervision to emphasis on the safety transformation. Also, our work does not require that the expert user know any temporal logic languages which has been case for most of the current literature.

## IV. Formal Model Synthesis Procedure for Functional Requirements

The first step of computing the TSs is to extract the APs from the requirements. We have developed three Atomic Proposition Extraction Rules (APERs) that work on the parse tree of the requirement obtained from Enju. The resulting APs are then used to compute the states and transitions. The APERs are described next.

### A. Atomic Proposition Extraction Rule 1 (APER 1)

APER 1 is based on the hypothesis that noun phrases in a requirement correspond to APs. Each subtree of the parse tree with an NX root (called an NX head) corresponds to a noun phrase and hence an AP. Therefore, APER 1 computes the subtrees corresponding to NX heads. If NX heads are nested, then the highest-level NX head is used to compute the AP. The terminal nodes of the subtree are conjoined together to form the noun phrase. APER 1 returns AP-list, which is the set of APs corresponding to a parse tree.

**Procedure 1** APER1

```
Require: Parse-tree
1: AP-list ← $\emptyset$ ;
2: for each $n \in$ TerminalNodes(Parse-tree) do
3:     Start-cat = head(head($n$));
4:     if Start-cat = NX then
5:         $X$ = Sub-tree(Start-cat);
6:         while (head($X$) = NX ) $\vee$ (head($X$) = COOD)
                 $\vee$ (head($X$) =NX-COOD ) do
7:             $X$ = Sub-tree(head($X$));
8:         AP-list ← AP-list $\cup$ TerminalNodes(X) ;
```

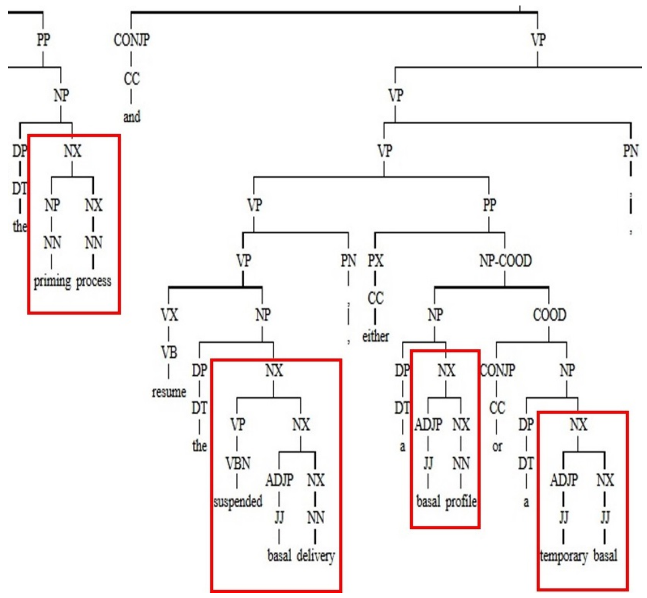


Figure 4. An Enju parsing tree portion shows some resulting APs by applying APER 1.

We now describe the procedure corresponding to APER 1 in detail. Firstly, AP-List is initialized to the empty set (line 1). The procedure then iterates through each terminal node $n$ (line 2). The head of a node is its parent. If a terminal node is part of an NX subtree, its level two head will be marked as NX, which is checked in line 3. The level-two NX node of the terminal node is stored in variable State-cat. If the Start-cat is of NX category (line 4), a function called Sub-tree is used to get the resulting subtree (line 5), which is stored in variable X. A while loop is used to traverse the tree of X upwards checking if the head syntactic category is NX or COOD or NX-COOD (line 6). Only when one of the conditions is satisfied the subtree is stored in X (line 7). The terminal nodes of the resulting sub tree 'X' will be added to AP-List as a new suggested AP (line 8). Figure 4 gives a sub tree example for APER 1. Note that APER 1 may result in the same AP being duplicated. Duplicates are checked and removed from the AP list in the overall approach.

As shown in Figure 4, the terminal nodes 'the' and 'priming' does not have head(head(n)) = NX. The first terminal node that has the NX category is 'process'. Traversing upwards, the NX related categories gives us the subtree which contains 'priming process'. This now constitutes the first AP for this part of requirement. Applying the APER 1 rule on the visible part of the sentence in Figure 4 gives us the following APs: 'priming process', 'suspended basal profile', 'basal profile', and 'temporary basal'.

### B. Atomic Proposition Extraction Rule 2 (APER 2)

APER 2 and APER 3 correspond to the two other parse tree patterns that also lead to noun phrases. APER 2 examines the parse tree for noun categories along with its upper verb head. APs will be the conjoined terminal nodes of the resulting sub tree. APER 2 states that APs are the terminal nodes under the head VP passing through NX (or its related phrases such as NX-COOD, COOD), NP (or its related phrases NP-COOD, COOD), and VX phrase.

**Procedure 2** APER 2

```
Require: Parse-tree
1: AP-list ← $\emptyset$ ;
2: for each $n \in$ TerminalNodes(Parse-tree) do
3:     Start-cat = head(head($n$));
4:     $X_1$ ← $\emptyset$;
5:     if Start-cat = NX then
6:         $X$ = Sub-tree(Start-cat);
7:         while (head($X$) = NX ) $\vee$ (head($X$) = COOD)
                 $\vee$ (head($X$) =NX-COOD ) do
8:             $X$= Sub-tree(head($X$));
9:         while (head($X$) = NP) $\vee$ (head($X$) = COOD)
                 $\vee$ (head($X$) = NP-COOD) do
10:            $X_1$ = Sub-tree(head($X$));
11:        if (head($X_1$) = VX) $\wedge$ (head(head($X_1$)) = VP) then
12:            $X$ = Sub-tree(head(head($X_1$));
13:        else
14:            if (head($X_1$) = VP) then
15:                $X$ = Sub-tree(head($X_1$);
16:        AP-list ← AP-list $\cup$ TerminalNodes(X);
```

APER 2 is built on top of APER 1 to get atomic propositions for requirements that APER 1 is not able to collect. While APER 1 looks only for APs that are noun phrases, APER 2 looks for noun phrases that are further characterized by verb phrases. For example, if APER 1 finds the AP "suspended basal delivery," APER 2 will find "resume the suspended basal delivery."

APER 1 and APER 2 have the same algorithmic flow until finding the sub tree of X that is the top NX head (line 8). However, APER 2 does not consider the resulting X to be an AP like APER 1 does. Instead, X is the input of the next step. A while loop is used to search if the head category of $X$ is in NP category or one of its related phrases (line 9). Only when the while loop condition is true, the new sub-tree is stored temporarily in the variable $X_1$ (line 10), where $X_1$ is a temporary variable initialized to null (line 4). This ensures that X does not change in this step for future use. The search for VX and VP categories is to be performed only when $X_1$ is not null.

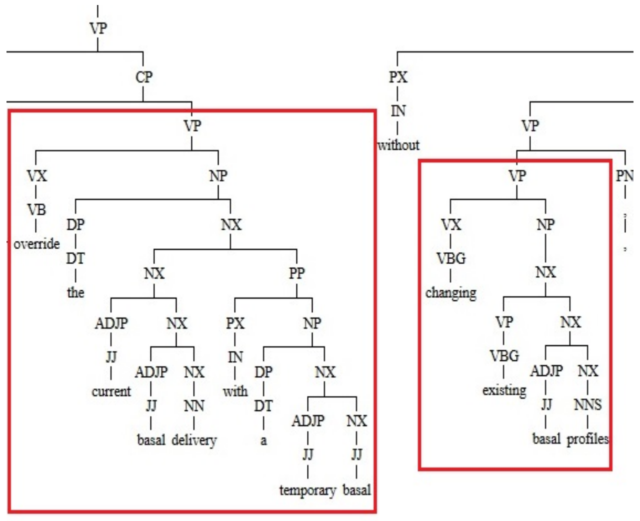


Figure 5. An Enju parsing tree portion shows some resulting APs by applying APER 2.

On the successful completion of NP category search, the search for VX category followed by VP categories is performed (line 11). When the if condition is satisfied, X is updated with the new sub-tree (line 12). In the case of failure of the if condition in line 11, a new search for VP category is performed on the head of NP category sub-tree (line 14). On success, X is updated with the new sub-tree (line 15). If either of the if conditions (line 11 and line 14) fail, then X will remain as the sub-tree of NX category. The terminal nodes of the resulting subtree in X is appended to the AP-list (line 16). Figure 5 shows a resulting sub tree example by applying APER 2.

Figure 5 shows that the procedure starts from left to right looking for level two NX nodes and traversing upward until higher NX nodes are accounted for. NP phrases are selected to expand the tree. Then choosing the upper level which is VP in this particular case (sometimes its VX $\rightarrow$ VP). The output of APER 2 for this tree portion is 'override the current basal delivery with a temporary basal', and 'changing existing basal profiles'.

## C. Atomic Proposition Extraction Rule 3 (APER 3)

APER 3 is built on top of APER 2, it explores the verb head levels in the parse tree like APER 2, but APER 3 eliminates some verb phrases that is not part of APs. This elimination is done based on the head of the VP category as illustrated in Procedure 3 below.

APER 3 and APER 2 have the same stream up to line 10. The algorithm starts with initializing temporary variables $X_1$ and Y to null (line 4). The search for syntactic categories start with the top NX phrase (line 7) and the resultant sub tree is stored in X (line 8). Then, the search begins for the top NP phrase (line 9) and the resultant sub tree is stored in $X_1$ (line 10) since the sub tree in X is needed for future use. As in APER2, the search for either VX phrase followed by VP phrase or just VP phrase is performed on $X_1$ and the resultant sub tree is stored in Y (lines 11-15). If and only if Y is not empty then the check on the head syntactic category is performed to ensure that it does not contain CP or COOD categories. In this case, X gets only the right child (line 16-18) i.e. the left child of Y is pruned. On the other hand, if Y has a CP or COOD head, X value will be updated to be equal to Y (line 20). Finally, terminal nodes of the resulting sub tree X will be saved in the AP-list as a new AP. The pruning process (line 18) is done to remove some action verbs which are not part of an AP.

**Procedure 3** APER 3

```
Require: Parse-tree
 1: AP-list ← $\emptyset$ ;
 2: for each $n \in$ TerminalNodes(Parse-tree) do
 3:     Start-cat = head(head($n$));
 4:     $X_1 \leftarrow \emptyset$ , $Y \leftarrow \emptyset$;
 5:     if Start-cat = NX then
 6:         $X$ = Sub-tree(Start-cat);
 7:         while (head($X$) = NX) $\vee$ (head($X$) = COOD)
                  $\vee$ (head($X$) =NX-COOD ) do
 8:             $X$ = Sub-tree(head($X$));
 9:         while (head($X$) = NP) $\vee$ (head($X$) = COOD)
                  $\vee$ (head($X$) = NP-COOD) do
10:             $X_1$ = Sub-tree(head($X$));
11:         if (head($X_1$) = VX) $\wedge$ (head(head($X_1$)) = VP) then
12:             $Y$ = Sub-tree(head(head($X_1$));
13:         else
14:             if (head($X_1$) = VP) then
15:                 $Y$ = Sub-tree(head($X_1$);
16:         if ($Y \neq \emptyset$) then
17:             if head($Y$) $\neq$ CP) $\wedge$ (head($Y$) $\neq$ COOD) then
18:                 $X$ = Sub-tree(RightChild($Y$));
19:             else
20:                 $X = Y$;
21:         AP-list ← AP-list $\cup$ TerminalNodes(X);
```

Like APER2, APER3 also works on verb head categories. However, APER3 has some pruning techniques to remove parts of the sentence that should not be part of an AP. Consider the snippet in Figure 6, the sub tree "issue an alert" is subjected to left branch pruning to remove the verb 'issue' since such verbs do not add value in the AP. According to the algorithm, since the head node of VP is COOD, only the terminal nodes of the right child are considered as an AP. Applying APER

3 on the visible part of the requirement in Figure 6 gives the following APs: 'pump', 'an alert', and 'deny the request'.

The proposed APERs may be used individually or in combination depending on the system requirement and model functionally. However, no one rule is considered to be the best for all models because of the natural language structure.

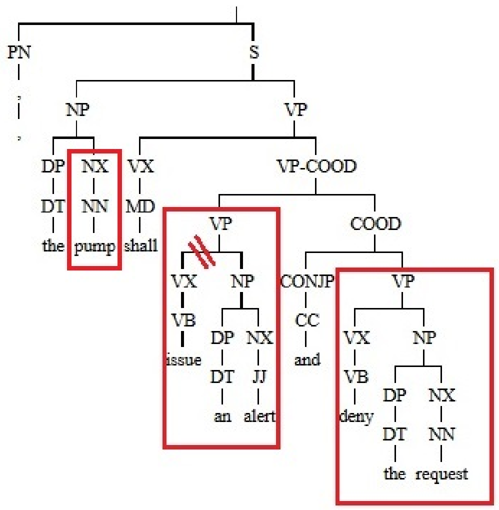


Figure 6. An Enju parsing tree portion shows some resulting APs using APER 3.

## D. *High-Level Procedure for Specification Transition System Synthesis*

**Procedure 4** Procedure for synthesizing TSs from system requirements

```
Require: set of requirements (System-requirements)
1:  TS-set ← ∅ ;
2:  for each  $Req$ ∈ System-requirements  do
3:      Parse-tree ← Get(Req_tree.xml);
4:      AP-list ← APER(Parse-tree);
5:      AP-list ← Eliminate_Dup(AP-list);
6:      AP-list ← USR_IN(AP-List);
7:      AP-truth-table ← Relation(AP-list);
8:      AP-truth-table ← USR_IN(AP-truth-table);
9:      S-list ← ∅;
10:     for each  A ∈ AP-truth-table  do
11:         S-list[$i$] = A$_i$ ;
12:     S-list ← USR_IN(S-list);
13:     T ← CreateT(S-list);
14:     T ← USR_IN(T);
15:     TS ← CreateTS(T, S-list);
16:     TS-set ← TS-set U TS;
17:     return TS-set;
```

Procedure 4 shows the overall flow for computing the TSs. A set of system requirements in natural language are fed as input to the procedure. TS-set is the output of the procedure and will contain the set of transition systems that capture the input requirements as a formal model. TS-set is initialized to null (line 1). Each requirement is input to the Enju parser. The parser gives an xml file as output. A function called Get is used to obtain the xml file into the variable Parse-tree (line 3). The xml output in Parse-tree is subjected to the proposed APERs, which give the atomic propositions (APs) as output. APs are stored in the AP-list (line 4). Each requirement is subject to all APERs and the AP-list obtained is the union of the APs produced by each of the rules. The output obtained by using the APERs may contain duplicates, which are eliminated by using the function Eliminate_Dup (line 5). AP-list is then subjected to an expert user check, where the AP(s) might be appended, eliminated or revised based on the expert user's domain knowledge (line 6). Some of the APs maybe expressible as a Boolean function of other APs.

Therefore, next, a truth table (AP-truth-table) is created, where each row corresponds to an AP from AP-list and each column also corresponds to an AP from AP-list (line 7). Each entry in the table is a Boolean value (true or false). Completing the truth table determines the relationship of each AP with the other APs in the AP-list. The truth table is completed by the expert user (line 8). The list of states for the input requirements are stored in the variable S-list. S-list is initialized to null (line 9). Each truth table entry (A) is defined to be a single state in the transition system (line 10). This heuristic has worked well in practice. S-list is subjected to expert user input (line 12).

The transitions of the TS are computed next. The list of transitions (T) is initialized to a transition between every two states using function 'CreateT' (line 13). The transition list is subjected to expert user input (line 14). A transition system (TS) is constructed using the CreateTS function, which takes the transitions (T) and the list of states (S-list) as input (line 15). This transition system (TS) is then added to the transition system set (TS-set) (line 16). The procedure finally returns a set of transition systems for all the requirements in an application (line 17).

## V. Formal Model Synthesis Procedure for timing requirements

In this section, the approach is extended to deal with timing requirements. When synthesizing transition systems (TSs), the core activity was the extraction of APs. For synthesizing timed transition system, the core activity is the extraction of APs and TCs. An additional extraction rule is developed, that can be applied on timed requirements not only to get APs but also to extract the timing constraints (TCs) on each state transition.

### A. *Atomic Proposition and Timing Constrains Extraction Rule (APTCER) for Timed Transition System*

This section explains a new proposed rule that analyzes timing requirements to get APs with their corresponding TCs as a base for building TTSs. This rule called Atomic Proposition and Timing Constrains Extraction Rule (APTCER) is specified as Procedure 5 and works as follows. First, the timing requirement is split into smaller phrases that are individually analyzed (lines 1-14 of Procedure 5). These phrases are called Timed Based Sentences (TBSs). Each resulting phrase is then analyzed to extract the APs and TCs in that phrase (lines 15-38 of Procedure 5). The list of APs and TSs are stored in $\langle AP - list, TC - list \rangle$.

APTCER takes the parse tree of the timing requirement as input. The parse tree is obtained by applying the Enju parser on the timing requirement. APTCER initilizes the list of TBSs (TBS-list) to the empty list (line 1). APTCER then searches for sub-trees with root as "S" and with left child of "NP" and right child as "VP" (lines 2-5). Each such sub-tree is a TBS.

Note that TBSs can be nested in that there can be a TBS inside of a TBS. The nested TBSs need to partitioned and analyzed individually. This is done by searching for sub-trees inside the TBS with "SCP" or "S" roots. Such sub trees are cut out and the resulting TBS is returned (lines 7-13).

Next, the TBSs are analyzed to extract the APs and TCs. The extraction is performed by analyzing both the left child and the right child of the TBS. The left and right sub-trees are assigned to variables A and B, respectively (lines 19 and 23). Then APER 1 is used to analyze both A and B. Through empirical observation, it has been determined that the APs extracted by APER 1 from sub-tree A (line 21, 22) corresponds to APs but the APs extracted by APER 1 from sub-tree B (line 25, 26) corresponds to TCs. The resulting AP-list and TC-list are corresponds to one TBS (line 27), the TBS's pair is saved in the final TBS-list (line 28).

Applying lines 1-14 of APTCER on the requirement in Figure 7 gives three TBSs, they are shown in separate red boxes. While the rest of the algorithm (lines 15-38) works on each TBS to find it's AP-list and TC-list. The left sentence has one AP which is "air-line-alarm" and one TC which is "maximum delay time of x minutes". The resulting AP and TC will be saved as a pair. This helps in identify that the AP and TC are correlated, which is used to determine the transition for which the TC should be applied. More specifically, the TC will be applied to a transition from a state in which the corresponding AP is true. Overall the three TBS from Figure 7 give the following. AP-list is: 'air-in-line alarm', 'air bubbles larger than y L', and 'insulin administrations'. TC-list will have one TC: maximum delay time of x seconds which related to the AP of 'air-in-line alarm' as one pair.

Note that a TBS can correspond to more than one AP and more than one TC. For example, Figure 8 shows a TBS that has two APs (in red boxes) and one TC (in a green box).

## *B. High-Level Procedure for Specification Timed Transition System Synthesis*

Procedure 6 shows the overall flow for computing the TTSs. A set of natural language timing requirements are input to the procedure. TTS-set is the output of the procedure and will contain the set of timed transition systems that capture the input requirements as a formal model.

TTS-set is initialized to null (line 1). Each timing requirement is input to the Enju parser. The parser gives an xml file as output. A function called Get is used to obtain the xml file into the variable Parse-tree (line 3). The xml output in Parse-tree is subjected to our proposed APTCER, which gives the TBS-list that are pairs of atomic propositions and their related timing constrains lists (line 4). The synthesizing procedure then iterates through all TBSs (line 5) to get thier corresponding pair of APs and TCs (line 6).

AP-lists is subjected to an expert user check, where the APs might be appended, eliminated or revised based on the expert users domain knowledge (line 7). Some of the APs maybe expressible as a Boolean function of other APs. Therefore, next, a truth table (AP-truth-table) is created, where each row corresponds to an AP from AP-lists and each column also corresponds to an AP from AP-lists (line 8). Each entry in the table is a Boolean value (true or false). Completing the truth table determines the relationship of each AP with the other APs in the AP-lists. The truth table is completed by the expert user (line 9). TC-list is then checked by the expert user, where some TCs might be appended, eliminated or revised based on the expert users domain knowledge (line 10).

**Procedure 5** APTCER

```
Require: Parse-tree
 1: TBS-list ← $\emptyset$ ;
 2: for each $Head - Cat \in$ Head(Parse-tree) do
 3:     if Head-Cat = S then
 4:         if (Left-Child (S)= NP $\vee$ NP-COOD ) $\wedge$
                                      (Right-Child (S)= VP) then
 5:             TBS = Sub-tree (S);
 6:             for each Child-Head (TBS) do
 7:                 if Child-Head (TBS) = SCP then
 8:                     Cut-Sub-tree (SCP);
 9:                     return TBS;
10:                 else
11:                     if Child-Head (TBS) = S then
12:                         Cut-Sub-tree (S);
13:                         return TBS;
14:             TBS-list ← TBS-list $\cup$ TBS;
15: $k \leftarrow \emptyset$;
16: for each $TBS \in$ TBS-list) do
17:     K = k + 1;
18:     $A \leftarrow \emptyset$ , $B \leftarrow \emptyset$;
19:     A = Sub-tree (left-Child (TBS));
20:     $AP - list_k \leftarrow \emptyset$;
21:     APER 1 (A) → AP-list;
22:     $AP - list_k \leftarrow$ AP-list ;
23:     B = Sub-tree (Right-Child (TBS));
24:     $TC - list_k \leftarrow \emptyset$;
25:     APER 1 (B) → AP-list;
26:     $TC - list_k \leftarrow$ AP-list ;
27:     $TBS_k = \langle AP - list_k, TC - list_k \rangle$;
28:     TBS-list ← TBS-list $\cup$ $TBS_k$;
```

Next, the states and transitions of the TTS are computed. S-list variable (list of states) is initialized to null (line 11). Each truth table entry (A) (line 12) is defined to be a single state in the transition system (line 13). S-list is subjected to expert user input (line 14). The transitions of the TTS are computed next. The list of transitions (T) is initialized to a transition between every two states using function CreateT (line 15). The transition list is subjected to expert user input (line 16) where some transitions might be pruned. A function called 'Apply-TC-list' is applied to link each TC to all transitions emanating from states in which the corresponding APs are true, based on the TBS pair → TBS $\langle AP - list, TC - list \rangle$ (line 17). The expert user will confirm, modify, or apply the TC on specific transition/s based on his domain knowledge (line 18). For the remaining transitions that do not have any timing bounds, the timing bounds are open from zero to infinity $\langle 0, \infty \rangle$. For this reason a new function called 'Apply-TC-bounds' is applied on each transition that has no TC (line 19).

A timed transition system (TTS) is constructed using the CreateTS function as in procedure 4, this function takes the transitions (T) linked with their timing conditions and the list of states (S-list) as input (line 20) to create a TTS. The resulting TTS is then added to the timing transition system set (TTS-set) (line 21). The procedure finally returns a set of timing transition systems for all timing requirements that have been fed to the algorithm (line 22).

**Procedure 6** Procedure for synthesizing TTSs from timing requirements

```
Require: set of requirements (Timed-requirements)
 1: TTS-set ← ∅ ;
 2: for each  $Req$ ∈ Timed-requirements  do
 3:     Parse-tree ← Get(Req_tree.xml);
 4:     TBS-list ← APTCER(Parse-tree);
 5:     for each  $TBS$ ∈ TBS-list do
 6:         Get ($\langle AP-list, TC-list\rangle$);
 7:     AP-list ← USR_IN(AP-list);
 8:     AP-truth-table ← Relation(AP-list);
 9:     AP-truth-table ← USR_IN(AP-truth-table);
10:     TC-list ← USR_IN(TC-list);
11:     S-list ← ∅;
12:     for each  A ∈ AP-truth-table  do
13:         S-list[$i$] = A$_i$ ;
14:     S-list ← USR_IN(S-list);
15:     T ← CreateT(S-list);
16:     T ← USR_IN(T);
17:     T ← Apply-TC-list ;
18:     T ← USR_IN(TC);
19:     T ← Apply-TC-bounds $\langle 0, \infty\rangle$;
20:     TTS ← CreateTS(T, S-list);
21:     TTS-set ← TTS-set U TTS;
22:     return TTS-set;
```

## VI. Case Study: Generic Insulin Infusion Pump (GIIP)

Insulin pump is a medical device that delivers doses of insulin 24 hours a day to patients with diabetes. It is typically used to keep the blood glucose level in an acceptable range. Overdose of insulin can lead to low blood sugar that can lead to coma/death. Therefore, the insulin pump is a safety-critical device.

The Generic Insulin Infusion Pump (GIIP) has been proposed [30], which lists a set of safety requirements for insulin pumps. We use these safety requirements to explain our approach. GIIP has proposed a list of both functional and timing requirements, examples will be given about both cases.

### A. Functional requirements of GIIP

GIIP model abstracts requirements that explains how specific critical behaviour of the system can be controlled, functional requirements are introduced to solve common hazards in the insulin pump's market that might happen during insulin administration and not related to specific timing constraints.

As an example, consider requirement 1.8.2 (from [30]) which is needed to address a hazard that may happen in the suspension mode of the pump. Suspension mode can occur when the pump may be in refill or priming or insulin delivery processes. The insulin pump has two type of insulin deliveries: bolus and basal. Bolus is a high insulin rate that is recommended in case of low blood glucose level.

**Requirement 1.8.2:** *When the pump is in suspension mode, insulin deliveries shall be prohibited. Any incomplete bolus delivery shall be stopped and shall not be resumed after the suspension.*

From safety requirement 1.8.2, it is clear that the pump should not resume a suspended bolus automatically after returning from suspension since they would be an unexpected amount of insulin.

**Requirement 1.8.5:** *When the pump resumes from suspension, calculations shall be performed to synchronize insulin used and remaining reservoir volume.*

Requirement 1.8.5 is an extension of how the pump should function after returning from the suspension mode. Here two requirements are needed to address one safety hazard. When algorithm 4 is applied on these two requirements, the first step is collecting the APs by using the extraction rules. Applying APER 2 on 1.8.2 gives: ”pump”, ”suspension mode”, ”insulin deliveries”, ”incomplete bolus delivery”, and ”suspension”. Applying APER 2 on 1.8.5 gives: ”pump”, ”suspension”, ”calculations”, and ”synchronize insulin used and remaining reservoir volume”. Next, duplicate APs are to be removed. This eliminates 'pump' and 'suspension' from the AP-list. Now, the expert user intervenes for manipulating the AP-list, where APs can be deleted, modified or even inserted based on the expert user's domain knowledge. This yields the final AP-list as ”suspension mode” (SPM), ”insulin deliveries” (INDV), ”incomplete bolus delivery” (IBO) and ”synchronize insulin used and remaining reservoir volume” (SYNC). Next, the AP-truth-table to define relations between APs is constructed as shown in Table I.

TABLE I. AP-TRUTH-TABLE FOR REQUIREMENT 1.8.2 AND 1.8.5 FROM AP-LIST

| **APs** → ↓ | SPM | INDV | IBO | SYNC |
|---|---|---|---|---|
| SPM | T | F | F | F |
| INDV | F | T | F | F |
| IBO | F | T | T | F |
| SYNC | F | F | F | T |

Here, each row represents a state. For example, SPM represents a state where suspension mode is true, IBO is false, INDV is false, and SYNC is also false; which emphasizes that insulin bolus should not be active during suspension.

Finally, Procedure 4 applies transitions between every two states as shown in Figure 9a. The expert user will approve or remove some unacceptable transitions. Figure 9b shows the final transition system.

### B. Timing requirements of GIIP

The application of APTCER to some timing requirements of GIIP are described next. Timing requirements are also critical to be preserved. In GIIP, a motor controls the fluid injection and therefore the fluid flow rate and dosage. The motor is in turn controlled by software and the speed of the motor is time controlled by the software. The timing requirements of GIIP are also safety-critical because if the software violates these requirements, the dosage can be affected. Overdose or under dose of medicines can be very harmful or even fatal to the patient.

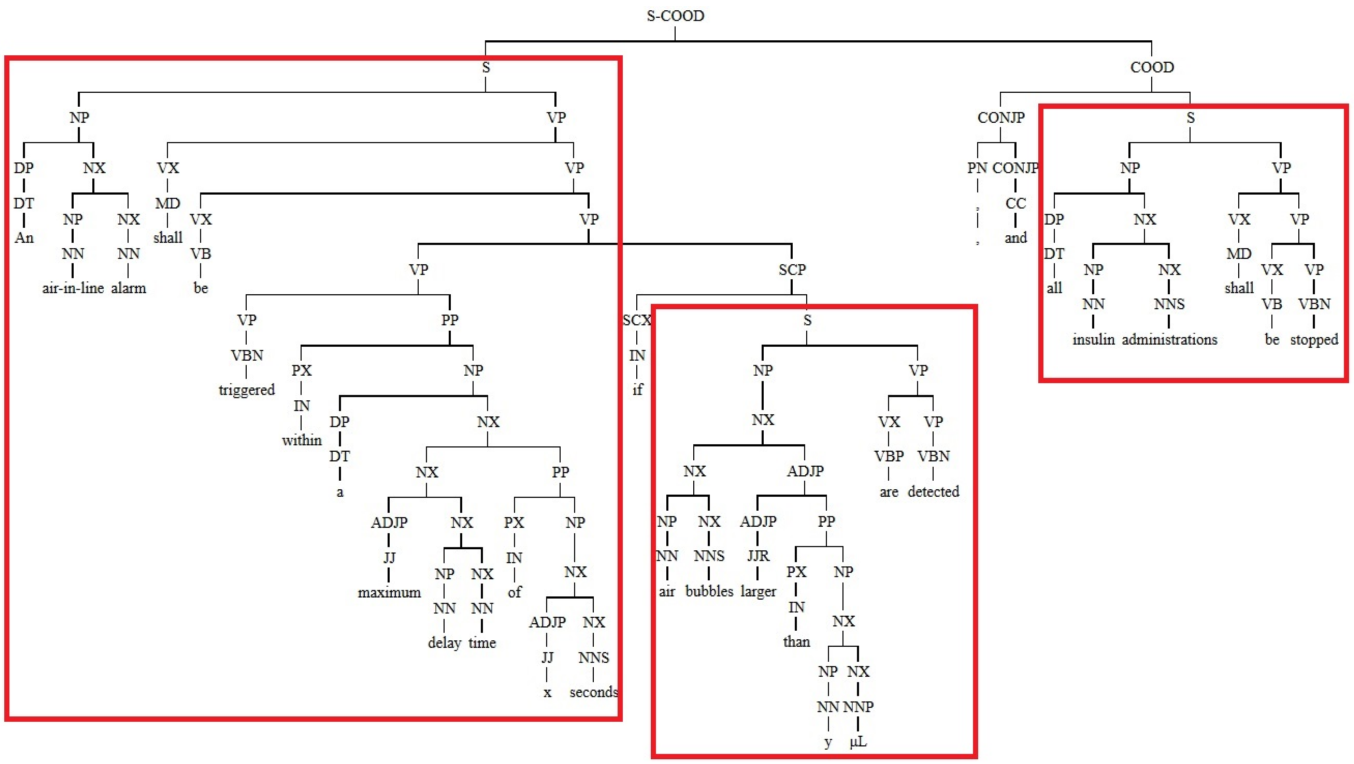


Figure 7. An Enju parsing tree shows three resulting TBSs after applying APTCER.

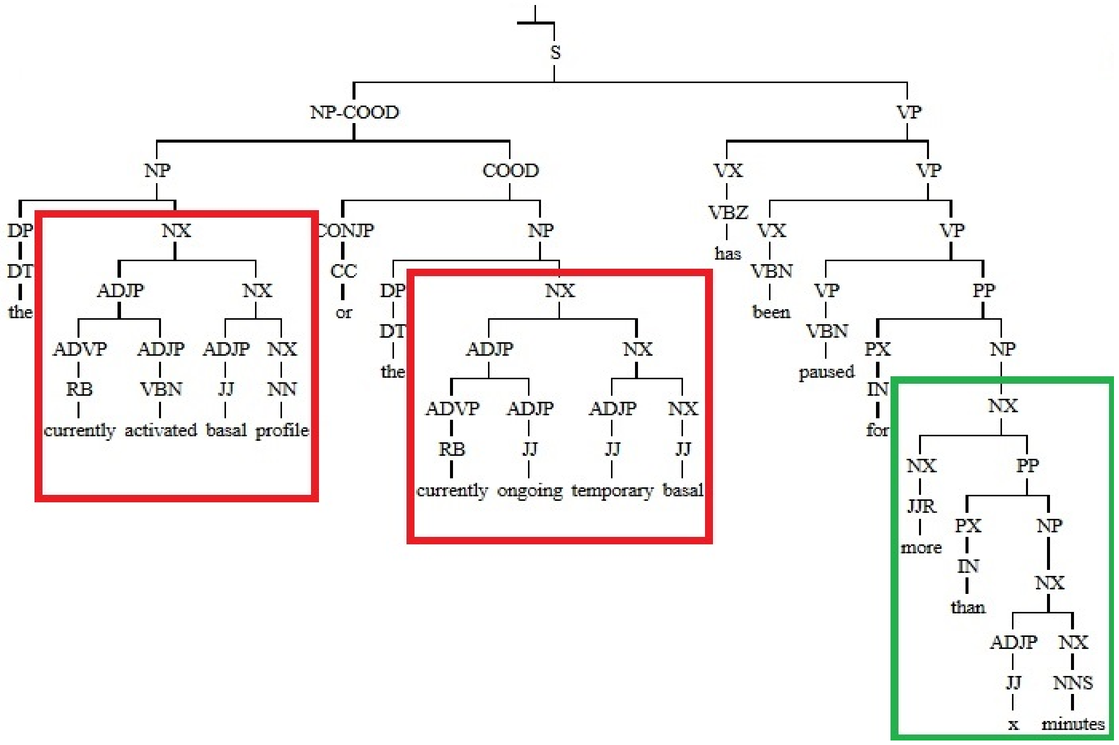


Figure 8. An Enju parsing tree portion shows the resulting TBS $\langle AP - list, TC - list \rangle$) after applying APTCER.

TABLE II. RESULTING TRANSITION SYSTEMS BY APPLYING PROCEDURE 4 AND APERS ON A SET OF SYSTEM REQUIREMENTS

| Req. NO. | APER | Total No. of APs | No. of APs Without DP | User input | | | Final | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | AP added | AP removed | AP modified | APs | states | transitions |
| 1.1.1 | 1 | 10 | 10 | 0 | 6 | 0 | 5 | 4 | 5 |
| | 2 | 10 | 10 | 0 | 5 | 0 | | | |
| | 3 | 10 | 10 | 0 | 6 | 0 | | | |
| 1.1.3 | 1 | 7 | 7 | 0 | 3 | 2 | 4 | 4 | 4 |
| | 2 | 7 | 7 | 0 | 3 | 2 | | | |
| | 3 | 7 | 7 | 0 | 3 | 1 | | | |
| 1.2.4 , 1.2.6, 1.2.7 | 1 | 24 | 12 | 3 | 5 | 1 | 10 | 10 | 14 |
| | 2 | 24 | 18 | 0 | 8 | 0 | | | |
| | 3 | 24 | 16 | 2 | 8 | 0 | | | |
| 1.3.5 | 1 | 11 | 6 | 1 | 3 | 0 | 4 | 4 | 4 |
| | 2 | 11 | 8 | 0 | 4 | 1 | | | |
| | 3 | 11 | 8 | 1 | 5 | 0 | | | |
| 1.8.2, 1.8.5 | 1 | 9 | 7 | 1 | 3 | 1 | 4 | 4 | 5 |
| | 2 | 9 | 7 | 0 | 3 | 0 | | | |
| | 3 | 9 | 7 | 0 | 3 | 0 | | | |
| 2.2.2, 2.2.3 | 1 | 6 | 6 | 0 | 3 | 1 | 3 | 3 | 4 |
| | 2 | 7 | 6 | 0 | 3 | 1 | | | |
| | 3 | 7 | 6 | 0 | 3 | 2 | | | |
| 3.1.1 | 1 | 15 | 14 | 0 | 9 | 0 | 5 | 3 | 3 |
| | 2 | 14 | 12 | 0 | 7 | 0 | | | |
| | 3 | 14 | 13 | 0 | 8 | 0 | | | |
| 3.2.5 | 1 | 10 | 9 | 0 | 7 | 2 | 3 | 3 | 3 |
| | 2 | 7 | 7 | 0 | 4 | 1 | | | |
| | 3 | 7 | 7 | 0 | 4 | 1 | | | |
| 3.2.7 | 1 | 4 | 4 | 0 | 1 | 0 | 3 | 3 | 3 |
| | 2 | 4 | 4 | 0 | 1 | 1 | | | |
| | 3 | 4 | 4 | 0 | 1 | 0 | | | |

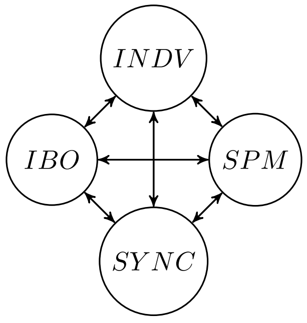
As an example, consider requirement 1.6.1 (from [30]) which helps patients to be aware of the occurrence of an air in line hazard. Air in line hazard is the presence of air bubbles in the pump above the acceptable range. The requirement states that if the air in line problem occurred during insulin delivery, an air in line alarm should start in a time not more than x minutes, in addition, every ongoing insulin delivery must be stopped. The alarm will give the patient a warning that a problem is going to happen, so the patient will interact with the pump and solve the issue to prevent incorrect insulin doses or other problems.

**Requirement 1.6.1:** *An air-in-line alarm shall be triggered within a maximum delay time of x seconds if air bubbles larger than y $\mu L$ are detected, and all insulin administrations shall be stopped.*

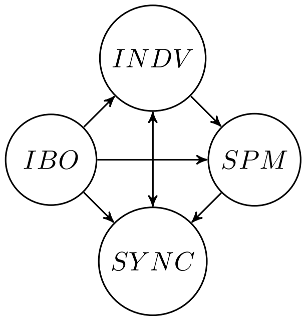
When procedure 6 is applied to this requirement, the first step is collecting the lists of TBS by applying APTCER, which gives three separate TBSs. TBS1 is "An air-in-line alarm shall be triggered within a maximum delay time of x seconds", while TBS2 is "air bubbles larger than y $\mu$L are detected", and TBS3 is "all insulin administrations shall be stopped".

Next, the AP-list and TC-list for each TBS is computed. AP-list contains: "air-in-line alarm", "air bubbles larger than y $\mu$L", and "insulin administrations". TC-list contains: "maximum delay time of x seconds" which is related to the AP: "air-in-line alarm" in TBS1.

Now, the expert user intervenes to manipulate the AP-list, where APs can be deleted, modified or even inserted based on the expert users domain knowledge. This yields the final AP-list as "air-in-line alarm" (ALRM), "air bubbles larger than y $\mu$L" (AIRB), "insulin administrations" (INSAD). Next, the AP-truth-table to define relations between APs is constructed as shown in Table III.

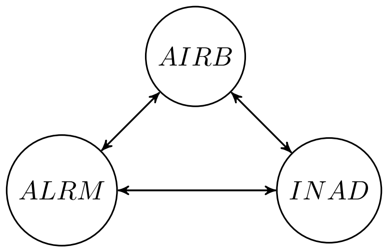


(a) TS with all suggested transitions.

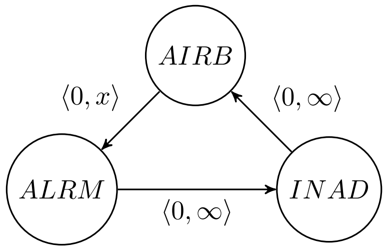


(b) TS after removing some transitions.

Figure 9. Finite state machine for suspension mode requirements (1.8.1 and 1.8.5).



(a) TTS with all suggested transitions.



(b) TTS after applying the TCs and removing some transitions.

Figure 10. Timed finite state machine for air-in-line requirement (1.6.1).

TABLE III. AP-TRUTH-TABLE FOR TIMING REQUIREMENT 1.6.1 FROM AP-LIST

| **APs** → ↓ | AIRB | INAD | ALRM |
|---|---|---|---|
| AIRB | T | T | F |
| INAD | F | T | F |
| ALRM | F | F | T |

As in Table I, each row in the Table III represents a state. For example, AIRB represents a state where AIRB is true, INAD is also true, while ALRM is false; which explains the problem of having air bubbles while an insulin administration is given to the patient. Now, the user can make changes to the TC-list which may have a TC that corresponds to one or more APs. After the states are computed, the expert user can add or modify any of the states if needed.

Then, Procedure 6 applies transitions between every two states, the expert user will approve or remove some unacceptable transitions as shown in Figure 10a. After following Procedure 6, the final TTS is shown in Figure 10b

## VII. Results analysis

An evaluation process is applied on the resulting TSs and TTSs by using NuSMV and UPPAAL model checkers respectively. Firslty, evaluation of the first approach (APERs and Procedure 4) for TSs is performed using the NuSMV model checker. A model checker is a tool that can check if a TS or a TTS satisfies a set of properties. The properties have to be expressed in a temporal logic. Here, we have used CTL to express the properties. The CTL properties are written manually for each of the requirements that are subjected to our approach. NuSMV is used to check if the TSs synthesized by the first approach satisfied the CTL properties corresponding to each functional requirement.

Secondly, UPPAAL is used to verify the resulting TTSs by applying APTCER and Procedure 6 (the second approach). UPPAAL is a tool that can verify real time systems and is based on the timed automata theory [31]. UPPAAL is used to check if the TTSs synthesized by the second approach satisfied the CTL properties corresponding to each timing requirement.

Table II shows the results of applying Procedure 4 on a number of GIIP requirements. The requirement numbers in the table are from [30]. All the final TSs satisfied their corresponding CTL properties. Each requirement or set of requirements (listed in column 1) have been subjected to the extraction rules (column 2), where column 3 shows the total number of APs resulting from each extraction rule. Column 4 gives the number of APs after removing the duplicate APs. In addition, a record of the suggested expert user intervention for adding, removing or modifying the APs is shown in column 5. The final number of APs, states, and transitions are shown in column 6.

As shown in Table II, when a requirement is subjected to the APERs, the resultant output from each APER may be different even though the number of APs is the same. For requirements 1.8.2 and 1.8.5, although applying APER1, APER2, and APER3 give the same number of APs, APER1 gives different list of APs from APER2 and APER3.

Table IV presents the results of applying Procedure 6 on a number of GIIP timing requirements from [30]. All applied CTL properties are satisfied by the resulting TTSs. The listed requirements (column 1) are subjected to the APTCER which gives list of TBSs for each requirement (column 2). column 3 and 4 show the number of the resulting APs and TCs respectively. Column 5 shows the pair of AP-list and TC-list. As in Table II, column 6 has the user interventions of appending, deleting, or modifying the AP-lists. The final TTS's components are shown in column 7: the number of APs, the number of states, and finally the number of transitions between states.

## VIII. Conclusion

The key ideas of our approach for transforming requirements into transition systems and timed transitions systems

TABLE IV. RESULTING TIMED TRANSITION SYSTEMS BY APPLYING PROCEDURE 6 AND APTCER ON A SET OF TIMING REQUIREMENTS

| **Req. NO.** | **Total No. of TBSs** | **Total No. of APs** | **Total No. of TCs** | **(AP,TC)** | **User input** | | | **Final** | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | AP added | AP removed | AP modified | APs | states | transitions |
| 1.2.8 | 2 | 3 | 3 | $\langle 2,1\rangle$ $\langle 1,2\rangle$ | 1 | 0 | 1 | 4 | 3 | 4 |
| 1.6.1 | 3 | 3 | 1 | $\langle 1,1\rangle$ | 0 | 0 | 0 | 3 | 3 | 7 |
| 1.8.4 | 2 | 3 | 2 | $\langle 1,1\rangle$ $\langle 1,1\rangle$ | 1 | 1 | 2 | 3 | 3 | 4 |
| 2.2.1 | 4 | 6 | 1 | $\langle 4,1\rangle$ | 0 | 0 | 1 | 6 | 3 | 4 |

are the following. The extraction rules work on the parse tree to get an initial list of APs and TCs. The AP truth table is used to establish relationships between the initial list of APs. For example, an AP may be expressible as a conjunction of two other APs. The initial expert user pruned list of APs gives insight into the states of the transition system. We have found empirically that having one state for this initial pruned AP list is a good heuristic to compute the states of the transition system. Transitions are applied between every two states and then pruned by the expert user. TCs are paired with APs and this information is used to assign TCs to transitions.

Transforming natural language requirements into formal models is quite a hard problem and hard to get right without input from domain expert. Our approach sets up a very structured process, where the tool does lot of the work in analyzing and synthesizing TSs and TTSs, but also allows for input from domain expert. The proposed methodology has worked very well in practice for the GIIP requirements. All the TSs and TTSs computed for the requirements satisfied their corresponding CTL properties.

## ACKNOWLEDGMENT

This publication was funded by a grant from the United States Government and the generous support of the American people through the United States Department of State and the United States Agency for International Development (USAID) under the Pakistan - U.S. Science & Technology Cooperation Program. The contents do not necessarily reflect the views of the United States Government. The authors would like to acknowledge Dr. Vinay Gonela for helping with proofreading the paper.

## REFERENCES

[1] E. M. Al-qtiemat, S. K. Srinivasan, M. A. L. Dubasi, and S. Shuja, “A methodology for synthesizing formal specification models from requirements for refinement-based object code verification,” in The Third International Conference on Cyber-Technologies and Cyber-Systems. IARIA, 2018, pp. 94–101.

[2] FDA, “List of Device Recalls, U.S. Food and Drug Administration (FDA),” 2018, last accessed: 2018-09-10. [Online]. Available: https://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfRES/res.cfm

[3] R. Kaivola et al., “Replacing testing with formal verification in intel coretm i7 processor execution engine validation,” in Computer Aided Verification, 21st International Conference, CAV, Grenoble, France, June 26 - July 2, 2009. Proceedings, ser. Lecture Notes in Computer Science, A. Bouajjani and O. Maler, Eds., vol. 5643. Springer, pp. 414–429. [Online]. Available: https://doi.org/10.1007/978-3-642-02658-4_32

[4] T. Ball, B. Cook, V. Levin, and S. K. Rajamani, “SLAM and static driver verifier: Technology transfer of formal methods inside microsoft,” in Integrated Formal Methods, 4th International Conference, IFM, Canterbury, UK, April 4-7, 2004, Proceedings, ser. Lecture Notes in Computer Science, E. A. Boiten, J. Derrick, and G. Smith, Eds., vol. 2999. Springer, pp. 1–20. [Online]. Available: https://doi.org/10.1007/978-3-540-24756-2_1

[5] K. Bhargavan et al., “Formal verification of smart contracts: Short paper,” in Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS, Vienna, Austria, October 24, T. C. Murray and D. Stefan, Eds. ACM, pp. 91–96. [Online]. Available: http://doi.acm.org/10.1145/2993600.2993611

[6] D. Delmas et al., “Towards an industrial use of fluctuat on safety-critical avionics software,” in International Workshop on Formal Methods for Industrial Critical Systems. Springer, 2009, pp. 53–69.

[7] P. Manolios, “Mechanical verification of reactive systems,” PhD thesis, University of Texas at Austin, August 2001, last accessed: 2018-10-10. [Online]. Available: http://www.ccs.neu.edu/home/pete/research/phd-dissertation.html

[8] M. A. L. Dubasi, S. K. Srinivasan, and V. Wijayasekara, “Timed refinement for verification of real-time object code programs,” in Working Conference on Verified Software: Theories, Tools, and Experiments. Springer, 2014, pp. 252–269.

[9] Tsujii laboratory, Department of Computer Science at The University of Tokyo, “Enju - a fast, accurate, and deep parser for English,” 2011, available from http://www.nactem.ac.uk/enju, [accessed: 2018-07-10].

[10] V. Ágel, Dependency and valency: an international handbook of contemporary research. Walter de Gruyter, 2003, vol. 1.

[11] S Ghosh et al., “Automatic requirements specification extraction from natural language (ARSENAL),” SRI International, Menlo Park, CA, Tech. Rep., 2014.

[12] D. Aceituna, H. Do, and S. Srinivasan, “A systematic approach to

transforming system requirements into model checking specifications," in Companion Proceedings of the 36th International Conference on Software Engineering. ACM, 2014, pp. 165–174.

[13] I. G. Harris, "Extracting design information from natural language specifications," in Proceedings of the 49th Annual Design Automation Conference. ACM, 2012, pp. 1256–1257.

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

[15] R. L. Smith, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil, "Propel: an approach supporting property elucidation," in Proceedings of the 24th International Conference on Software Engineering. ACM, 2002, pp. 11–21.

[16] K. Shimizu, "Writing, verifying, and exploiting formal specifications for hardware designs," Ph.D. dissertation, PhD thesis, Stanford University, 2002.

[17] D. Zowghi, V. Gervasi, and A. McRae, "Using default reasoning to discover inconsistencies in natural language requirements," in Software Engineering Conference. APSEC 2001. Eighth Asia-Pacific. IEEE, pp. 133–140.

[18] V. Gervasi and D. Zowghi, "Reasoning about inconsistencies in natural language requirements," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 14, no. 3, 2005, pp. 277–330.

[19] W. Scott, S. Cook, and J. Kasser, "Development and application of a context-free grammar for requirements," in SETE 2004: Focussing on Project Success; Conference Proceedings; 8-10 November 2004. Systems Engineering Society of Australia, 2004, p. 333.

[20] X. Xiao, A. Paradkar, S. Thummalapenta, and T. Xie, "Automated extraction of security policies from natural-language software documents," in Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. ACM, 2012, p. 12.

[21] Z. Ding, M. Jiang, and J. Palsberg, "From textual use cases to service component models," in Proceedings of the 3rd International Workshop on Principles of Engineering Service-Oriented Systems. ACM, 2011, pp. 8–14.

[22] C. Rolland and C. Proix, "A natural language approach for requirements engineering," in International Conference on Advanced Information Systems Engineering. Springer, 1992, pp. 257–277.

[23] P. Bouyer, U. Fahrenberg, K. G. Larsen, N. Markey, J. Ouaknine, and J. Worrell, "Model checking real-time systems," in Handbook of Model Checking., E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds. Springer, 2018, pp. 1001–1046.

[24] D. Knorreck, L. Apvrille, and P. de Saqui-Sannes, "TEPE: a sysml language for time-constrained property modeling and formal verification," ACM SIGSOFT Software Engineering Notes, vol. 36, no. 1, 2011, pp. 1–8.

[25] A. Shrivastava, M. Mehrabian, M. Khayatian, P. Derler, H. A. Andrade, K. Stanton, Y. Li-Baboud, E. Griffor, M. Weiss, and J. C. Eidson, "A testbed to verify the timing behavior of cyber-physical systems: Invited," in Proceedings of the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA, June 18-22, 2017. ACM, 2017, pp. 69:1–69:6.

[26] J. Peters, N. Przigoda, R. Wille, and R. Drechsler, "Clocks vs. instants relations: Verifying CCSL time constraints in UML/MARTE models," in 2016 ACM/IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE, Kanpur, India, November 18-20. IEEE, pp. 78–84.

[27] E. Kang, L. Huang, and D. Mu, "Formal verification of energy and timed requirements for a cooperative automotive system," in Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, H. M. Haddad, R. L. Wainwright, and R. Chbeir, Eds. ACM, pp. 1492–1499.

[28] G. Carvalho, A. Cavalcanti, and A. Sampaio, "Modelling timed reactive systems from natural-language requirements," Formal Asp. Comput., vol. 28, no. 5, 2016, pp. 725–765.

[29] J. Hassine, "Early modeling and validation of timed system requirements using timed use case maps," Requir. Eng., vol. 20, no. 2, 2015, pp. 181–211.

[30] Y. Zhang, R. Jetley, P. L. Jones, and A. Ray, "Generic safety requirements for developing safe insulin pump software," Journal of diabetes science and technology, vol. 5, no. 6, 2011, pp. 1403–1419.

[31] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on uppaal," in Formal methods for the design of real-time systems. Springer, 2004, pp. 200–236.