SYNTHESIS OF SPECIFICATIONS AND REFINEMENT MAPS FOR REAL-TIME OBJECT CODE VERIFICATION

A Dissertation
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Eman Mohammad Al-Qtiemat

In Partial Fulfillment of the Requirements
for the Degree of
DOCTOR OF PHILOSOPHY

Major Department:
Electrical and Computer Engineering

April 2020

Fargo, North Dakota

NORTH DAKOTA STATE UNIVERSITY

Graduate School

Title

SYNTHESIS OF SPECIFICATIONS AND REFINEMENT MAPS FOR REAL-TIME OBJECT CODE VERIFICATION

By

Eman Mohammad Al-Qtiemat

The supervisory committee certifies that this dissertation complies with North Dakota State University's regulations and meets the accepted standards for the degree of

DOCTOR OF PHILOSOPHY

SUPERVISORY COMMITTEE:

Dr.Sudarshan Srinivasan
Chair

Dr.Danling Wang

Dr.Yechun Wang

Dr.Dharmakeerthi Nawarathna

Approved:

26, April, 2020
Date

Ben Braaten
Department Chair

ABSTRACT

Formal verification methods have been shown to be very effective in finding corner-case bugs and ensuring the 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 into formal specifications is currently a big gap. While there is some work in this area, we proposed 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. The next step in the verification process is to develop the refinement map, which is a mapping function that can relate an implementation state (in this context, the state of the object code program to be verified) with the specification state. Actually, constructing refinement maps often requires deep understanding and intuitions about the specification and implementation, it is shown very difficult to construct refinement maps manually. To go over this obstacle, the construction of refinement maps should be automated. As a first step toward the automation process, we manually developed refinement maps for various safety properties concerning the software control operation of infusion pumps. In addition, we identified possible generic templates for the construction of refinement maps. Recently, synthesizing procedures of refinement maps for functional and timing specifications are proposed. The proposed work develops a process that significantly increases the automation in the generation of these refinement maps. The refinement maps can then be used for refinement-based verification. This automation procedure has been successfully applied on the transformed safety requirements in the first part of our work. This approach is based on the identified generic refinement map templates which can be increased in the future as the application required.

ACKNOWLEDGEMENTS

All praises and thanks to Allah almighty, my Creator, my Sustainer, for giving me courage and strength to pursue my PhD and fulfill the requirements of this disquisition.

My heartiest and sincere appreciation and gratitude to my mentor and adviser Dr. Sudarshan Srinivasan, who always encouraged me, and persistently conveyed the spirit and guidance required for the research. Without his kind guidance and continuous efforts, this disquisition would not have been possible.

Special thanks to my committee members, Dr. Danling Wang, Dr. Yechun Wang, Dr. Scott Smith and Dr. Dharmakeerthi Nawarathna for their support, guidance and helpful recommendations. Thanks to the Electrical and Computer Engineering staff members for all the unconditional help and favor. I owe my heartiest thanks to all my friends and colleagues here in the US and Jordan, who always helped me in the time of need. I am grateful to the United States Agency for International Development (USAID) Government for the financial support that I have received as a graduate student. Getting funding for the most part of my Ph.D. journey has helped reduced the constant worry of supporting myself financially.

I would like to thank my beloved family, each one of you has contributed to helping me become the person I am today. A special thanks to my husband 'Zeyad' who has been through the various ups and downs of our life together during my Ph.D. journey. He has indeed taught me a different aspect of seeing life. He has been through all my tantrums and still supported me in achieving this goal. He supported me in many ways. Thank you for coming into my life. Also, I would like to thank my sons Adam and Karam, for their patience, time, and support as you both are the source of happiness for me. Today, as I fulfill my dad's and mum's dream who have supported me emotionally throughout this challenging time. Your motivational words have helped me to survive this big journey of Ph.D. A Special thanks to you as you are the only and every reason for whatever I am today and whatever I achieved in my life.

My brothers, sisters, and your beloved kids, your continuous support is always a source of motivation and encouragement for me. Thank you for you all and your families as well. I would like to thank my Friends who have supported me and believed in my abilities to do such work, i am

lucky to have you in my life. Thank you each and everyone for your encouraging words whenever I needed them the most. Without all of your support, I would not have come this far and achieved all this.

“Do [as you will], for Allah will see your deeds, and [so, will] His Messenger and the believers. And you will be returned to the Knower of the unseen and the witnessed, and He will inform you of what you used to do.” Holly Quran, surah At Taubah, chapter 9, verse 105.

”And the last of our call will be, Praise to Allah, Lord of the worlds!”

DEDICATION

I would like to dedicate this dissertation to my family, especially to my parents, my husband, and my sons (Adam and Karam) for all the love, support, and motivation.

TABLE OF CONTENTS

ABSTRACT . . . iii

ACKNOWLEDGEMENTS . . . iv

DEDICATION . . . vi

LIST OF TABLES . . . x

LIST OF FIGURES . . . xi

LIST OF APPENDIX TABLES . . . xiii

LIST OF APPENDIX FIGURES . . . xiv

1 INTRODUCTION . . . 1

1.1 Overview . . . 1

1.2 Motivation . . . 1

1.3 Problem Statement . . . 2

1.3.1 Testing . . . 2

1.3.2 Formal Verification . . . 3

1.4 Thesis Outline . . . 6

1.5 References . . . 7

2 SYNTHESIS OF FORMAL SPECIFICATIONS FROM REQUIREMENTS FOR REFINEMENT BASED REAL TIME OBJECT CODE VERIFICATION . . . 10

2.1 Introduction . . . 10

2.2 Background . . . 12

2.2.1 Parsing Tree . . . 12

2.2.2 Transition Systems . . . 13

2.2.3 Timed Transition Systems . . . 14

2.3 Related Work . . . 14

2.4 Formal Model Synthesis Procedure for Functional Requirements . . . 17

2.4.1 Atomic Proposition Extraction Rule 1 (APER 1) . . . 17

2.4.2 Atomic Proposition Extraction Rule 2 (APER 2) 19

2.4.3 Atomic Proposition Extraction Rule 3 (APER 3) 22

2.4.4 High-Level Procedure for Specification Transition System Synthesis 24

2.5 Formal Model Synthesis Procedure for Timing Requirements 26

2.5.1 Atomic Proposition and Timing Constrains Extraction Rule (APTCER) for Timed Transition System 26

2.5.2 High-Level Procedure for Specification Timed Transition System Synthesis . 28

2.6 Case Study: Generic Insulin Infusion Pump (GIIP) 31

2.6.1 Functional Requirements of GIIP 31

2.6.2 Timing Requirements of GIIP 32

2.7 Results Analysis 37

2.8 Conclusion 40

2.9 References 41

3 SYNTHESIS OF REFINEMENT MAPS FOR REAL-TIME OBJECT CODE VERIFICATION 45

3.1 Introduction 45

3.2 Background 46

3.2.1 Transition Systems 46

3.2.2 Timing Transition Systems 47

3.2.3 Refinement-Based Verification 48

3.2.4 Synthesis of Functional Formal Specifications 49

3.2.5 Synthesis of Timing Formal Specifications 49

3.3 Related Work 52

3.4 Refinement Maps and Refinement Map Templates 55

3.5 Synthesis of Refinement Maps for System Requirements 62

3.6 Conclusion and Future Work 67

3.7 References 67

4 CONCLUSIONS AND FUTURE WORK 72
4.1 Conclusions 72
4.2 Directions for Future Research 73
APPENDIX A. INSULIN PUMP SAFETY REQUIREMENTS 75
APPENDIX B. PARSED TREES OF INSULIN PUMP SAFETY REQUIREMENTS 78
APPENDIX C. LIST OF PUBLICATIONS 97

LIST OF TABLES

Table | Page

2.1 AP-Truth table for requirements 1.8.2 and 1.8.5 from AP-list 32

2.2 Resulting transition systems by applying procedure 4 and APERS on a set of system requirements 35

2.3 AP-truth-table for timing requirement 1.6.1 from AP-list 36

2.4 Resulting timed transition systems by applying procedure 6 and APTCER on a set of timing requirements 39

3.1 List of abbreviations for Figures 3.6-3.15 65

LIST OF FIGURES

Figure | Page

1.1 Integrating safety requirements into software development life cycle. 3

1.2 Requirement 1.1.1 in both natural language and formal forms. 5

2.1 A simple example of a parsing tree using Enju parser 13

2.2 An example of a transition system (TS). 13

2.3 An example of a timed transition system (TTS) 14

2.4 An Enju parsing tree portion shows some resulting APs by applying APER 1. 18

2.5 An Enju parsing tree portion of requirement 2.2.2 shows some resulting APs by applying APER 1. 19

2.6 An Enju parsing tree portion of requirement 1.8.2 shows some resulting APs by applying APER 1. 20

2.7 An Enju parsing tree portion of requirement 1.2.6 shows some resulting APs by applying APER 2. 21

2.8 An Enju parsing tree portion shows some resulting APs by applying APER 2. 22

2.9 An Enju parsing tree portion shows some resulting APs by applying APER 2 on requirement 1.1.3. 23

2.10 An Enju parsing tree portion shows some resulting APs using APER 3. 25

2.11 An Enju parsing tree portion shows some resulting APs by applying APER 3 on requirement 3.2.5. 26

2.12 An Enju parsing tree shows three resulting TBSs after applying APTCER. 33

2.13 An Enju parsing tree portion shows the resulting TBS $\langle AP - list, TC - list \rangle)$ after applying APTCER. 34

2.14 Finite state machine for suspension mode requirements (1.8.1 and 1.8.5). 37

2.15 Timed finite state machine for air-in-line requirement (1.6.1). 38

3.1 An example of a transition system (TS). 47

3.2 An example of a timing transition system (TTS). 48

3.3 Flowchart of formal model synthesis procedure for Functional Requirements. 50

3.4 Flowchart of formal model synthesis procedure for Timing Requirements. 51

3.5 A formal presentation of requirement 1.1.1 and the suggested refinement maps. 55

3.6 A formal presentation of requirement 1.1.3 and the suggested refinement maps. 56

3.7 A formal presentation of requirement 1.8.2 and 1.8.5 and the suggested refinement maps. 56

3.8 A formal presentation of requirement 1.3.5 and the suggested refinement maps. 57

3.9 A formal presentation of requirement 2.2.2 and 2.2.3 and the suggested refinement maps. 57

3.10 A formal presentation of requirement 3.2.5 followed by the suggested refinement maps. . 58

3.11 A formal presentation of requirement 3.2.7 followed by the suggested refinement maps. . 58

3.12 A formal presentation of the timing requirement 1.2.8 and the suggested refinement maps. 59

3.13 A formal presentation of the timing requirement 1.6.1 followed by the suggested refinement maps. 59

3.14 A formal presentation of the timing requirement 1.8.4 followed by the suggested refinement maps. 60

3.15 A formal presentation of the timing requirement 2.2.1 followed by the suggested refinement maps. 60

LIST OF APPENDIX TABLES

Table | Page

B.1 The main syntactic categories used by Enju trees 78

LIST OF APPENDIX FIGURES

Figure | Page

B.1 Enju parsed tree for requirement 1.1.1. 79
B.2 Enju parsed tree for requirement 1.1.3. 80
B.3 Enju parsed tree for requirement 1.2.4. 81
B.4 Enju parsed tree for requirement 1.2.6. 82
B.5 Enju parsed tree for requirement 1.2.7. 83
B.6 Enju parsed tree for requirement 1.3.5. 84
B.7 Enju parsed tree for requirement 3.1.1 (part one). 85
B.8 Enju parsed tree for requirement 3.1.1 (part two). 86
B.9 Enju parsed tree for requirement 1.8.2. 87
B.10 Enju parsed tree for requirement 1.8.5. 88
B.11 Enju parsed tree for requirement 2.2.2. 89
B.12 Enju parsed tree for requirement 2.2.3. 90
B.13 Enju parsed tree for requirement 3.2.5. 91
B.14 Enju parsed tree for requirement 3.2.7. 92
B.15 Enju parsed tree for requirement 1.2.8. 93
B.16 Enju parsed tree for requirement 1.6.1. 94
B.17 Enju parsed tree for requirement 1.8.4. 95
B.18 Enju parsed tree for requirement 2.2.1. 96

1. INTRODUCTION

1.1. Overview

Safety critical device is a device that its failure can cause a big problem such as loose of money in commercial fields, human health issues or even a death. For example, a medical device that affects patient's life is called a safety critical device. A medical device is an instrument or machine that is used to address, monitor, or control a health situation. It is used for delivering controlled doses of fluid medications to patients such as infusion pump, controlling some important functionalities like pacemaker that helps in controlling patient's heartbeat, or even as simple as monitoring the vital signs such as a blood pressure monitor. Medical devices like infusion pumps, pacemaker, etc. must be safe which means that failure is unacceptable. Software development cycles usually have testing processes to explore expected issues. however, it is not considered to be a comprehensive technique, it can catch errors but can not prove their absence. A proof of system correctness becomes essential, this proof can be accomplished by the use of verification processes such as formal verification, our work presents formal verification processes addressing safety issues in the safety critical devices.

1.2. Motivation

Ensuring the correctness of control software used in safety-critical 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 Internet of Things (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 functionalities to fix existing hazards, or modifying some defined functionalities that cause faulty behaviours. Since

critical devices are considered as real time systems, most of their functionalities have well defined timing conditions must be met else wise the system will fail. System functionalities keeping the system safe are called safety requirements, they are usually written in natural language,therefor, embedding them into the design is a difficult job. requirements need to be transformed into formal presentation (specifications). A procedure to address the gap between informal and formal requirements is required, this is the first goal of this thesis, we propose a structured method to transform requirements in natural language into formal presentation. Embedding the safety requirements in a formal form is not enough to assure the system correctness, critical systems should be checked before use especially medical devices. Testing is the commonly used process to check the software, but it is not sufficient because it does not check all system's functionalities comprehensively, it can only help in finding some errors but not all of them, it should be complemented or even replaced by other techniques such as formal methods [3]. Formal verification can provide proofs of correctness of the system such as model checking, static analysis, and program proof. In our work, we utilize model checking in verifying the transformation procedure of a case study requirements to make sure that formal presentation meets safety requirements. In addition, we study refinement based verification which is a scalable verification technique, actually it is not commonly used because: 1) It requires formal specifications in transition system forms. 2) It requires to build functions called refinement maps which is a complex and difficult mission. We present solutions to address both obstacles by proposing algorithms for each job. Our proposed algorithms are successfully applied on safety requirements for infusion pumps proposed by group of researcher with the help of the FDA [4] [5] as a case study. The existence of safety requirements for infusion pump is considered a motivation of us, the new proposed safety requirements are based on a comprehensive knowledge of the common existing hazards. Figure 1.1 illustrates the process of integrating safety requirements into software development life-cycle, this model needs to be implemented, here our engineering model takes place.

1.3. Problem Statement

1.3.1. Testing

Testing is a process evaluates if the software meets the desired requirements. Despite the fact that testing is the dominant verification technique currently used in commercial design cycles [6], testing can only show the presence of faults, but it never proves their absence [7]. Although it

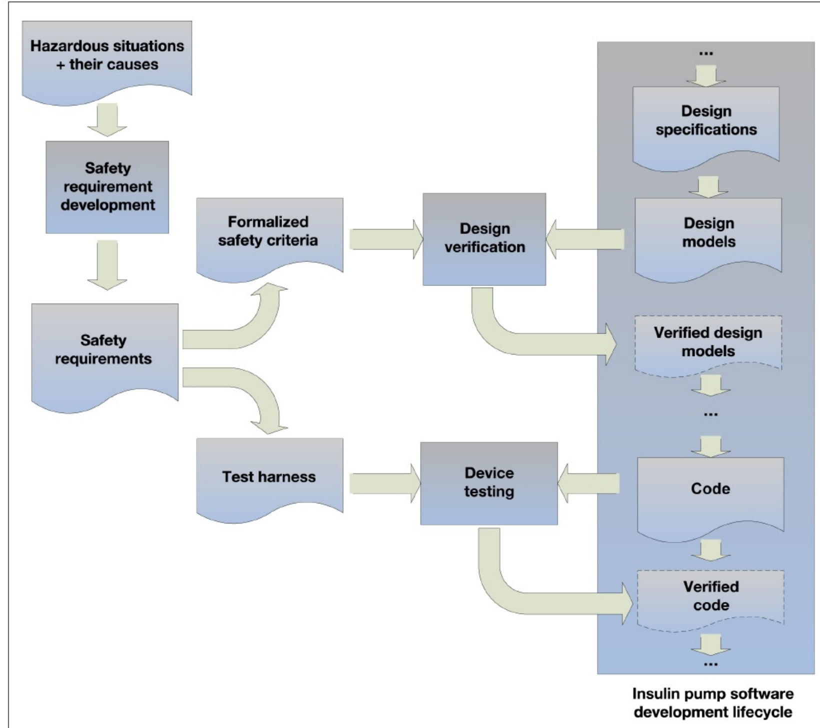


Figure 1.1. Integrating safety requirements into software development life cycle [4].

checks the system functionalities, this process is time consuming because each function should be tested individually. Large systems might have millions of functions, this makes testing insufficient. Alternate verification processes should be applied to the software design in conjunction with testing to assure system correctness and reliability. Formal verification can address testing limitations by providing proofs of correctness for software safety. Intel [8], Microsoft [9] and [10], and Airbus [11] have successfully applied formal verification processes.

1.3.2. Formal Verification

Formal verification is a proofing process of the correctness of system properties, it shows if the system is acting as specified by its requirements or not [12]. it is considered as the only way guarantees that the system is free from errors [13]. This verification method uses the system mathematical model to assure its correctability [12]. In this thesis, we study two verification process; model checking and refinement based verification as explained below.

1.3.2.1. Model Checking

To reduce the reliance on testing, model checking technique is used to check the correctness of software designs. This verification system can be applied early on a design cycle, or even late on

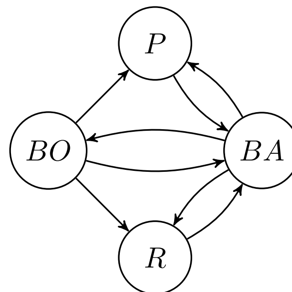
the final design [14]. A model checker is a tool that can check if a model satisfies a set of properties. The properties have to be expressed in a temporal logic. NuSMV is a well structured, flexible, robust, open and documented tool for model checking [15]. NUSMV allows for the representation of synchronous and asynchronous finite state systems, in additional to analyzing of specifications expressed in Computation Tree Logic (CTL) and Linear Temporal Logic (LTL) [16]. In our work, we perform evaluations of transition systems using the NuSMV model checker. Another model checker called UUPPAL is specifically designed for verifying real-time systems. It is based on the timed automata theory [17]. Our work uses UUPPAL to verify resulting transition systems with timing constraints.

1.3.2.2. Refinement Based Verification

Refinement-based verification [18] is a formal verification technology that has been demonstrated to be applicable to the verification of embedded control software at the object-code level [19]. 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. Our verification process is based on the theory of Well-Founded Equivalence Bisimulation refinement. A detailed description of this theory can be found in [18]. Here, we give a very high-level overview of the key concepts. As stated earlier, WEB refinement provides a notion of correctness that can be used to check an implementation TS against a specification TS. One of the key features is that WEB refinement accounts for stuttering, which is the phenomenon where multiple but finite transitions of the implementation can match a single transition of the specification. This is a very key feature because the control code implements many functions and only some of these functions may be relevant to the safety property being verified. Therefore, the code may be doing a number of things that do not relate to the property and will therefore be stuttering a lot w.r.t. the specification. Another key feature of WEB refinement is refinement maps, which is the focus of this work. Refinement maps are functions that map implementation states to specification states. There is a lot of flexibility in how

Requirement 1.1.1: *The pump shall suspend all active basal delivery and stop any active bolus during a pump prime or refill. It shall prohibit any insulin administration during the priming process and resume the suspended basal delivery, either a basal profile or a temporary basal, after the prime or refill is successfully completed. [4]*

(a) Safety requirement 1.1.1 in natural language.



(b) Formal specification representation of requirement 1.1.1.

Figure 1.2. Requirement 1.1.1 in both natural language and formal forms.

refinement maps can be defined. This allows for low-level implementations to be verified against high-level specifications.

Definition 1 *(WEB Refinement): Let* $M = \langle S, R, L\rangle$*,* $M' = \langle S', R', L'\rangle$*, and r:* $S \to S'$*. M is a WEB refinement of* M' *with respect to refinement map r, written* $M \approx r\ M'$*, if there exists a relation, B, such that* $\langle \forall\ s \in S :: sB(r.s)\rangle$ *and B is a WEB on the TS* $\langle\ S \uplus S',\ R \uplus R',\ L\ \rangle$*, where* $L.s = L'(s)$ *for s and S' state and* $L.s = L'(r.s)$ *otherwise.*

Sections 1.3.2.3 and 1.3.2.4 explains the main obstacles of applying refinement based verification.

1.3.2.3. Formal Specifications

Formal specification is a mathematical presentation of system requirements. 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. This work presents a novel methodology to transform natural language requirements into formal specifications to be used in the context of refinement based verification. The new methodology is applied on insulin pump safety requirements which were proposed to solve the common hazards of insulin pump models in the market. Figure 1.2 shows an example of an insulin pump safety requirement (Requirement 1.1.1). Figure 1.2a shows the requirement in natural language, while Figure 1.2b shows the resulting formal specification after applying our methodology.

1.3.2.4. Refinement Maps

One of the key features of refinement-based verification 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. Historically, one of the reasons that refinement-based verification is much less explored than other formal verification paradigms such as model checking is that the construction of refinement maps often requires deep understanding and intuitions about the specification and implementation [20]. However, once a refinement map is constructed, the benefit is that refinement-based verification is a very scalable approach for dealing with low-level artifacts such as real-time object code verification. We build refinement maps corresponding to formal specifications related to infusion pump safety and we also propose three possible generic refinement map templates, which is the first step toward automating the construction of refinement maps. Recently, we presents a synthesising procedure of refinement maps, This is an automation procedure to choose a refinement map template based on heuristic data. This methodology is applied on safety requirements of insulin pump based on heuristics that are developed based on the output of the Enju parser to select a refinement map template for each atomic proposition.

1.4. Thesis Outline

The remainder of the thesis consists of the following chapters:

- Chapter 2 presents a novel approach that can synthesize natural language requirements to formal specifications that are useful for refinement-based verification. The proposed approach addresses both functional and timing requirements and has been demonstrated in the context of safety requirements for software control of infusion pumps. The ability to model and validate the system properties for critical systems at the requirements level supports the

detection of design errors during the early stages of a software development life cycle and helps reduce the cost of later redesign activities.

- In Chapter 3, we develop refinement maps for various safety properties concerning the software control operation of insulin pumps. We also identify possible generic templates for construction of refinement maps as a first step towards developing a process to construct refinement maps in an automated fashion. Finally, we develop a synthesising procedure of refinement maps. Heuristics are developed based on the output of the Enju parser to select a refinement map template for each atomic proposition.

- In Chapter 4, we summarize the problems and how our work presents solutions for each issue. In addition, some suggested future directions are presented.

- Chapter 5 states the safety requirements we work on, they are written as in the main source.

- Chapter 6 is an appendix of parsed trees for all used safety requirements using an English parser called Enju.

Note that our work studied the correctness of critical systems, transforming natural language requirements into formal specifications will help the designers to include safety requirements easily to assure that the studied system is working correctly as specified. however, our work does not address the reliability issues but it can help indirectly in adding safety requirements that are proposed specially to solve reliability bugs.

1.5. References

[1] Eman M Al-qtiemat, Sudarshan K Srinivasan, Mohana Asha Latha Dubasi, and Sana 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)*. last accessed: 2018-09-10. 2018. URL: https://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfRES/res.cfm.

[3] Xavier Leroy. "Formal verification of a realistic compiler". In: *Communications of the ACM* 52.7 (2009), pp. 107–115.

[4] Yi Zhang, Raoul Jetley, Paul L Jones, and Arnab Ray. "Generic safety requirements for developing safe insulin pump software". In: *Journal of diabetes science and technology* 5.6 (2011), pp. 1403–1419.

[5] Yi Zhang, Paul L Jones, and Raoul Jetley. "A hazard analysis for a generic insulin infusion pump". In: *Journal of diabetes science and technology* 4.2 (2010), pp. 263–283.

[6] SMK Quadri and Sheikh Umar Farooq. "Software Testing-Goals, Principles, and Limitations". In: *International Journal of Computer Applications* 6.9 (2010), pp. 7–10.

[7] Edward Miller and William E Howden. *Tutorial, software testing & validation techniques.* IEEE Computer Society Press, 1981.

[8] 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.* Ed. by Ahmed Bouajjani and Oded Maler. Vol. 5643. Lecture Notes in Computer Science. Springer, 2009, pp. 414–429. ISBN: 978-3-642-02657-7. DOI: `10.1007/978-3-642-02658-4_32`. URL: `https://doi.org/10.1007/978-3-642-02658-4%5C_32`.

[9] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram 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.* Ed. by Eerke A. Boiten, John Derrick, and Graeme Smith. Vol. 2999. Lecture Notes in Computer Science. Springer, 2004, pp. 1–20. ISBN: 3-540-21377-5. DOI: `10.1007/978-3-540-24756-2_1`. URL: `https://doi.org/10.1007/978-3-540-24756-2%5C_1`.

[10] Karthikeyan Bhargavan et al. "Formal verification of smart contracts: Short paper". In: *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security.* ACM. 2016, pp. 91–96.

[11] 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.

[12] Robert M Keller. “Formal verification of parallel programs”. In: *Communications of the ACM* 19.7 (1976), pp. 371–384.

[13] Gerwin Klein et al. “seL4: Formal verification of an OS kernel”. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 2009, pp. 207–220.

[14] Gerard J Holzmann. *The SPIN model checker: Primer and reference manual*. Vol. 1003. Addison-Wesley Reading, 2004.

[15] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. “NuSMV: a new symbolic model checker”. In: *International Journal on Software Tools for Technology Transfer* 2.4 (2000), pp. 410–425.

[16] Roberto Cavada et al. “Nusmv 2.4 user manual”. In: *CMU and ITC-irst* (2005).

[17] Gerd Behrmann, Alexandre David, and Kim G Larsen. “A tutorial on uppaal”. In: *Formal methods for the design of real-time systems*. Springer. 2004, pp. 200–236.

[18] Panagiotis Manolios. “Mechanical Verification of Reactive Systems”. last accessed: 2018-10-10. PhD thesis. University of Texas at Austin, Aug. 2001. URL: `http://www.ccs.neu.edu/home/pete/research/phd-dissertation.html`.

[19] Mohana Asha Latha Dubasi, Sudarshan K Srinivasan, and Vidura 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.

[20] Martıén Abadi and Leslie Lamport. “The existence of refinement mappings”. In: *Theoretical Computer Science* 82.2 (1991), pp. 253–284.

2. SYNTHESIS OF FORMAL SPECIFICATIONS FROM REQUIREMENTS FOR REFINEMENT BASED REAL TIME OBJECT CODE VERIFICATION

2.1. 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.

Safety critical devices such as insulin pump still have safety 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 safety 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 chapter 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

The content of this chapter has been published in the International Journal on Advances in Internet Technology 2019. The material in this chapter was co-authored by Eman M. Al-Qtiemat, Sudarshan Srinivasan, Zeyad Al-Odat, Mohana Asha Latha Dubas,and Sana Shuja. Eman M. Al-Qtiemat had primary responsibility for conducting experiments and collecting results. Eman M. Al-Qtiemat was the primary developer of the conclusions that are advanced here. Eman M. Al-Qtiemat also drafted and revised all versions of this chapter. Sudarshan Srinivasan and Mohana Asha Latha Dubas drafted and revised all versions of this chapter. Sudarshan Srinivasan served as proofreader.

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 Internet of Things (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 chapter, 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 chapter is organized as follows. An overview of the background is presented in Section 2.2. Section 2.3 details the related work. A formal model describing the synthesis procedure of functional requirements is presented in Section 2.4, while Section 2.5 presents a different formal

model describing the synthesis procedure of timing requirements. Section 2.6 details the case study. Section 2.7 gives the verification results for the proposed formal model. Conclusions and direction for future work are noted in Section 2.8.

2.2. 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.

2.2.1. 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 2.2.2). 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 2.1 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 2.1 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.

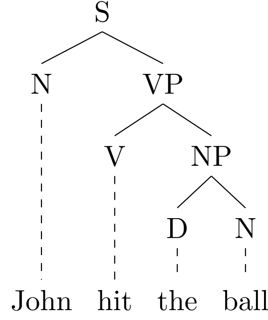


Figure 2.1. A simple example of a parsing tree using Enju parser[10].

2.2.2. 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 2 *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.*

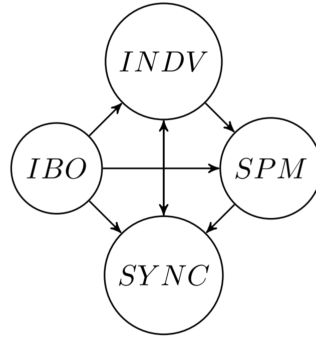


Figure 2.2. An example of a transition system (TS).

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 2.2. 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.

2.2.3. 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 2.2.2), 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.

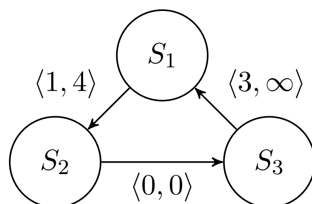


Figure 2.3. An example of a timed transition system (TTS)

Definition 3 *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 2.3 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.

2.3. 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 requirement's 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 work.

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.

2.4. 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.

2.4.1. 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 $\leftarrow \emptyset$;
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 $\leftarrow$ AP-list $\cup$ TerminalNodes(X) ;

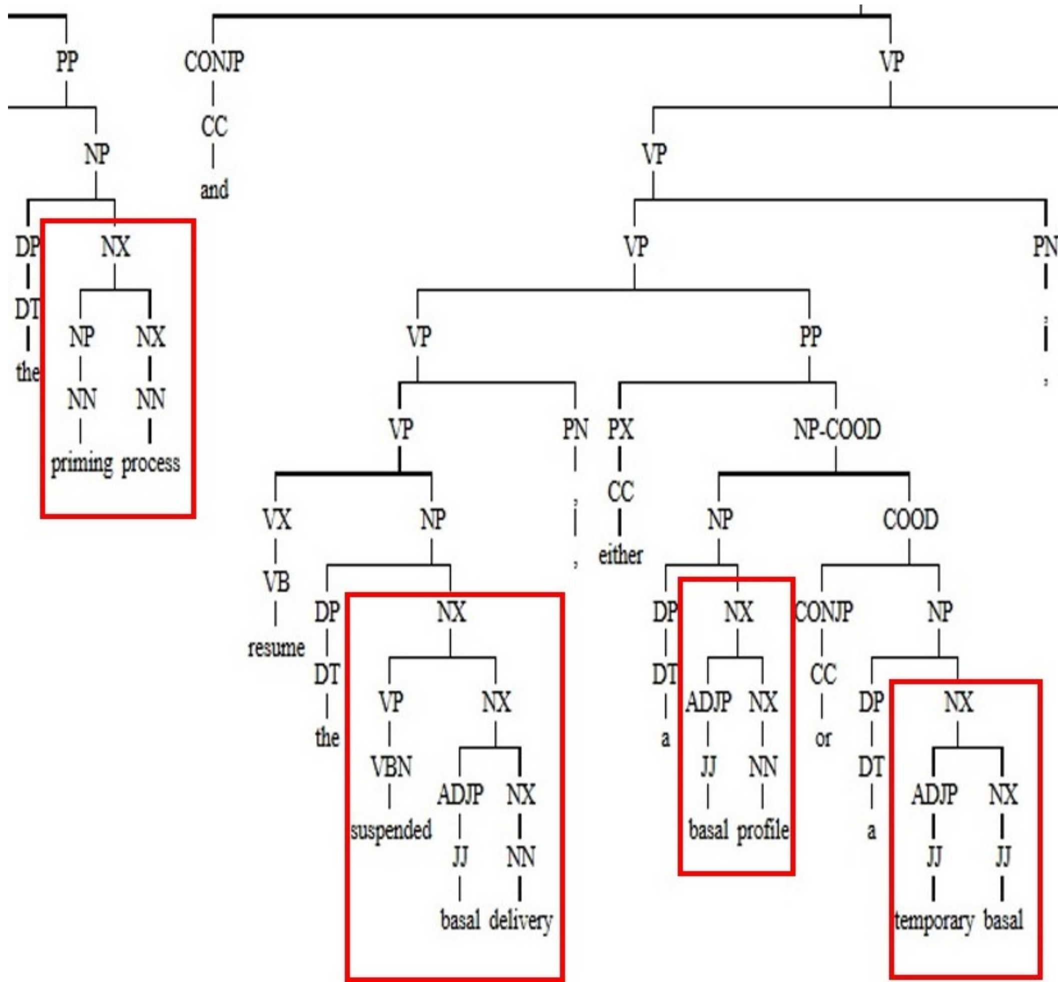


Figure 2.4. An Enju parsing tree portion shows some resulting APs by applying APER 1.

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 2.4 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.

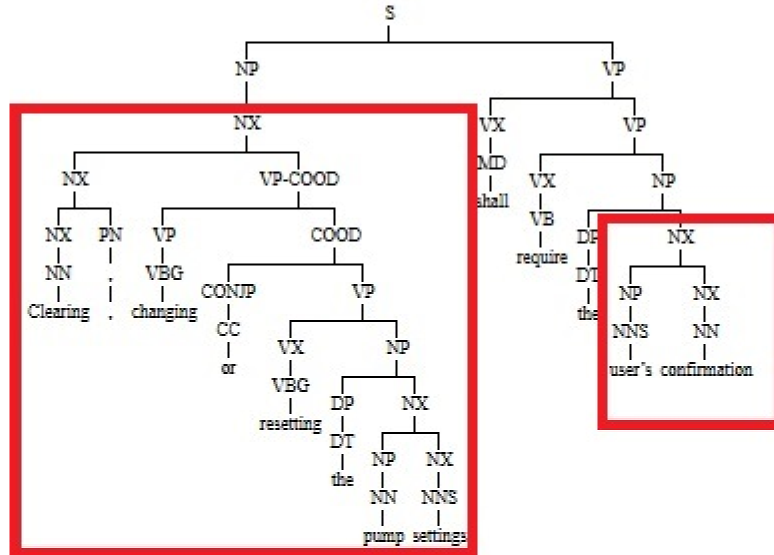


Figure 2.5. An Enju parsing tree portion of requirement 2.2.2 shows some resulting APs by applying APER 1.

As shown in Figure 2.4, 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 2.4 gives us the following APs: 'priming process', 'suspended basal profile', 'basal profile', and 'temporary basal'. Also, Figure 2.6 and Figure 2.5 show some resulting APs by applying APER 1 on requirement 1.8.2 and 2.2.2 respectively from [30].

2.4.2. 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.

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."

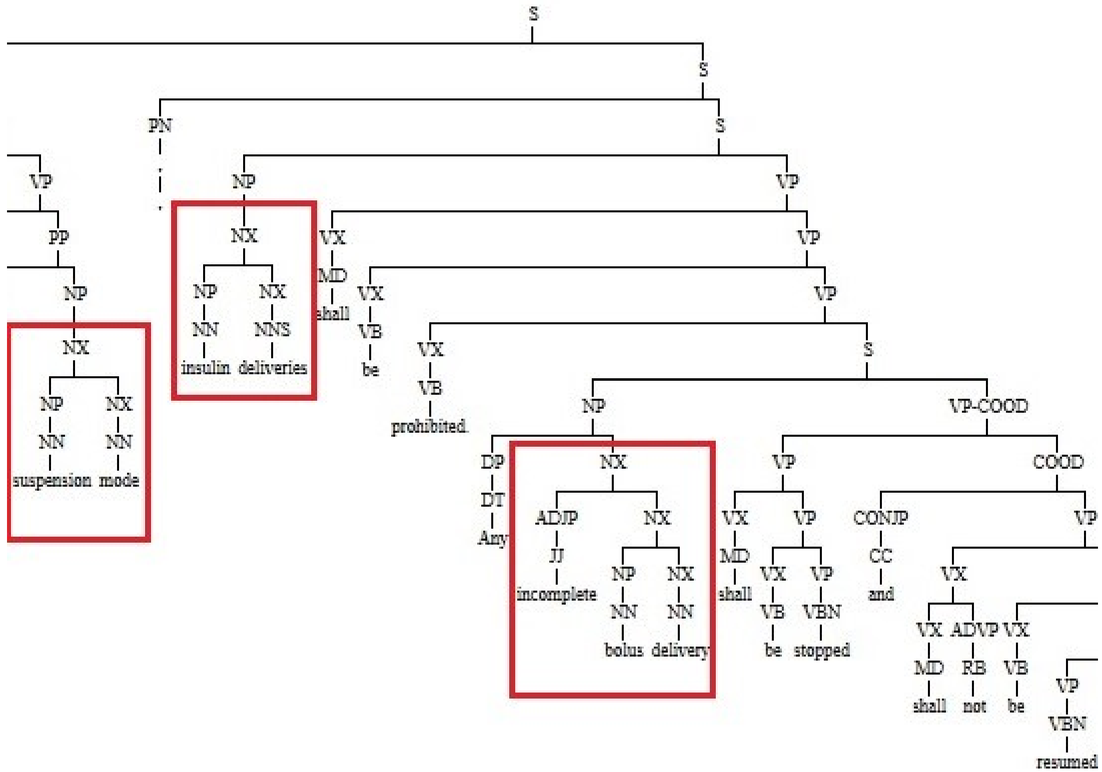


Figure 2.6. An Enju parsing tree portion of requirement 1.8.2 shows some resulting APs by applying APER 1.

Procedure 2 APER 2

Require: Parse-tree
1: AP-list $\leftarrow \emptyset$;
2: **for each** $n \in$ TerminalNodes(Parse-tree) **do**
3: Start-cat = head(head(n));
4: $X_1 \leftarrow \emptyset$;
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: 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 $\leftarrow$ AP-list $\cup$ TerminalNodes(X);

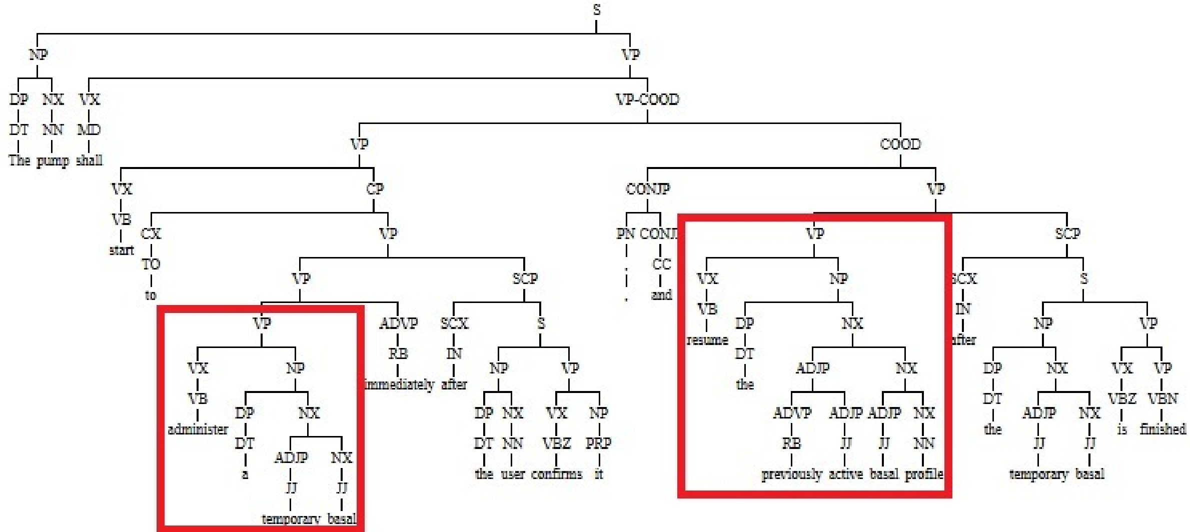


Figure 2.7. An Enju parsing tree portion of requirement 1.2.6 shows some resulting APs by applying APER 2.

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.

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 2.8 and Figure 2.7 show resulting sub tree examples by applying APER 2.

Figure 2.8 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

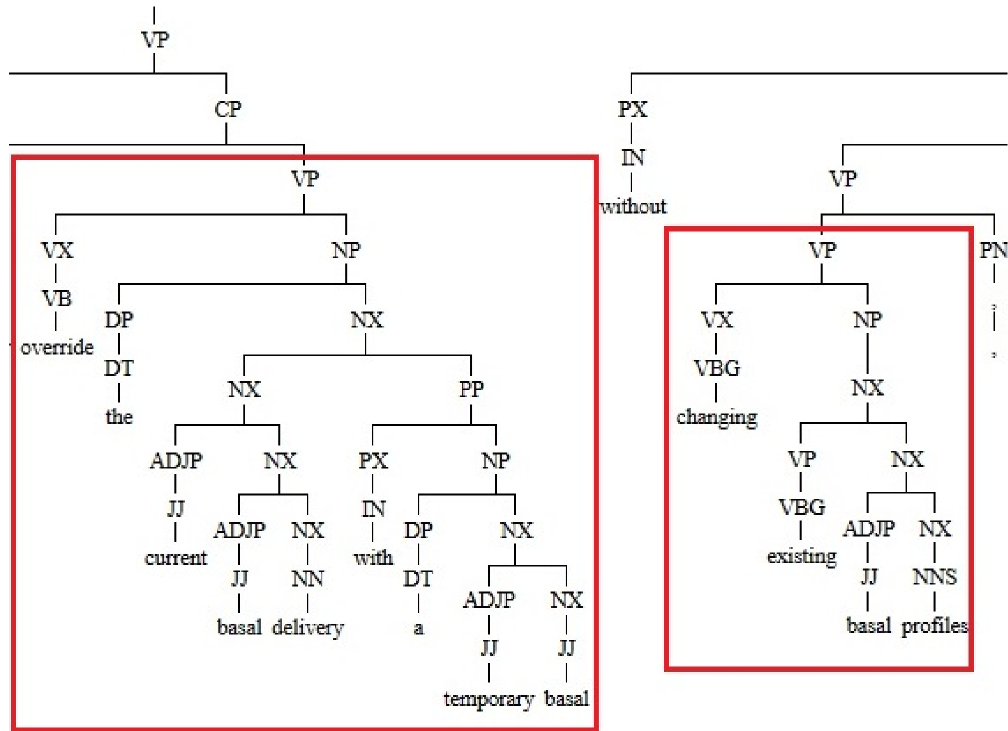


Figure 2.8. An Enju parsing tree portion shows some resulting APs by applying APER 2.

temporary basal', and 'changing existing basal profiles'. Also, Figure 2.9 shows some resulting APs by applying APER 2 on requirement 1.1.3 from [30].

2.4.3. 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

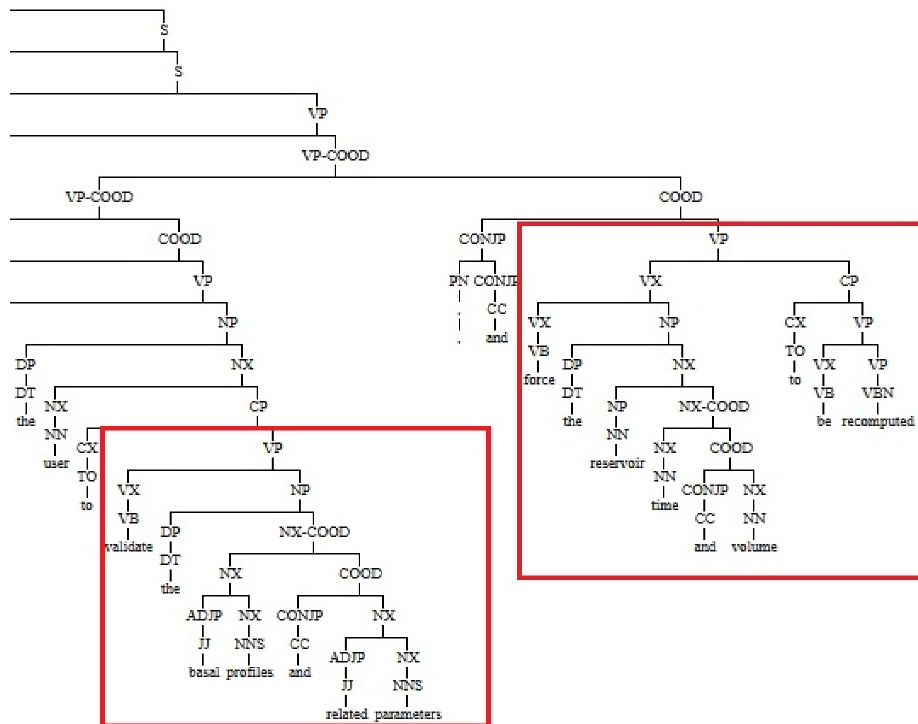


Figure 2.9. An Enju parsing tree portion shows some resulting APs by applying APER 2 on requirement 1.1.3.

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.

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 2.10, 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 2.10 gives the following APs: ’pump’, ’an alert’, and ’deny the request’. Also, Figure 2.11 shows some resulting APs by applying APER 3 on requirement 3.2.5 from [30].

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.

Procedure 3 APER 3

Require: Parse-tree

1: AP-list $\leftarrow \emptyset$;
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 $\leftarrow$ AP-list $\cup$ TerminalNodes(X);

2.4.4. High-Level Procedure for Specification Transition System Synthesis

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

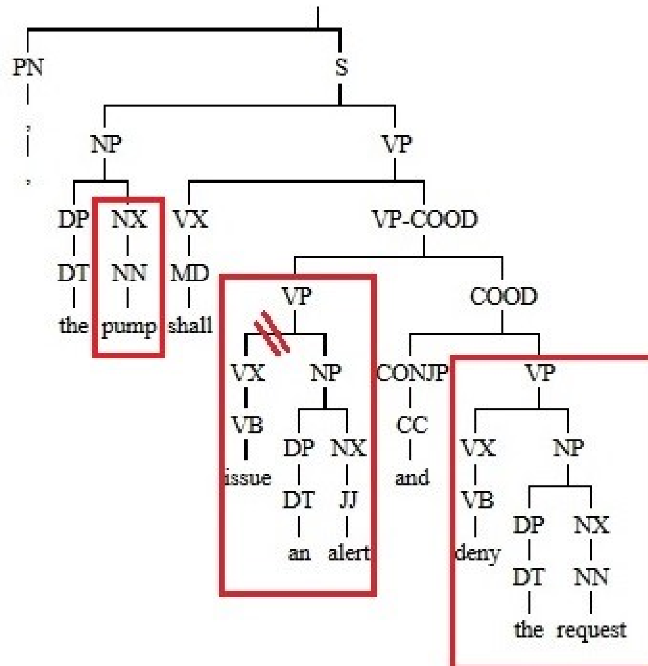


Figure 2.10. An Enju parsing tree portion shows some resulting APs using APER 3.

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). o07 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).

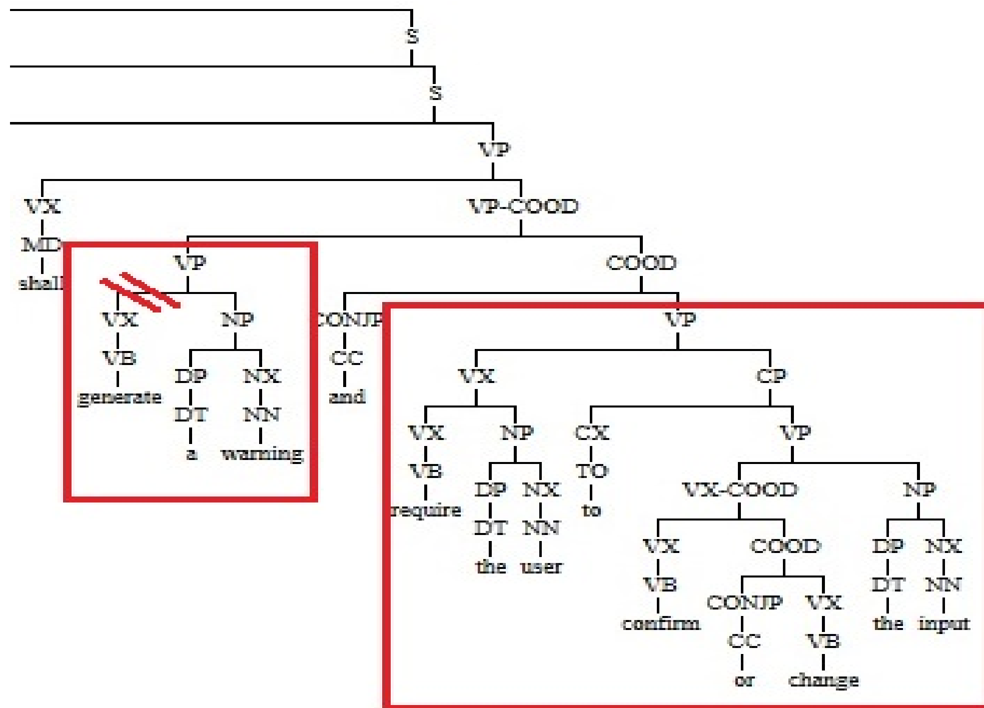


Figure 2.11. An Enju parsing tree portion shows some resulting APs by applying APER 3 on requirement 3.2.5.

2.5. 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.

2.5.1. 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

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;

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 2.12 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 2.12 give the following. AP-list is: 'air-in-line alarm', 'air bubbles larger than y μL', 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 2.13 shows a TBS that has two APs (in red boxes) and one TC (in a green box).

2.5.2. 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 user's 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 user's domain knowledge (line 10).

Procedure 5 APTCER

Require: Parse-tree

1: TBS-list $\leftarrow \emptyset$;
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 $\leftarrow$ 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) $\rightarrow$ 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) $\rightarrow$ AP-list;
26:     $TC - list_k \leftarrow$ AP-list ;
27:     $TBS_k = \langle AP - list_k, TC - list_k \rangle$;
28:     TBS-list $\leftarrow$ 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 (⟨AP − list, TC − list⟩);
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 ⟨0, ∞⟩;
20:     TTS ← CreateTS(T, S-list);
21:     TTS-set ← TTS-set U TTS;
22:     return TTS-set;
```

2.6. 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.

2.6.1. 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

Table 2.1. AP-Truth table for requirements 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

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 2.1.

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 2.14a. The expert user will approve or remove some unacceptable transitions. Figure 2.14b shows the final transition system.

2.6.2. 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.

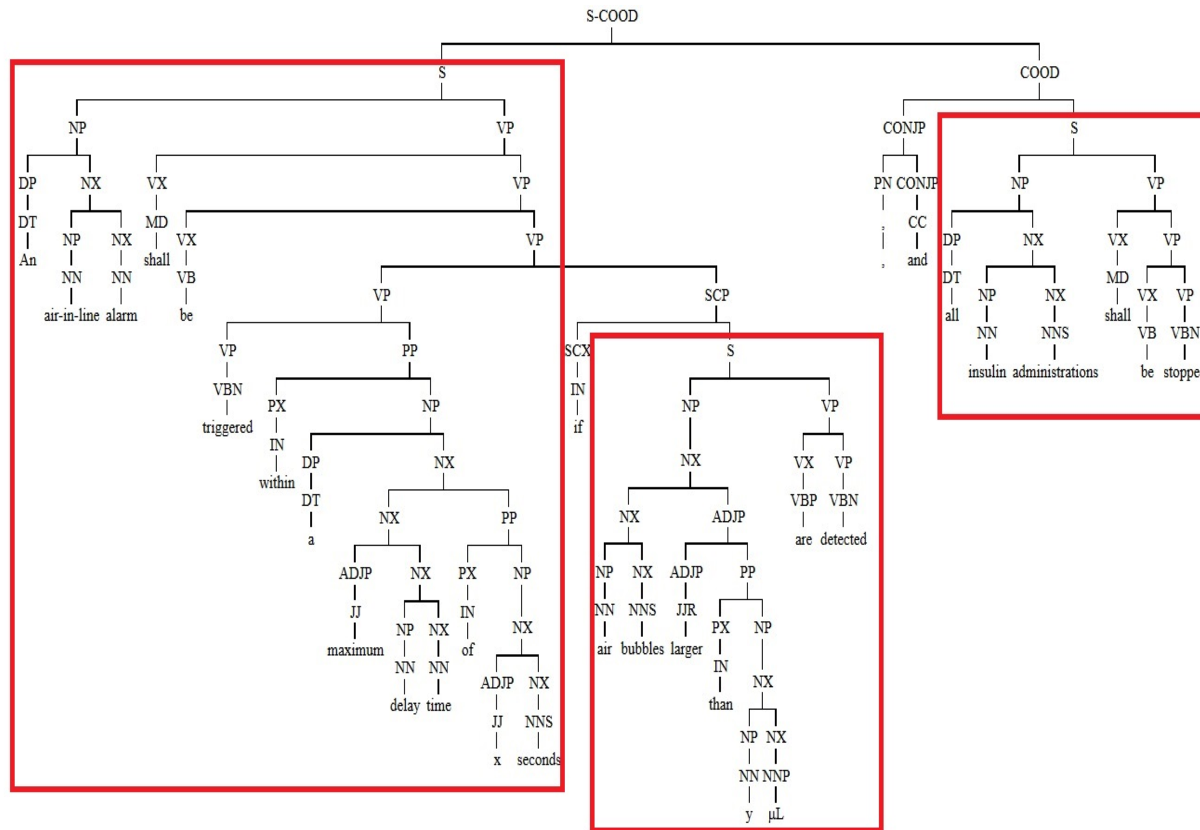


Figure 2.12. An Enju parsing tree shows three resulting TBSs after applying APTCER.

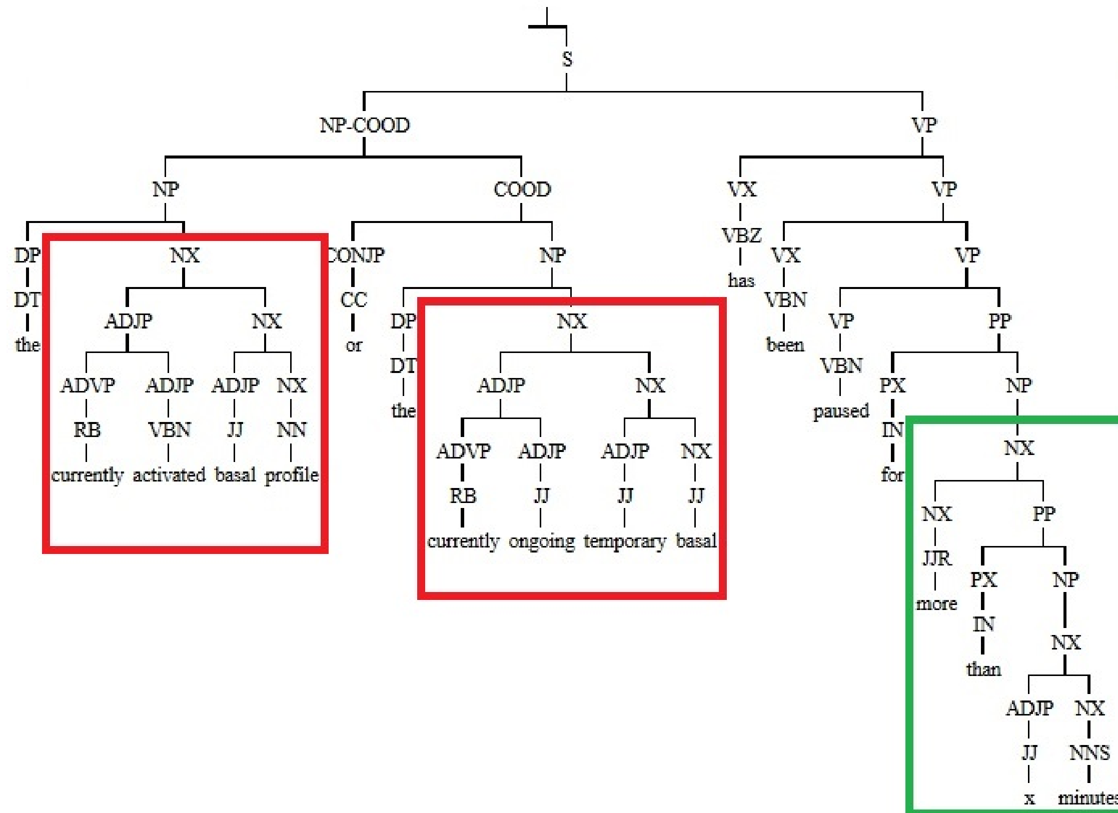


Figure 2.13. An Enju parsing tree portion shows the resulting TBS $\langle AP-list, TC-list \rangle)$ after applying APTCER.

Table 2.2. 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	10	10	0	6	0			
1.1.1	2	10	10	0	5	0	5	4	5
	3	10	10	0	6	0			
	1	7	7	0	3	2			
1.1.3	2	7	7	0	3	2	4	4	4
	3	7	7	0	3	1			
	1	24	12	3	5	1			
1.2.4 , 1.2.6, 1.2.7	2	24	18	0	8	0	10	10	14
	3	24	16	2	8	0			
	1	11	6	1	3	0			
1.3.5	2	11	8	0	4	1	4	4	4
	3	11	8	1	5	0			
	1	9	7	1	3	1			
1.8.2, 1.8.5	2	9	7	0	3	0	4	4	5
	3	9	7	0	3	0			
	1	6	6	0	3	1			
2.2.2, 2.2.3	2	7	6	0	3	1	3	3	4
	3	7	6	0	3	2			
	1	15	14	0	9	0			
3.1.1	2	14	12	0	7	0	5	3	3
	3	14	13	0	8	0			
	1	10	9	0	7	2			
3.2.5	2	7	7	0	4	1	3	3	3
	3	7	7	0	4	1			
	1	4	4	0	1	0			
3.2.7	2	4	4	0	1	1	3	3	3
	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 μL are detected, and all insulin administrations shall be stopped.*

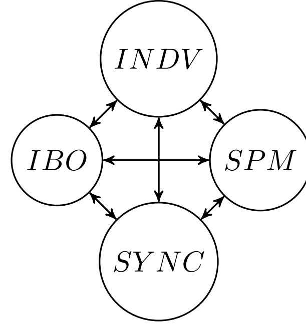
When procedure 6 is applied to this requirement, the first step is collecting the lists of TBS by applying APTCER, which gives three separate TBSs. TBS1 is "An air-in-line alarm shall be triggered within a maximum delay time of x seconds", while TBS2 is "air bubbles larger than y μL are detected", and 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 μL", and "insulin administrations". TC-list contains: "maximum delay time of x seconds" which is related to the AP: "air-in-line alarm" in TBS1.

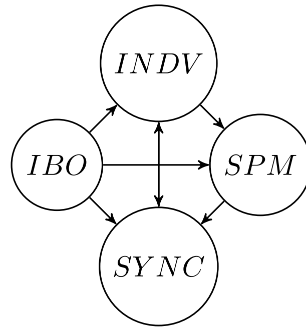
Now, the expert user intervenes to manipulate the AP-list, where APs can be deleted, modified or even inserted based on the expert user's domain knowledge. This yields the final AP-list as "air-in-line alarm" (ALRM), "air bubbles larger than y μL" (AIRB), "insulin administrations" (INSAD). Next, the AP-truth-table to define relations between APs is constructed as shown in Table 2.3.

Table 2.3. 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



(a) TS with all suggested transitions.



(b) TS after removing some transitions.

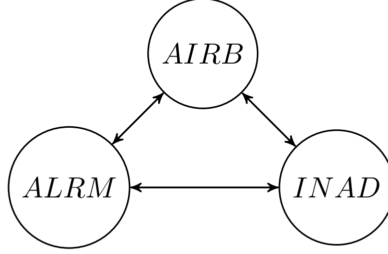
Figure 2.14. Finite state machine for suspension mode requirements (1.8.1 and 1.8.5).

As in Table 2.1, each row in the Table 2.3 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.

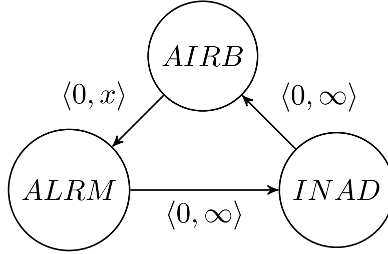
Then, Procedure 6 applies transitions between every two states, the expert user will approve or remove some unacceptable transitions as shown in Figure 2.15a. After following Procedure 6, the final TTS is shown in Figure 2.15b.

2.7. 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



(a) TTS with all suggested transitions.



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

Figure 2.15. Timed finite state machine for air-in-line requirement (1.6.1).

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. Firstly, 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 2.2 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.

Table 2.4. 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

As shown in Table 2.2, 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 2.4 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 2.2, 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.

2.8. Conclusion

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

2.9. References

[1] Eman M Al-qtiemat, Sudarshan K Srinivasan, Mohana Asha Latha Dubasi, and Sana 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)*. last accessed: 2018-09-10. 2018. URL: `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*. Ed. by Ahmed Bouajjani and Oded Maler. Vol. 5643. Lecture Notes in Computer Science. Springer, 2009, pp. 414–429. ISBN: 978-3-642-02657-7. DOI: `10.1007/978-3-642-02658-4_32`. URL: `https://doi.org/10.1007/978-3-642-02658-4%5C_32`.

[4] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram 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*. Ed. by Eerke A. Boiten, John Derrick, and Graeme Smith. Vol. 2999. Lecture Notes in Computer Science. Springer, 2004, pp. 1–20. ISBN: 3-540-21377-5. DOI: `10.1007/978-3-540-24756-2_1`. URL: `https://doi.org/10.1007/978-3-540-24756-2%5C_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*. Ed. by Toby C. Murray and Deian Stefan. ACM, 2016, pp. 91–96. ISBN: 978-1-4503-4574-3. DOI: `10.1145/2993600.2993611`. URL: `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] Panagiotis Manolios. "Mechanical Verification of Reactive Systems". last accessed: 2018-10-10. PhD thesis. University of Texas at Austin, Aug. 2001. URL: `http://www.ccs.neu.edu/home/pete/research/phd-dissertation.html`.

[8] Mohana Asha Latha Dubasi, Sudarshan K Srinivasan, and Vidura 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". In: last accessed: 2018-07-10. 2011.

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

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

[12] Daniel Aceituna, Hyunsook Do, and Sudarshan 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] Ian G Harris. "Extracting design information from natural language specifications". In: *Proceedings of the 49th Annual Design Automation Conference*. ACM. 2012, pp. 1256–1257.

[14] Hadas Kress-Gazit, Georgios E Fainekos, and George J Pappas. "Translating structured English to robot controllers". In: *Advanced Robotics* 22.12 (2008), pp. 1343–1359.

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

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

[17] Didar Zowghi, Vincenzo Gervasi, and Andrew 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] Vincenzo Gervasi and Didar Zowghi. "Reasoning about inconsistencies in natural language requirements". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 14.3 (2005), pp. 277–330.

[19] William Scott, Stephen Cook, and Joseph 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] Xusheng Xiao, Amit Paradkar, Suresh Thummalapenta, and Tao 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] Zuohua Ding, Mingyue Jiang, and Jens 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] Colette Rolland and Christophe Proix. "A natural language approach for requirements engineering". In: *International Conference on Advanced Information Systems Engineering*. Springer. 1992, pp. 257–277.

[23] Patricia Bouyer et al. "Model Checking Real-Time Systems". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. Springer, 2018, pp. 1001–1046.

[24] Daniel Knorreck, Ludovic Apvrille, and Pierre de Saqui-Sannes. "TEPE: a SysML language for time-constrained property modeling and formal verification". In: *ACM SIGSOFT Software Engineering Notes* 36.1 (2011), pp. 1–8.

[25] Aviral Shrivastava et al. "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, 69:1–69:6.

[26] Judith Peters, Nils Przigoda, Robert Wille, and Rolf Drechsler. "Clocks vs. instants relations: Verifying CCSL time constraints in UML/MARTE models". In: *2016 ACM/IEEE Interna-*

tional Conference on Formal Methods and Models for System Design, MEMOCODE, Kanpur, India, November 18-20. IEEE, pp. 78–84. ISBN: 978-1-5090-2791-0.

[27] Eun-Young Kang, Li Huang, and Dongrui 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*. Ed. by Hisham M. Haddad, Roger L. Wainwright, and Richard Chbeir. ACM, pp. 1492–1499.

[28] Gustavo Carvalho, Ana Cavalcanti, and Augusto Sampaio. “Modelling timed reactive systems from natural-language requirements”. In: *Formal Asp. Comput.* 28.5 (2016), pp. 725–765.

[29] Jameleddine Hassine. “Early modeling and validation of timed system requirements using Timed Use Case Maps”. In: *Requir. Eng.* 20.2 (2015), pp. 181–211.

[30] Yi Zhang, Raoul Jetley, Paul L Jones, and Arnab Ray. “Generic safety requirements for developing safe insulin pump software”. In: *Journal of diabetes science and technology* 5.6 (2011), pp. 1403–1419.

[31] Gerd Behrmann, Alexandre David, and Kim G Larsen. “A tutorial on uppaal”. In: *Formal methods for the design of real-time systems*. Springer. 2004, pp. 200–236.

3. SYNTHESIS OF REFINEMENT MAPS FOR REAL-TIME OBJECT CODE VERIFICATION

3.1. Introduction

Software safety is one of the key challenges facing the design process [1] of safety-critical embedded systems such as medical devices [2]. For example, infusion pump (a medical device that delivers medication such as pain medication, insulin, cancer drugs etc., in controlled doses to patients intravenously) has 54 class 1 recalls related to software issued by the US Food and Drug Administration (FDA) [3]. Class 1 means that the use of the medical device can cause serious adverse health consequences or death.

Despite the fact that testing is the dominant verification technique currently used in commercial design cycles [4], testing can only show the presence of faults, but it never proves their absence [5]. Alternate verification processes should be applied to the software design in conjunction with testing to assure system correctness and reliability. Formal verification can address testing limitations by providing proofs of correctness for software safety. Intel [6], Microsoft [7] and [8], and Airbus [9] have successfully applied formal verification processes.

Refinement-based verification [10] is a formal verification technique that has been demonstrated to be effective for verification of software correctness at the object code level [11]. To apply refinement-based verification, software requirements should be expressed as a formal model. Previously, we have proposed a novel approach to synthesize formal specifications from natural language requirements [12], and in a later work, we have also addressed timing requirements and specifications [13].

Our verification approach is based on the theory of Well-Founded Equivalence Bisimulation (WEB) refinement [10]. In the context of WEB refinement, both the implementation and specification are treated as Transition Systems (TSs). If every behavior of the implementation is matched

The content of this chapter has been submitted to the International Journal On Advances in Life Sciences 2020. The material in this chapter was co-authored by Eman M. Al-Qtiemat, Sudarshan Srinivasan, Zeyad Al-Odat, Mohana Asha Latha Dubas,and Sana Shuja. Eman M. Al-Qtiemat had primary responsibility for conducting experiments and collecting results. Eman M. Al-Qtiemat was the primary developer of the conclusions that are advanced here. Eman M. Al-Qtiemat also drafted and revised all versions of this chapter. Sudarshan Srinivasan drafted and revised all versions of this chapter. Sudarshan Srinivasan served as proofreader.

by a behavior of the specification and vice versa, then the implementation behaves correctly as prescribed by the specification. However, this is not easy to check in practice as the implementation TS and specification TS can look very different. The specification states obtained from the software requirements are marked with atomic propositions (predicates that are true or false in a given state). The implementation states are states of the microcontroller that the object code program modifies. As such, the microcontroller states includes registers, flags, and memory. The various possible values that these components can have during the execution of the object code program gives rise to the many millions of states of the implementation. To overcome this difference, WEB refinement uses the concept of a refinement map, which is a function that provided an implementation state, gives the corresponding specification state. Historically, one of the reasons that refinement-based verification is much less explored than other formal verification paradigms such as model checking is that the construction of refinement maps often requires deep understanding and intuitions about the specification and implementation [14]. However, once a refinement map is constructed, the benefit is that refinement-based verification is a very scalable approach for dealing with low-level artifacts such as real-time object code verification. We build refinement maps corresponding to formal specifications related to infusion pump safety and we also propose three possible generic refinement map templates, which is the first step toward automating the construction of refinement maps. Finally, we propose synthesising procedures of refinement maps for both functional and timing requirements, the new procedure allow an expert user intervention to assure the correctness of the system. The remainder of this chapter is organized as follows. Section 3.2 summarizes background information. Section 3.3 details related work. Section 3.4 presents the refinement maps and refinement map templates. Section 3.5 shows the proposed synthesis of refinement maps for system requirement. Conclusions and direction for future work are noted in Section 3.6.

3.2. Background

This section explores the definition of transition systems, the definition of refinement-based verification, and the synthesis of formal specifications as key terms related to our work.

3.2.1. Transition Systems

As stated earlier, transition systems (TSs) are used to model both specification and implementation in refinement-based verification. TSs are defined below.

Definition 4 *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.*

States are marked with Atomic Propositions (APs), which are predicates that are true or false in each state. The labeling function maps states to the APs that are true in every state. An example TS is shown in Figure 3.1. Here S = {S1, S2, S3, S4}, R = {(S1, S2), (S2, S4), (S4, S3), (S3, S4), (S3, S2), (S1, S3)} and, $L(S2)$ represents the atomic propositions that are true for the S2 state.

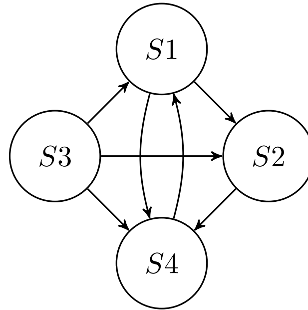


Figure 3.1. An example of a transition system (TS).

3.2.2. Timing 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 (Section 3.2.1), 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, timing transition systems (TTSs) [11] are used to represent the implementation and specification.

Definition 5 *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.*

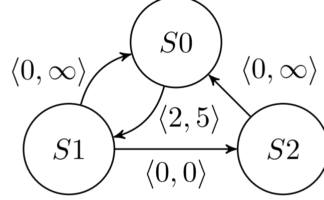


Figure 3.2. An example of a timing transition system (TTS).

Figure 3.2 shows an example of a timing transition system that consists of three states { S0, S1, S2 }, for instance; if the system is in state $S0$ it can go to state $S1$ only between 2 and 5 units of time, while going from $S1$ to $S2$ the time is zero meaning that it should happen immediately, going from $S2$ to $S1$ the time is zero to infinity which means that it can happen any point of time, and so on.

3.2.3. Refinement-Based Verification

Our verification process is based on the theory of Well-Founded Equivalence Bisimulation refinement. A detailed description of this theory can be found in [10]. Here, we give a very high-level overview of the key concepts. As stated earlier, WEB refinement provides a notion of correctness that can be used to check an implementation TS against a specification TS. One of the key features is that WEB refinement accounts for stuttering, which is the phenomenon where multiple but finite transitions of the implementation can match a single transition of the specification. This is a very key feature because the control code implements many functions and only some of these functions maybe relevant to the safety property being verified. Therefore, the code maybe doing a number of things that do not relate to the property and will therefore be stuttering a lot w.r.t. the specification. Another key feature of WEB refinement is refinement maps, which is the focus of this work. Refinement maps are functions that map implementation states to specification states. There is a lot of flexibility in how refinement maps can be defined. This allows for low-level implementations to be verified against high-level specifications.

Definition 6 *(WEB Refinement): Let* $M = \langle S, R, L \rangle$*,* $M' = \langle S', R', L' \rangle$*, and r:* $S \rightarrow S'$*. M is a WEB refinement of* M' *with respect to refinement map r, written* $M \approx r\ M'$*, if there exists a relation, B, such that* $\langle\ \forall\ s \in S :: sB(r.s)\rangle$ *and B is a WEB on the TS* $\langle\ S \uplus S',\ R \uplus R',\ L\ \rangle$*, where* $L.s = L'(s)$ *for s and S' state and* $L.s = L'(r.s)$ *otherwise.*

3.2.4. Synthesis of Functional Formal Specifications

Our approach for development and study of refinement maps is based on the formal TS specifications. We have developed a previous approach to transform functional requirements into formal specifications [11]. Since this work is closely tied to the prior work, we briefly review it here. Figure 3.3 summarizes the transformation procedure, the main steps are explained as follows: functional requirement is fed as an input, an English parser called Enju was used to get the parse tree the requirement. The first step of computing the TSs is to apply Atomic Proposition Extraction Rule (APER) extract the APs from the requirements. We have developed three Atomic Proposition Extraction Rules (APERs) that work on the parse tree of the requirement to get an initial list of APs. The resulting list is subjected to an expert user check (User Input), where the APs might be appended, eliminated or revised based on the expert users domain knowledge. A high-level procedure for specification transition system synthesis has been proposed to compute the states and transitions using the resulting list of APs under expert user supervision. Finally, the transition system is created using the resulting list of states and transitions. The output of the procedure is a formal specification in a TS form.

3.2.5. Synthesis of Timing Formal Specifications

Some system requirements have timing constraints, they are called timing requirements. We have proposed a previous approach that working on transforming timing requirements into formal specifications [13]. Figure 3.4 shows the main steps of the synthesising procedure. A brief description of this approach is explained as follow: Timing requirement is fed an an input of the procedure. As in the previous procedure, Enju parser is used to get the parse tree that corresponds to the entered requirement. An Atomic Proposition and Timing Constrains Extraction Rule (APTCER) has been applied on the resulted parse tree to get an initial list of APs and Timing Constrains (TCs). APs and TCs are paired together and they are considered the base of a TTS. Then, set of states are defined based on the resulting list of APs. Transitions are applied between every two states. TCs are assigned to the transitions. This procedure allows input from domain expert as shown in Figure 3.4. Finally, the TTS is created. The output of this procedure is a formal specification with timing constraints as a TTS.

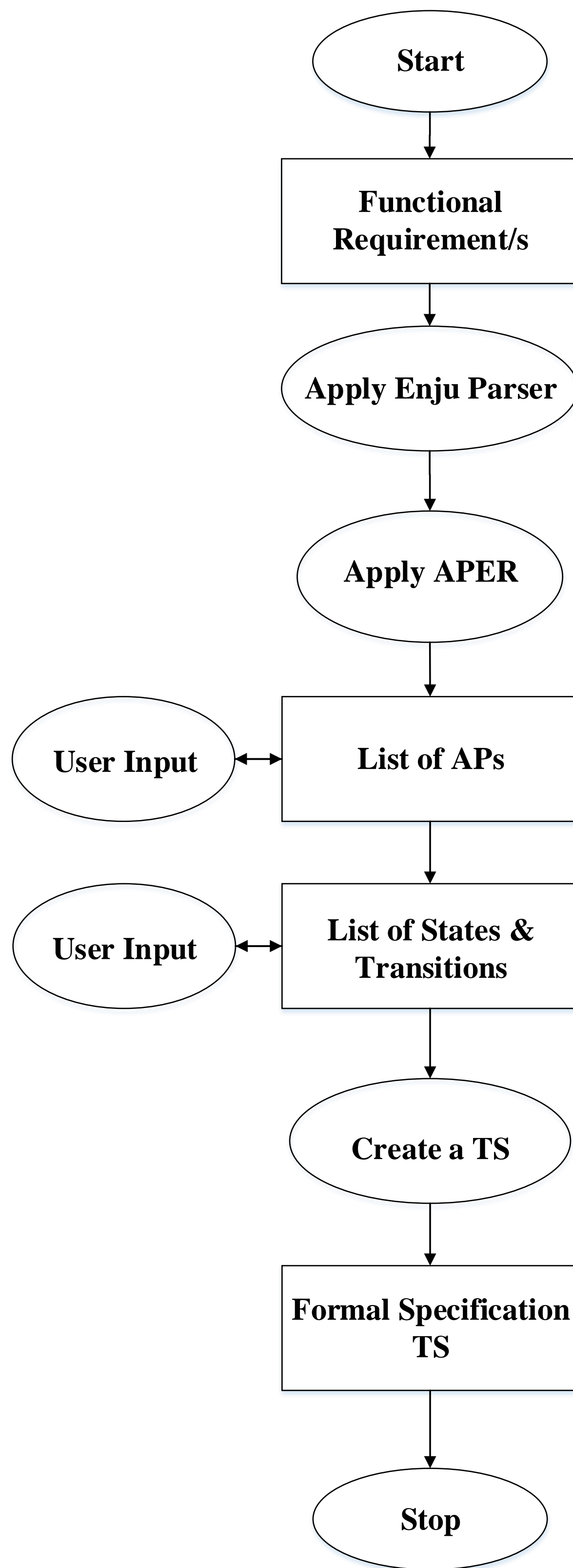


Figure 3.3. Flowchart of formal model synthesis procedure for Functional Requirements.

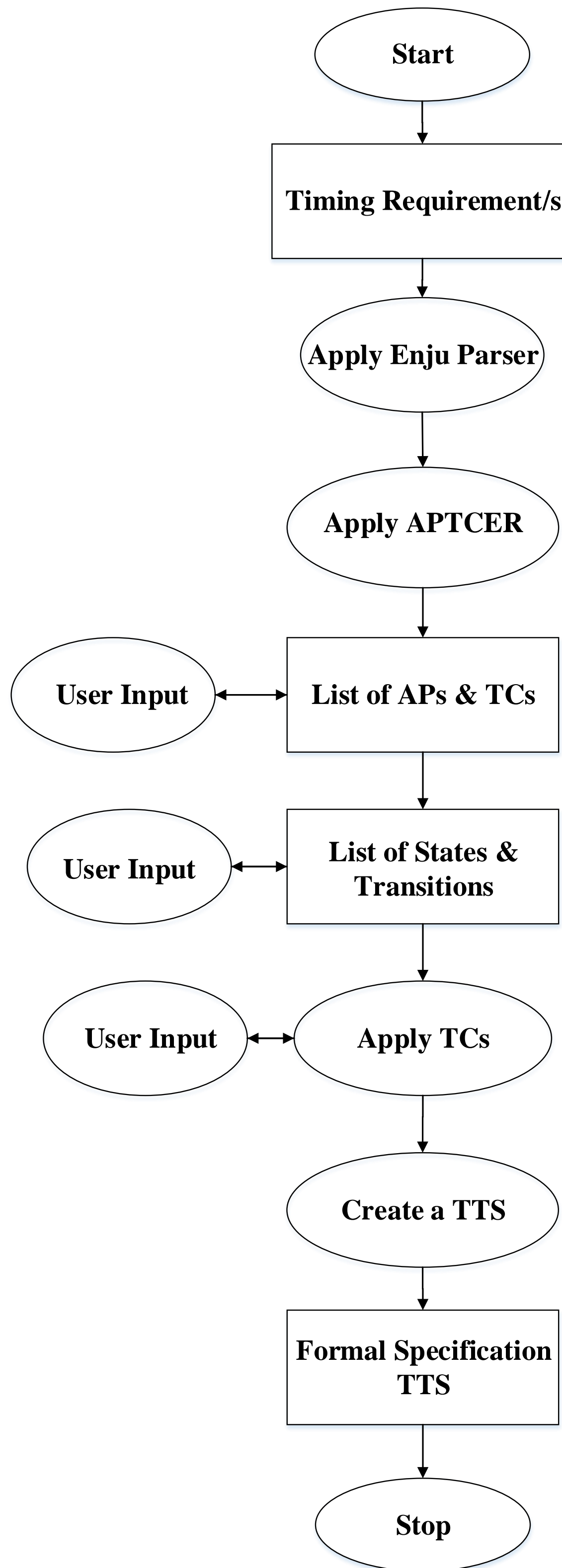


Figure 3.4. Flowchart of formal model synthesis procedure for Timing Requirements.

3.3. Related Work

This section summarizes a few works on applying refinement processes to get more concrete specifications and refinement-based verification. None of these works are applied to insulin pump formal specifications as our work. To the best of our knowledge, these are the most related state of art in this area of study.

Dubasi *et al.* [15] presented a formal verification technique based on the theory of Well-Founded Simulation (WFS) refinement in which the correctness of low-level real-time interrupt-driven object code programs is automated. The main difference between this work and our work is that our work automates the refinement map construction while Dubasi's work constructed the refinement maps manually and automated the refinement verification process. Klein *et al.* [16] introduced a new technique called State Transition Diagrams (STD). It is a graphical specification technique that provides refinement rules, each rule defines an implementation relation on STD specification. The proposed approach was applied to the feature interaction problem. The refinement relation was utilized to add a feature or to define the notion of conflicting features.

Rabiah *et al.* [17] developed a reliable autonomous robot system by addressing A* path planning algorithm reliability issue. A refinement process was used to capture more concrete specifications by transforming High-Level specification into equivalent executable program. Traditional mathematical concepts were used to capture formal descriptions.Then, Z specification language was employed to transform mathematical description to Z schemas to get formal specifications. Z formal refinement theory was used to obtain the implementation specification.

Spichkova [18] proposed a refinement-based verification scheme for interactive real time systems. The proposed work solves the mistakes that rise from the specification problems by integrating the formal specifications with the verification system. The proposed scheme translates the specifications to a higher-order logic, and then uses the theorem prover (Isabelle) to prove the specifications. Using the refinement-based verification, this scheme validates the refinement relations between two different systems. The proposed design was tested and verified using a case study of electronic data transmission gateway.

A new approach that focuses on the refinement verification using state flow charts has been presented by Miyazawa *et al.*[19]. They proposed a refinement strategy that supports the

sequential C implementations of the state flow charts. The proposed design benefited from the architectural features of model to allow a higher level of automation by retrieving the data relation in a calculation style and rendering the data into an automated system. The proposed design was tested and verified using Matlab Simulink SDK. Through the provided case study, the scheme was able to be scaled to different state charts problems.

Cimatti *et al.* proposed a contract-refinement scheme for embedded systems [20]. The contract-refinement provides interactive composition reasoning, step-wise refinement, and principled reuse refinements for components for the already designed or independently designed components. The proposed design addresses the problem of architectural decomposition of embedded systems based on the principles of temporal logic to generate a set of proof obligations. The testing and verification of the Wheel Braking System (WBS) case study show that the proposed design can detect the problems in the architectural design of the WBS.

Bibighaus [21] employed the Doubly Labeled Transition Systems (DLTS) to reason about possibilities security properties and refinement. This work was compared with three different security frameworks when applied to large class systems. The refinement framework in this work preserves and guarantees the liveness of the model by verifying the timing parameter of the model. The analysis results show that the proposed design preserves the security properties to a series of availability requirements.

A novel approach has been presented [22] to formally specify and analyze the certification process of Partitioning Operating Systems (POSs) by integrating refinement and ontology. An ontology of POSs was developed as an intermediate model between informal descriptions of ARINC 653 and the formal specification in Event-B. A semiautomatic translation has been implemented from the ontology and ARINC 653 into Event-B. Six hidden failures in ARINC were happened and fixed during the formal analysis in the Event-B specification. The existence of these errors has been validated in two open-source POSs: XtratuM and POK. The degree of automatic verification of the Event-B specification reached a higher level because of the ontology. By validation, they have also noticed some errors in open-source POSs. The proposed methodology has shown capability to to formalize and verify systems according to system's informal standards and requirements.

Human factors consider as the most obvious cause of failures especially when a human deals with critical systems such as nuclear and medical systems. A new methodology for develop-

ing a Human-Machine Interface (HMI) has been proposed [23], it uses a correct by construction approach. A HMI was developed independently using incremental refinement. Human interactions is dependent on testing, which can not guarantee the absence of failures. Formal method was used to assure the correctness of the human interactions. Even-B modeling language has been used to formalize the internal consistency with respect to the safety properties and events. This generic refinement strategy supports a development of the Model-View-Controller (MVC) architecture.

A specification development method and a generic security model were proposed based on refinement for ARINC Separation Kernels (KSs) [24]. A step-wise refinement framework was presented. Two levels of functional specification are developed by the refinement. Kernel initialization, inter-partition communication, two-level scheduling, and partition and process management were modeled. Isabelle/HOL theorem prover was used to carry out the formal specification and its security proofs. Mechanical check proofs were given to solve convert channels in separation kernels.

Fayolle *et al.* joined Algebraic State-Transition Diagrams (ASTD) with an Event-B specification for better understanding of the system behaviour [25]. They proposed an approach that works on incrementally refine the specification couplings, it takes the new refinement relations and consistency into consideration between data and control system specifications. This work had shown how to use two complementary languages for formal modeling, a railway CBTC-like case study were used. In addition, the principle of complementarity and consistency was explored between ASTD and B-like refinements. Separation between data and behavioural system's aspects were accomplished.

The issues of validating formal models were studied and executed using Event-B method [26]. Firstly, new techniques were created and discussed which allow model execution to be at all abstraction levels. To overcome barriers comes form non-deterministic features, users intervention such as modifying the model or providing ad hoc implementations were needed. Secondly, a new formal definition of the notion of fidelity was given, this definition assures specifying all the observable behaviors of the executable models by the non-deterministic models.

Smith *et al.* in [27] provided formal link between trace refinement and linearizability, a comparison between these correctness conditions were explored. The main conclusion of this work is generally that trace refinement reveals linearizability, but linearizabilit does not reveals trace refinement. However, linearizability can reveal trace refinement but under specific conditions.

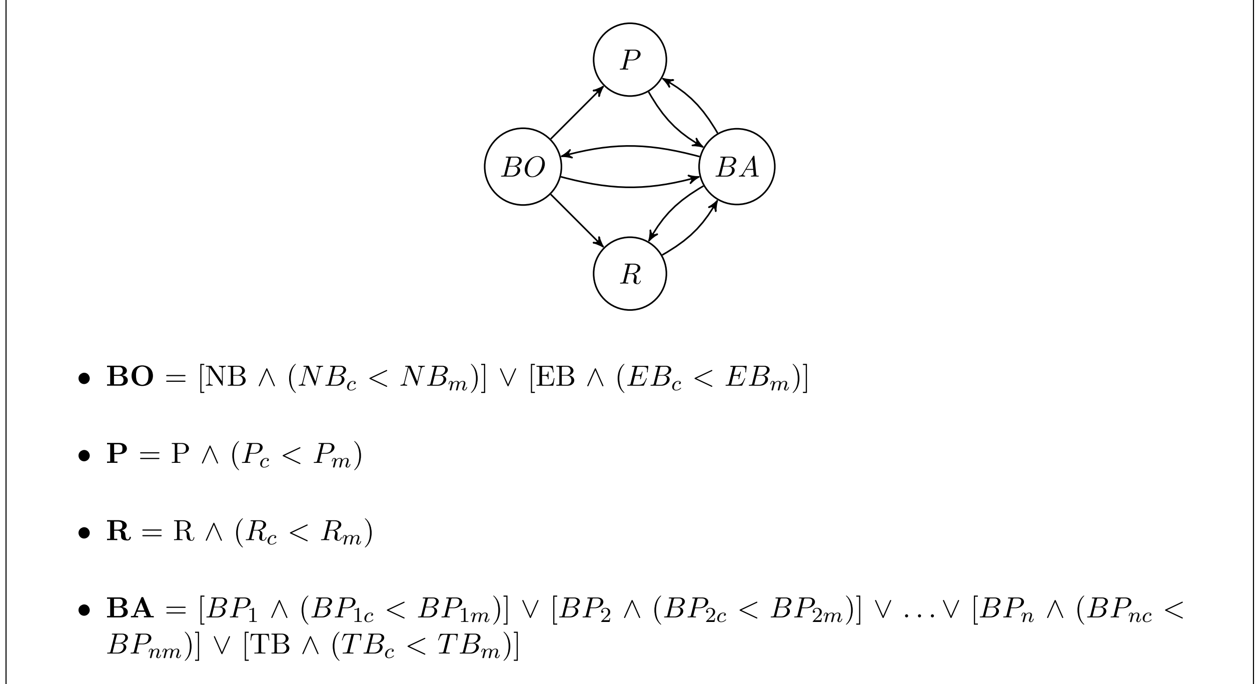
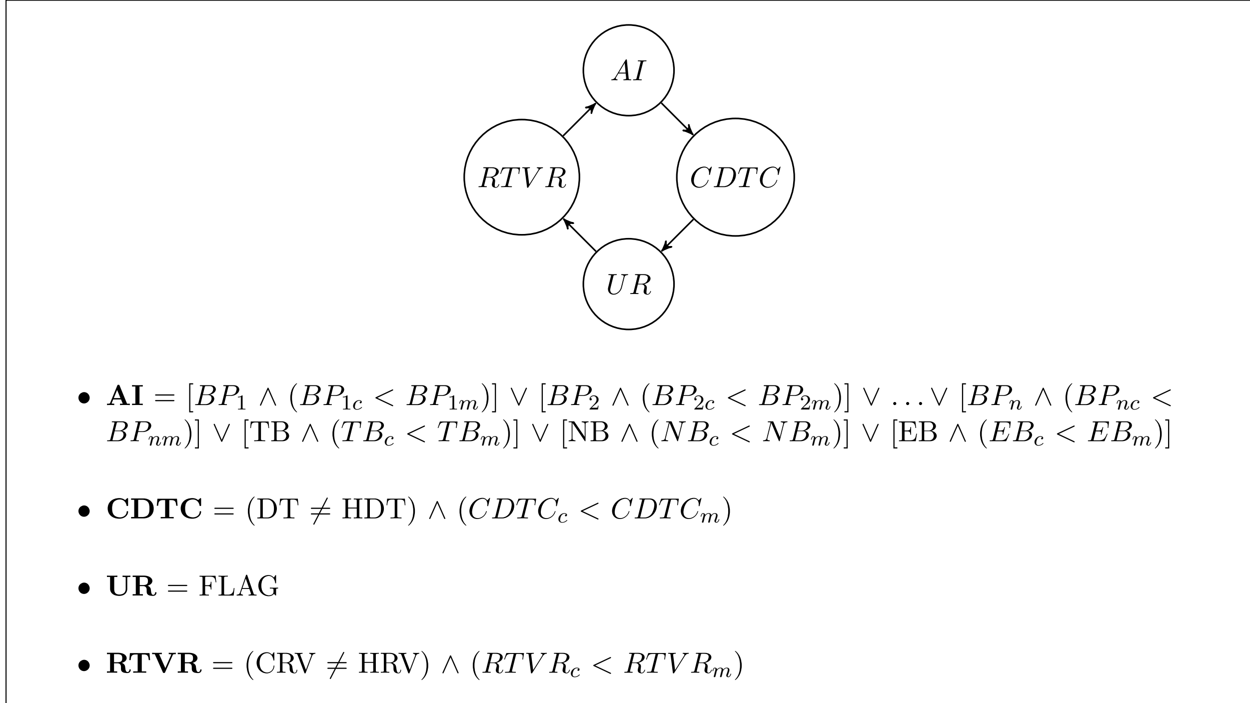


Figure 3.5. A formal presentation of requirement 1.1.1 and the suggested refinement maps.

Firstly, trace refinement can prove both safety and liveness properties, while linearizability can only prove safety properties. Secondly, the fact that trace refinement based on the identification of when the implementation operations are noticed to happen. They also studied these differences in the verification context of concurrent objects. Many other papers discussed and analyzed refinement concepts in the context of verifying concurrent objects [28–38].

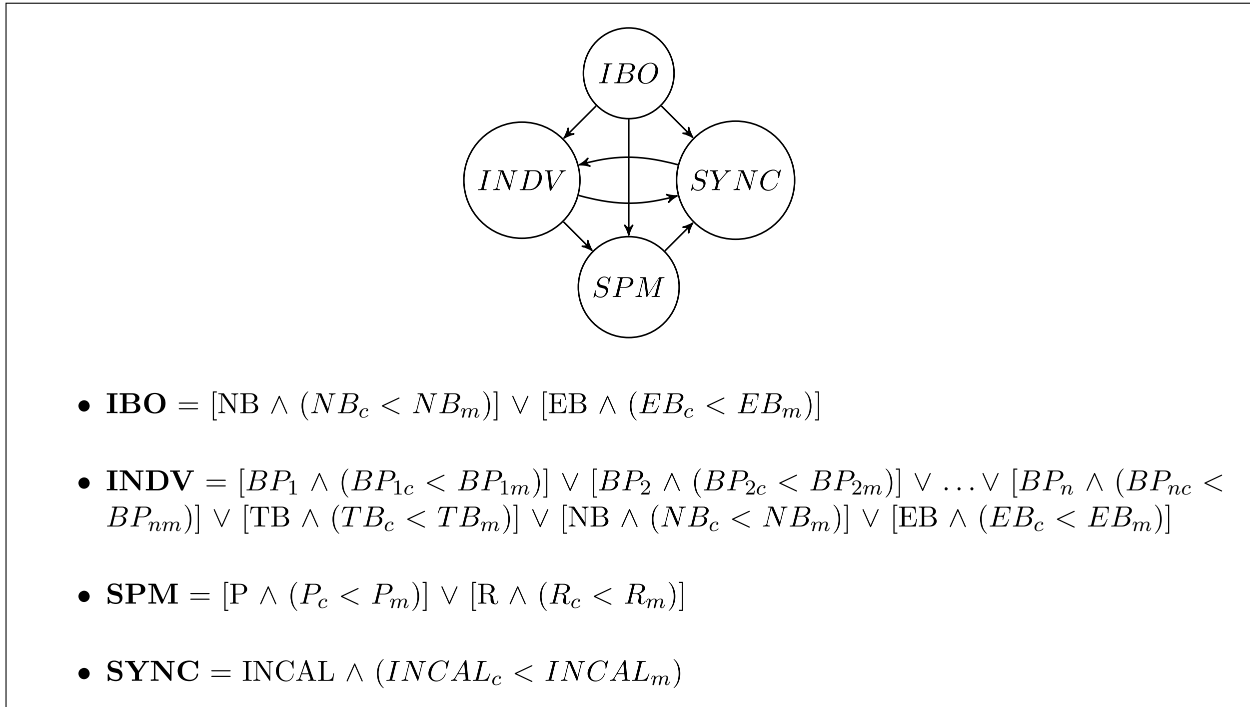
3.4. Refinement Maps and Refinement Map Templates

Figures 3.5-3.11 show the formal specification TS for 8 insulin pump safety requirements. In this section, formal specification TTS corresponding to 4 insulin pump timing requirements are added in Figures 3.12-3.15. The figures show the refinement map we have developed corresponding to each specification. The formal specifications TSs [12] and TTSs [13] were developed as part of our previous work in this area. As can be seen from the figures, each TS or TTS consists of a set of states and the transitions between the states. Also, each state is marked with the atomic propositions that are true in the state. For TTSs in Figures 3.12-3.15, time bounds conditions are added on each transition. Our strategy for constructing the refinement maps is as follows. A specification state can be constructed from an implementation state by determining the APs that



- **AI** = $[BP_1 \wedge (BP_{1c} < BP_{1m})] \vee [BP_2 \wedge (BP_{2c} < BP_{2m})] \vee \ldots \vee [BP_n \wedge (BP_{nc} < BP_{nm})] \vee [\text{TB} \wedge (TB_c < TB_m)] \vee [\text{NB} \wedge (NB_c < NB_m)] \vee [\text{EB} \wedge (EB_c < EB_m)]$
- **CDTC** = $(\text{DT} \neq \text{HDT}) \wedge (CDTC_c < CDTC_m)$
- **UR** = FLAG
- **RTVR** = $(\text{CRV} \neq \text{HRV}) \wedge (RTVR_c < RTVR_m)$

Figure 3.6. A formal presentation of requirement 1.1.3 and the suggested refinement maps.



- **IBO** = $[\text{NB} \wedge (NB_c < NB_m)] \vee [\text{EB} \wedge (EB_c < EB_m)]$
- **INDV** = $[BP_1 \wedge (BP_{1c} < BP_{1m})] \vee [BP_2 \wedge (BP_{2c} < BP_{2m})] \vee \ldots \vee [BP_n \wedge (BP_{nc} < BP_{nm})] \vee [\text{TB} \wedge (TB_c < TB_m)] \vee [\text{NB} \wedge (NB_c < NB_m)] \vee [\text{EB} \wedge (EB_c < EB_m)]$
- **SPM** = $[\text{P} \wedge (P_c < P_m)] \vee [\text{R} \wedge (R_c < R_m)]$
- **SYNC** = $\text{INCAL} \wedge (INCAL_c < INCAL_m)$

Figure 3.7. A formal presentation of requirement 1.8.2 and 1.8.5 and the suggested refinement maps.

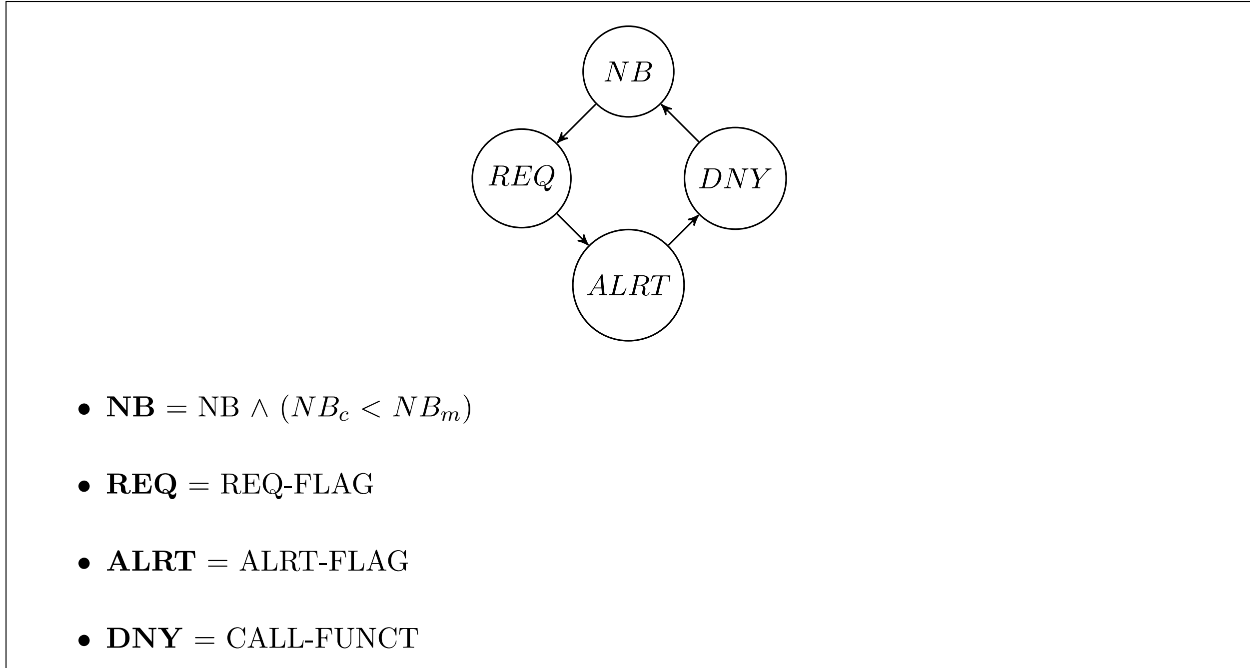


Figure 3.8. A formal presentation of requirement 1.3.5 and the suggested refinement maps.

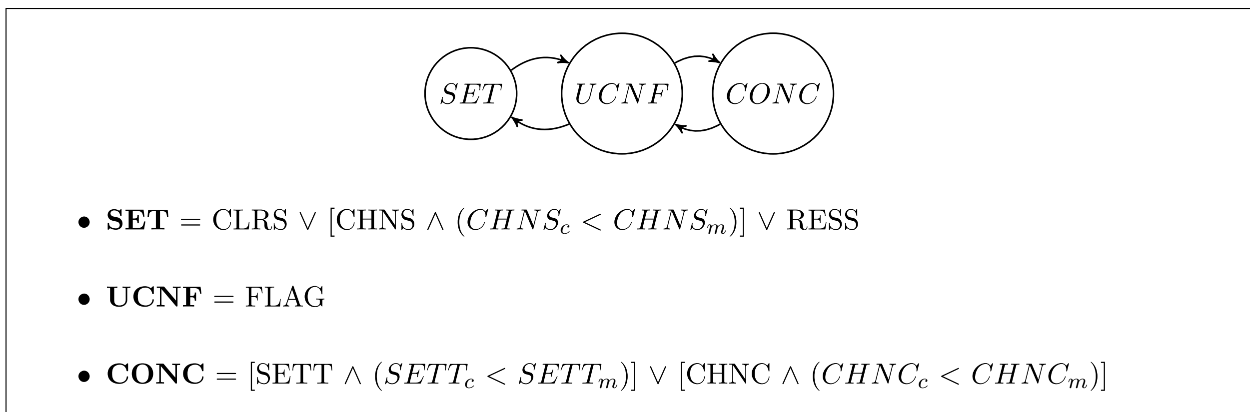


Figure 3.9. A formal presentation of requirement 2.2.2 and 2.2.3 and the suggested refinement maps.

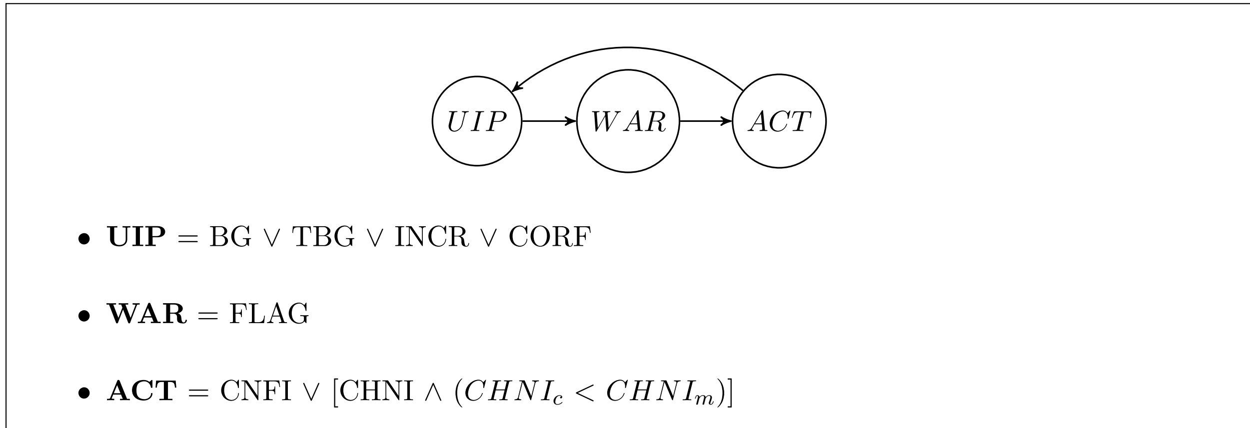


Figure 3.10. A formal presentation of requirement 3.2.5 followed by the suggested refinement maps.

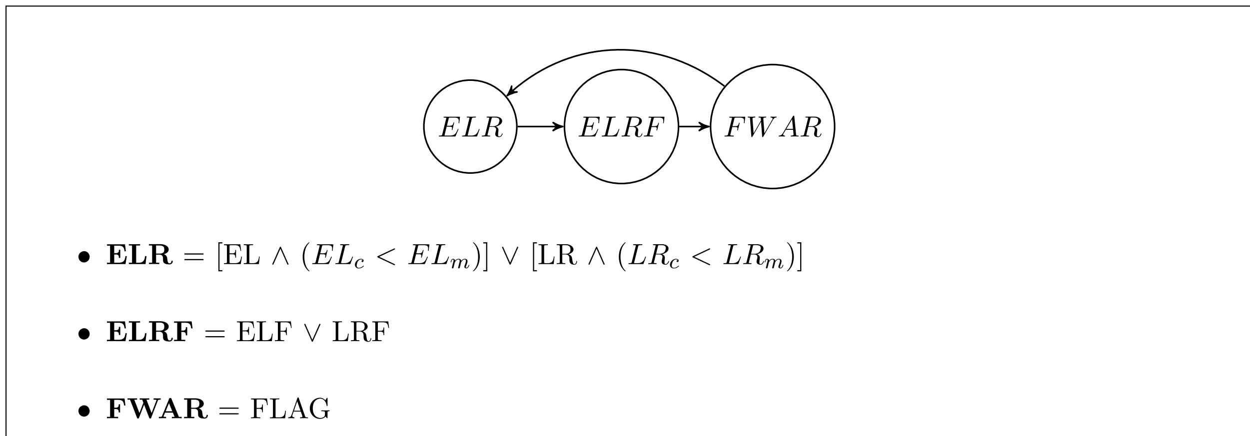


Figure 3.11. A formal presentation of requirement 3.2.7 followed by the suggested refinement maps.

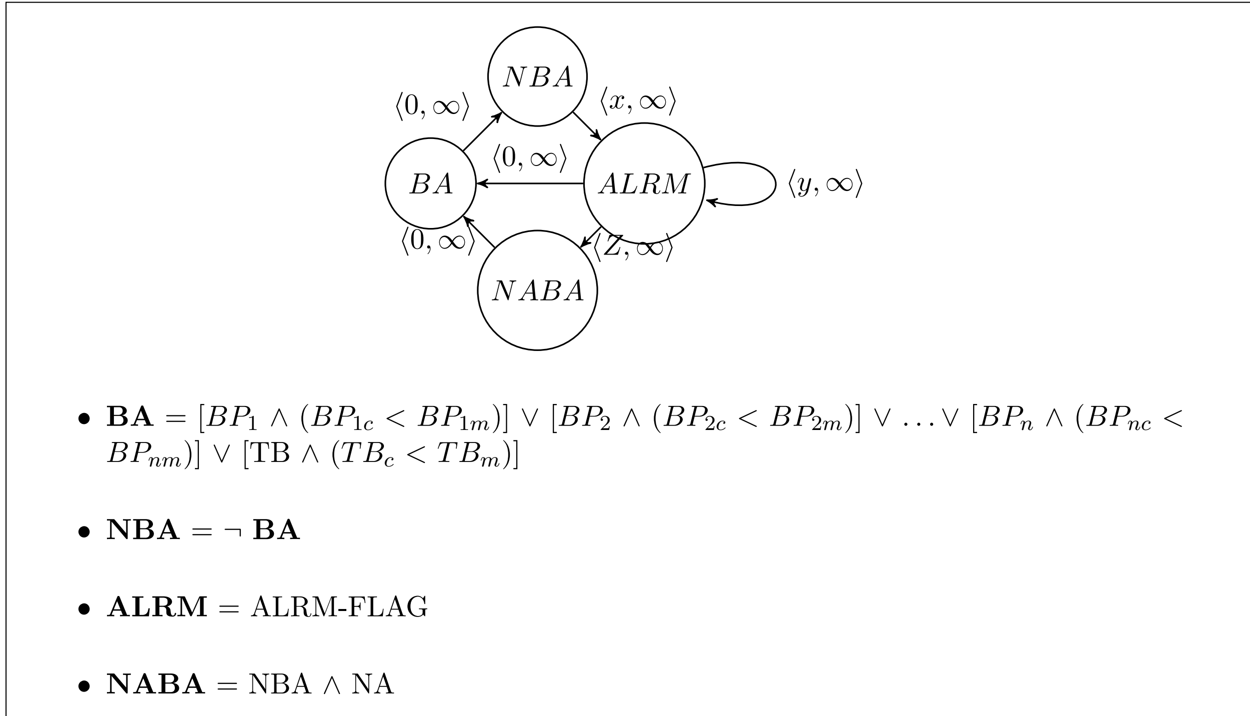


Figure 3.12. A formal presentation of the timing requirement 1.2.8 and the suggested refinement maps.

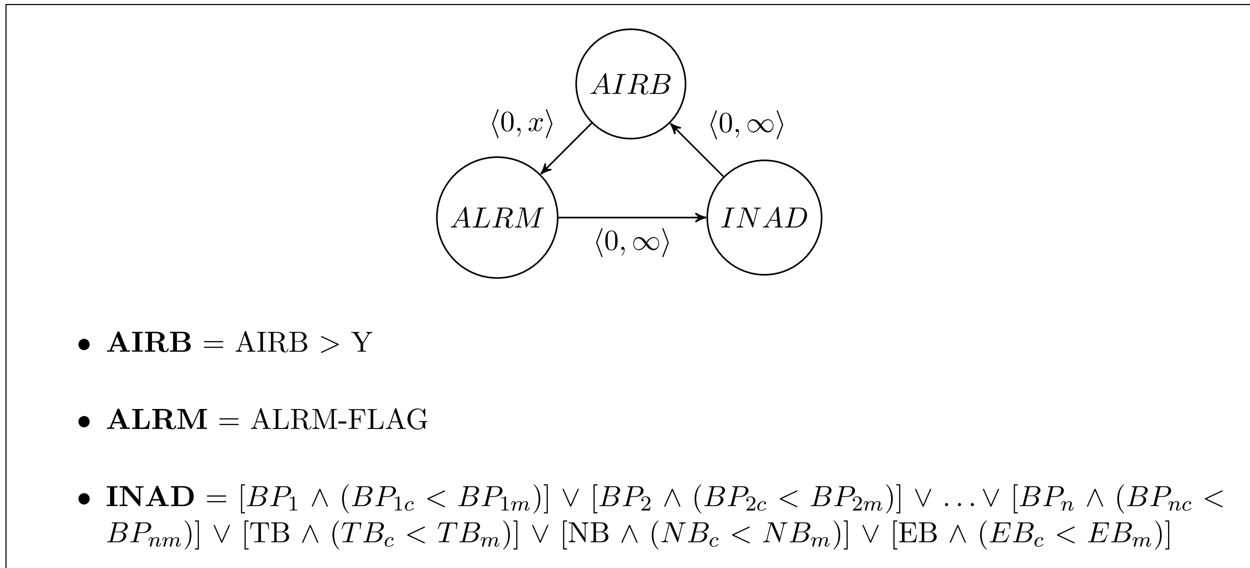


Figure 3.13. A formal presentation of the timing requirement 1.6.1 followed by the suggested refinement maps.

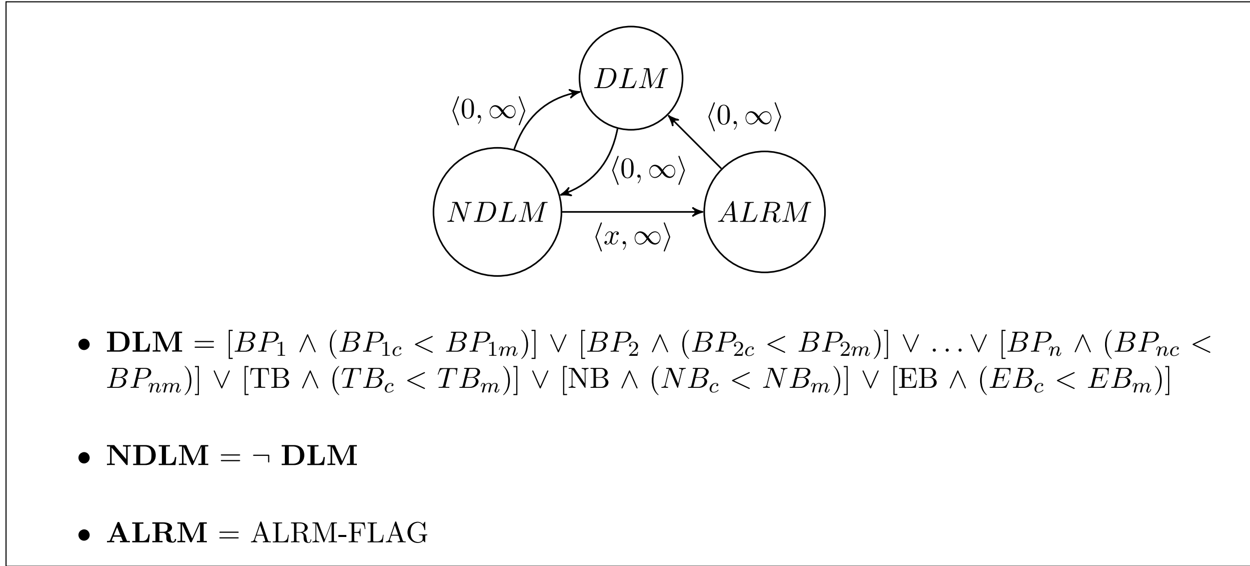


Figure 3.14. A formal presentation of the timing requirement 1.8.4 followed by the suggested refinement maps.

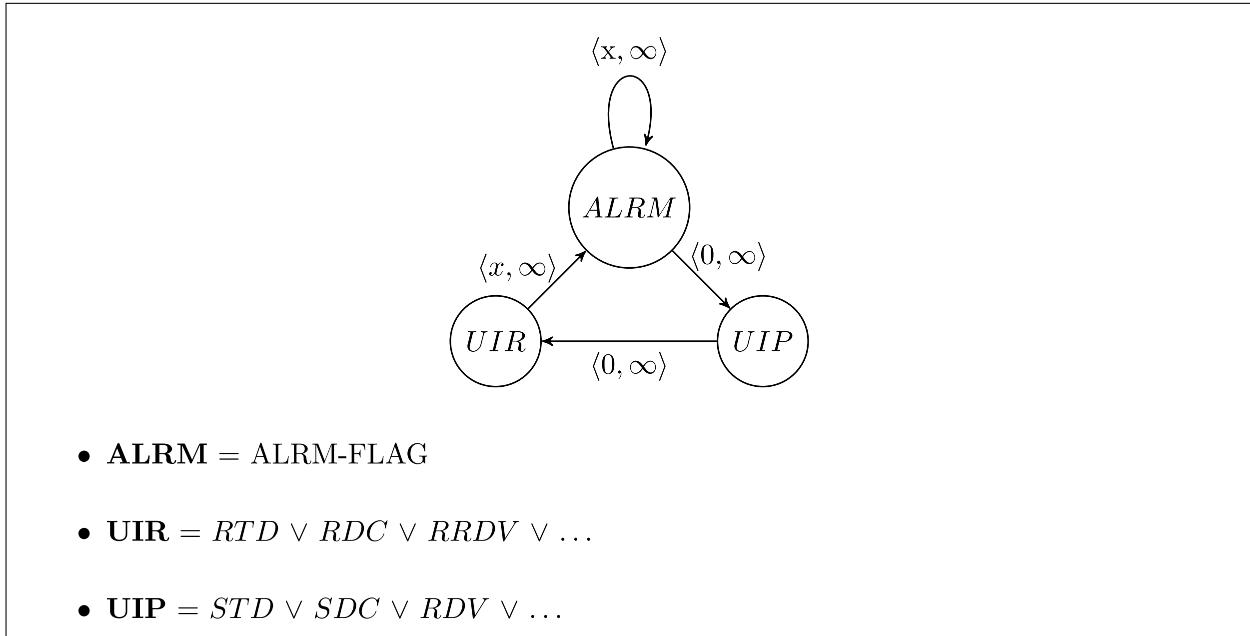


Figure 3.15. A formal presentation of the timing requirement 2.2.1 followed by the suggested refinement maps.

are true in the implementation state. If a specification has n APs, then we construct one predicate function for each AP. The predicate functions take the implementation state as input and output a predicate value that indicates if the AP is true in that state or not. Thus, the collection of such predicate functions is the refinement map.

We next discuss the refinement map for the specification in Figure 3.5. The safety specification from [39] is as follows: "The pump shall suspend all active basal delivery and stop any active bolus during a pump prime or refill. It shall prohibit any insulin administration during the priming process and resume the suspended basal delivery, either a basal profile or a temporary basal, after the prime or refill is successfully completed." The APs corresponding to this safety requirement are (1) BO: active bolus delivery; (2) BA: active basal delivery; (3) P: priming process; and (4) R: refill process. The refinement map however has to account for what is happening in the implementation code and relate that to the atomic propositions.

The predicate function for BO uses several variables from the code including NB: Normal Bolus and EB: Extended Bolus as there are more than one type of Bolus dose supported by the system. So the AP BO should be true if there is a NB or an EB. NB is only a flag that indicates that a normal bolus should be in progress. The actual bolus itself will continue to occur as long as a counter that keeps track of the bolus has not reached its maximum value. Therefore, for example for a normal bolus, we use a conjunction of NB and the condition that the NB counter (NB_c) is less than its possible maximum value (NB_m). We use a similar strategy for the extended bolus as well. This refinement map template works for all processes similar to a Bolus dosage delivery, such as basal dosage delivery, priming process, and refill process. Therefore, we term this refinement map template as "process template." For the basal dosage (BA AP) a number of basal profiles (BPs) are possible that accounts for BP_1 thru BP_n. TB stands for temporary basal. As can be noted from Figures 3.6-3.15, the process template accounts for a large number of predicate functions corresponding to APs.

The second refinement map template is a simple one called the "projection template," which is used when the AP in the specification TS corresponds directly to a variable in the code. An example of the projection template can be found in Figure 3.6, where the User Reminder (UR) AP is mapped directly from a flag variable in the code that corresponds to the user reminder. A

variation of this template is a boolean expression of Boolean variables in the code. An example of such an AP is the UIP AP in Figure 3.10.

The third refinement map template is called the "value change template," which is used when the AP is true only when a value has changed. An example use of this template can be found in Figure 3.6 for the CDTC AP. CDTC corresponds to the change in drug type and concentration and is true when the drug type or concentration is changed. For the drug type change, DT is the variable that corresponds to the drug type. The question here is how to track that a value has changed. The idea is to use history variables. HDT is a history variable that corresponds to the history of the drug type, i.e., the value of the drug type in the previous cycle. If HDT is not equal to DT in a code state, then we know the drug type has changed. The inequality of HDT and DT is used to construct the predicate function. For all the safety requirements analyzed, these three refinement map templates cover all the APs. For timed specifications, we next discuss the refinement map for the specification in Figure 3.13. The safety specification from [39] is as follows: "An air-in-line alarm shall be triggered within a maximum delay time of x seconds if air bubbles larger than y μL are detected, and all insulin administrations shall be stopped." The APs corresponding to this safety requirement are (1) AIRB: air bubbles; (2) ALRM: air-in-line alarm; (3) INAD: insulin administration. The predicate function for INAD uses several variables from the code including BPs; TB; NB; and EB as explained above. The AP INAD should be true if one of these variables is true and its counter variable is less than the maximum value. This AP is considered as an example use of the process template. For the ALRM AP, it is a simple example of the projection template, it should be true if its corresponding flag is true. The AIRB AP shows another variation of the value change template, which is depends on the changing value of the Air Bubbles (AB) variable. If the AB variable value is greater than Y (Y is a predefined value of the number of bubbles and it is based on the pump model), the AirB AP will be true. Table 3.1 gives the expansions for all the abbreviations used in Figures 3.6-3.15, so that the corresponding refinement maps can be comprehended by the reader.

3.5. Synthesis of Refinement Maps for System Requirements

This section explains new automation procedures for constructing refinement maps for both functional and timing system requirements from [39]. The first part of this work uses our previously

proposed algorithms for synthesising formal specifications from natural language requirements [12] [13].

Procedure 7 shows the overall flow for computing the refinement map template for each AP in a functional requirement. A set of functional requirements in natural language form are fed as input to the procedure. Three templates lists are the output of the procedure, each list will contain set of APs based on the heuristic data from the parsed trees belong to every input requirement.

Procedure 7 Procedure for synthesizing Refinement Maps for functional requirements

---

Require: Set of Functional Requirements
1: Projection-Temp-list $\leftarrow \emptyset$;
2: Process-Temp-list $\leftarrow \emptyset$;
3: Value-Temp-list $\leftarrow \emptyset$;
4: AP_f-list $\leftarrow \emptyset$;
5: **for each** $Req_f \in$ Functional Requirements **do**
6: Apply_SPFR(Req_f);
7: AP_f-list $\leftarrow$ Get_AP-list(SPFR);
8: **for each** $AP_f \in AP_f$-list **do**
9: Sub-tree $\leftarrow$ Get_Sub-tree (AP_f);
10: X = Head(Sub-tree);
11: **if** $[(X = \text{NX}) \wedge (\text{RightChild}(X) = \text{PP}) \wedge (\text{LeftChild}(X) = \text{NX})] \vee [(X = \text{VP}) \wedge (\text{RightChild}(X) = \text{NP}) \wedge (\text{LeftChild}(X) = \text{VX})]$ **then**
12: Projection-Temp-list $\leftarrow$ Projection-Temp-list $\cup$ AP_f;
13: **else**
14: **if** $[(X = \text{NX}) \wedge (\text{LeftChild}(X) = \text{VP})] \vee [(X = \text{VP}) \wedge (\text{RightChild}(X) = \text{CP}) \wedge (\text{LeftChild}(X) = \text{VX})]$ **then**
15: Value-Temp-list $\leftarrow$ Value-Temp-list $\cup$ AP_f;
16: **else**
17: Process-Temp-list $\leftarrow$ Process-Temp-list $\cup$ AP_f;
18: Projection-Temp-list $\leftarrow$ USR_IN(Projection-Temp-list);
19: Value-Temp-list $\leftarrow$ USR_IN(Value-Temp-list);
20: Process-Temp-list $\leftarrow$ USR_IN(Process-Temp-list);

Three empty template lists are defined; projection template list (line 1), process template list (line 2), and value template list (line 3). A list for functional requirement's APs (AP_f-list) is

initialized to null (line 4). Each requirement is input to the Synthesising Procedure for Functional Requirements (SPFR) (line 6) which comes up with formal specifications (explained in Section 3.2.4). A function called Get_AP-list is used to obtain the resulting AP-list from the SPFR into the AP_f-list (line 7). A function called Get_Sub-tree is applied to each entry (AP_f) in the AP_f-list, this function returns the sub tree that corresponds to the AP_f from Enju parsed tree (line 9). A function is called Head stores the head category of the sub tree in variable X (line 10). Check if X is of NX category, right child of X is PP, and left child of X is NX (line 11), then AP is added to the projection template list (line 12). Also if X is of VP category, right child of X is NP, and left child of X is VX (line 11), so AP is added to the projection template list too. For the AP to be stored in the value change template (line 15), there are two cases; case 1: If X is of NX category, and left child of X is VP (line 14). Case 2: if X is of VP category, right child of X is CP, and left child of X is VX (line 14). If the sub tree of AP does not meet any of the previous mentioned conditions, then AP will be stored in the process template list (line 17). The procedure allows an expert user input to the final template lists (lines 18-20), the user can modify, delete, add or exchange APs from any list if any AP is classified in the wrong list.

Procedure 8 shows the overall flow for computing the refinement map template for each AP in a timing requirement. A set of timing requirements in natural language form are fed as input to the procedure. Three templates lists are the output of the procedure as in procedure I, each list will contain set of APs based on the heuristic data from the parsed trees belong to input requirements. Three empty template lists are defined; projection template list (line 1), process template list (line 2), and value template list (line 3). A list for functional requirement's APs (AP_t-list) is initialized to null (line 4). Each requirement is input to the Synthesising Procedure for Timing Requirements (SPTR) (line 6) which comes up with formal specifications (explained in 3.2.5). A function called Get_AP-list is used to obtain the resulting AP-list from the SPTR into the AP_t-list (line 7). A function called Get_Sub-tree is applied to each entry (AP_t) in the AP_t-list, this function returns the sub tree that corresponds to the AP_t from Enju parsed tree (line 9). A function is called Head stores the head category of the sub tree in variable $X1$ (line 10). Check if $X1$ is of VP category, right child of $X1$ is NP, and left child of $X1$ is VX (line 11), then AP is added to the projection template list (line 12). Also if $X1$ is of VX category, right child of $X1$ is NP, and left child of $X1$ is VX (line 11), so AP is added to the projection template list too. For the AP to be stored in the

Table 3.1. List of abbreviations for Figures 3.6-3.15

Abbreviation	**Meaning**
AI	Active Infusion
CDTC	Change Drug Type and Concentration
DT	Data Type
HDT	Historical Data Type
UR	User Reminder
RTVR	Reservoir Time and Volume Recomputed
CRV	Current Reservoir Volume
HRV	Historical Reservoir Volume
IBO	Incomplete Bolus
INDV	Insulin Delivery
SPM	Suspension Mode
SYNC	Synchronization
INCAL	Insulin Calculations
REQ-FLAG	Request Flag
CALL-FUNCT	Call-Function for Calculation
SET	Settings
CLRS	Clear Settings
CHNS	Change Settings
RESS	Reset Settings
UCNF	User Confirmation
SETT	Setting the concentration
CHNC	Changing the Concentration
BG	Blood Glucose
TBG	Targeted Blood Glucose
INCR	Insulin to Carbohydrate ratio
CORF	Correction Factor
ACT	User Action
CNFI	Confirm Input
CHNI	Change Input
ELR	Event or Log Retrieving
EL	Event Logging
LR	Log Retrieving
ELRF	Event Logging or Logging Retrieving Failure
ELF	Event Logging Failure
LRF	Logging Retrieving Failure
ELF	Event Logging Failure
FWAR	Failure Warning
NBA	NO Basal delivery
NABA	No Alarm or Basal delivery
NA	No Alarm
AIRB	Air Bubbles
DLM	Delivery Mode
NDLM	Non-Delivery Mode
UIR	User Input Requested
RTD	Requested Time and Date
RDC	Requested Drug type and Concentration
RRDV	Requested Reloading Drug reservoir
UIP	User Input Provided
STD	Setting Time and Date
SDC	Setting Drug type and Concentration
RDV	Reloading Drug reservoir

value change template (line 15), there is only one case; If $X1$ is of NX category, right child of $X1$ is of ADJ category and left child of $X1$ is also NX (line 14). If the sub tree of AP does not meet any of the previous mentioned conditions, then it will be stored in the process template list (line 17). As in procedure I, this procedure allows an expert user input to the final template lists (lines 18-20), the user can modify, delete, add or exchange APs from any list if any AP is classified in the wrong list.

Procedure 8 Procedure for synthesizing Refinement Maps for timing requirements

Require: Set of Timing Requirements
1: Projection-Temp-list ← ∅ ;
2: Process-Temp-list ← ∅ ;
3: Value-Temp-list ← ∅ ;
4: AP_t-list ← ∅ ;
5: **for each** Req_t ∈ Timing Requirements **do**
6: $Apply_SPTR(Req_t)$;
7: AP_t-list ← Get_AP-list (SPTR);
8: **for each** $AP_t \in AP_t$-list **do**
9: Sub-tree ← Get_Sub-tree (AP_t);
10: $X1$ = Head(Sub-tree);
11: **if** [($X1$ = VP) ∧ (RightChild($X1$) = NP) ∧ (LeftChild($X1$) = VX)]∨ [($X1$ = VX) ∧ (RightChild($X1$) = NP) ∧ (LeftChild($X1$) = VX)] **then**
12: Projection-Temp-list ← Projection-Temp-list ∪ AP_t;
13: **else**
14: **if** [($X1$ = NX) ∧ (RightChild($X1$) = ADJ) ∧ (LeftChild($X1$) = NX)] **then**
15: Value-Temp-list ← Value-Temp-list ∪ AP_t;
16: **else**
17: Process-Temp-list ← Process-Temp-list ∪ AP_t;
18: Projection-Temp-list ← USR_IN(Projection-Temp-list);
19: Value-Temp-list ← USR_IN(Value-Temp-list);
20: Process-Temp-list ← USR_IN(Process-Temp-list); =0

3.6. Conclusion and Future Work

In this chapter, we have developed a process for refinement maps construction. Heuristics have been developed based on the output of the Enju parser to select a refinement map template for each atomic proposition. The key ideas of our approach for synthesising refinement maps are the following. The system requirement is fed as an input, then the previously proposed synthesising procedure of formal specification is applied on the input requirement, the heuristic data from the requirement's parse tree is used to select the suitable refinement map template which are either process template, projection template or changing value template. The development and testing of this process is part of future work.

3.7. References

[1] Eman M. Al-Qtiemat, Sudarshan K. Srinivasan, Zeyad A. Al-Odat, and Sana Shuja. "Refinement Maps for Insulin Pump Control Software Safety Verification". In: *The Eleventh International Conference on Advances in System Testing and Validation Lifecycle VALID 2019*. IARIA.

[2] Baowei Fei, Wan Sing Ng, Sunita Chauhan, and Chee Keong Kwoh. "The safety issues of medical robotics". In: *Reliability Engineering & System Safety* 73.2 (2001), pp. 183–192.

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

[4] SMK Quadri and Sheikh Umar Farooq. "Software Testing-Goals, Principles, and Limitations". In: *International Journal of Computer Applications* 6.9 (2010), pp. 7–10.

[5] Edward Miller and William E Howden. *Tutorial, software testing & validation techniques*. IEEE Computer Society Press, 1981.

[6] 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*. Ed. by Ahmed Bouajjani and Oded Maler. Vol. 5643. Lecture Notes in Computer Science. Springer, 2009, pp. 414–429. ISBN: 978-3-642-02657-7. DOI: 10.1007/978-3-642-02658-4_32. URL: https://doi.org/10.1007/978-3-642-02658-4%5C_32.

[7] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram 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*. Ed. by Eerke A. Boiten, John Derrick, and Graeme Smith. Vol. 2999. Lecture Notes in Computer Science. Springer, 2004, pp. 1–20. ISBN: 3-540-21377-5. DOI: `10.1007/978-3-540-24756-2_1`. URL: `https://doi.org/10.1007/978-3-540-24756-2%5C_1`.

[8] Karthikeyan Bhargavan et al. "Formal verification of smart contracts: Short paper". In: *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. ACM. 2016, pp. 91–96.

[9] 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.

[10] Panagiotis Manolios. "Mechanical Verification of Reactive Systems". last accessed: 2018-10-10. PhD thesis. University of Texas at Austin, Aug. 2001. URL: `http://www.ccs.neu.edu/home/pete/research/phd-dissertation.html`.

[11] Mohana Asha Latha Dubasi, Sudarshan K Srinivasan, and Vidura 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.

[12] Eman M Al-qtiemat, Sudarshan K Srinivasan, Mohana Asha Latha Dubasi, and Sana 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.

[13] Eman M. Al-Qtiemat, Sudarshan K. Srinivasan, Zeyad A. Al-Odat, and Sana Shuja. "Synthesis of Formal Specifications From Requirements for Refinement-based Real Time Object Code Verification". In: *International Journal on Advances in Internet Technology* 12 (Aug. 2019), pp. 95–107. ISSN: 1942-2652.

[14] Martıén Abadi and Leslie Lamport. "The existence of refinement mappings". In: *Theoretical Computer Science* 82.2 (1991), pp. 253–284.

[15] Mohana Asha Latha Dubasi, Sudarshan K Srinivasan, Sana Shuja, and Zeyad A Al-Odat. “Refinement Checker for Embedded Object Code Verification”. In: ().

[16] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. “Feature Specification and Refinement with State Transition Diagrams”. In: *arXiv preprint arXiv:1409.7232* (2014).

[17] Eman Rabiah and Boumediene Belkhouche. “Formal specification, refinement, and implementation of path planning”. In: *12th International Conference on Innovations in Information Technology (IIT)*. IEEE. 2016, pp. 1–6.

[18] Maria Spichkova. “Refinement-based verification of interactive real-time systems”. In: *Electronic Notes in Theoretical Computer Science* 214 (2008), pp. 131–157.

[19] Alvaro Miyazawa and Ana Cavalcanti. “Refinement-based verification of sequential implementations of Stateflow charts”. In: *arXiv preprint arXiv:1106.4094* (2011).

[20] Alessandro Cimatti and Stefano Tonetta. “Contracts-refinement proof system for component-based embedded systems”. In: *Science of computer programming* 97 (2015), pp. 333–348.

[21] David L Bibighaus. *Applying doubly labeled transition systems to the refinement paradox*. Tech. rep. Naval Postgraduate School Monterey CA, 2005.

[22] Yongwang Zhao, David Sanán, Fuyuan Zhang, and Yang Liu. “Formal specification and analysis of partitioning operating systems by integrating ontology and refinement”. In: *IEEE Transactions on Industrial Informatics* 12.4 (2016), pp. 1321–1331.

[23] Romain Geniet and Neeraj Kumar Singh. “Refinement Based Formal Development of Human-Machine Interface”. In: *Federation of International Conferences on Software Technologies: Applications and Foundations*. Springer. 2018, pp. 240–256.

[24] Yongwang Zhao, David Sanán, Fuyuan Zhang, and Yang Liu. “Refinement-based specification and security analysis of separation kernels”. In: *IEEE Transactions on Dependable and Secure Computing* 16.1 (2017), pp. 127–141.

[25] Thomas Fayolle, Marc Frappier, Régine Laleau, and Frédéric Gervais. “Formal refinement of extended state machines”. In: *arXiv preprint arXiv:1606.02016* (2016).

[26] Atif Mashkoor, Faqing Yang, and Jean-Pierre Jacquot. “Refinement-based validation of Event-B specifications”. In: *Software & Systems Modeling* 16.3 (2017), pp. 789–808.

[27] Graeme Smith and Kirsten Winter. "Relating trace refinement and linearizability". In: *Formal Aspects of Computing* 29.6 (2017), pp. 935–950.

[28] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. "Tractable refinement checking for concurrent objects". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2015, pp. 651–662.

[29] Aaron Joseph Turon and Mitchell Wand. "A separation logic for refining concurrent objects". In: *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2011, pp. 247–258.

[30] Hongjin Liang, Jan Hoffmann, Xinyu Feng, and Zhong Shao. "Characterizing progress properties of concurrent objects via contextual refinements". In: *International Conference on Concurrency Theory*. Springer. 2013, pp. 227–241.

[31] Yang Liu, Wei Chen, Yanhong A Liu, and Jun Sun. "Model checking linearizability via refinement". In: *International Symposium on Formal Methods*. Springer. 2009, pp. 321–337.

[32] Graeme Smith and John Derrick. "Refinement and verification of concurrent systems specified in Object-Z and CSP". In: *First IEEE international conference on Formal engineering methods*. IEEE. 1997, pp. 293–302.

[33] Emil Sekerinski. "Verification and refinement with fine-grained action-based concurrent objects". In: *Theoretical computer science* 331.2-3 (2005), pp. 429–455.

[34] Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. "Automated and modular refinement reasoning for concurrent programs". In: *International Conference on Computer Aided Verification*. Springer. 2015, pp. 449–465.

[35] Yang Liu et al. "Verifying linearizability via optimized refinement checking". In: *IEEE Transactions on Software Engineering* 39.7 (2012), pp. 1018–1039.

[36] Brijesh Dongol and Lindsay Groves. "Contextual trace refinement for concurrent objects: Safety and progress". In: *International Conference on Formal Engineering Methods*. Springer. 2016, pp. 261–278.

[37] Ivana Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. "Abstraction for concurrent objects". In: *Theoretical Computer Science* 411.51-52 (2010), pp. 4379–4398.

[38] Serdar Tasiran and Shaz Qadeer. "Runtime refinement checking of concurrent data structures". In: *Electronic notes in theoretical computer science* 113 (2005), pp. 163–179.

[39] Yi Zhang, Raoul Jetley, Paul L Jones, and Arnab Ray. "Generic safety requirements for developing safe insulin pump software". In: *Journal of diabetes science and technology* 5.6 (2011), pp. 1403–1419.

4. CONCLUSIONS AND FUTURE WORK

4.1. Conclusions

This thesis focused on formal verification techniques and their role in assuring system correctness. Formal verification has become the bedrock for ensuring software correctness when dealing with safety-critical systems. One of the biggest obstacles in applying formal techniques to commercial systems is the lack of formal specifications. Software requirements are expressed only in natural language.

In Chapter 2 we proposed novel methodologies for transforming natural language requirements into formal specifications. The key ideas of our approach for transforming functional requirements into transition systems are the following. The key ideas of our approach for transforming requirements into transition systems are the following. The extraction rules work on the parse tree to get an initial list of APs. 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 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. 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, but also allows for input from domain expert. The proposed methodology worked very well in practice for the GIIP requirements. All the TSs computed for the requirements satisfied their corresponding CTL properties.

The key ideas of our approach for transforming timing requirements into timed transition systems 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 TTSs, but also allows for input from domain expert. The proposed methodology worked very well in practice for the GIIP requirements. All the TTSs computed for the requirements satisfied their corresponding CTL properties.

In Chapter 3 two main objectives accomplished. First, we developed refinement maps for various safety properties concerning the software control operation of insulin pumps. We then identified possible generic templates for construction of refinement maps as a first step towards developing a process to construct refinement maps in an automated fashion.

Second, we developed a process for synthesising refinement maps. Heuristics were developed based on the output of the Enju parser to select a refinement map template for each atomic proposition. The key ideas of our approach are the following. Firstly, the system requirement is fed as an input. Secondly, the previously proposed synthesising procedure of formal specification is applied on the input requirement. Finally, the heuristic data from the requirement's parsed tree is utilized to select the suitable refinement map template. The identified refinement map templates are the process template, the projection template or changing value template. our work is considered to be a key solution for the complication of the construction of refinement maps, The advantage of this work is that it can be generalized and used for any critical device, this allows application of the refinement based verification to be expanded and more common to use because of its scalability and ability to assure systems correctness.

4.2. Directions for Future Research

The scope of research is still open in the area of ensuring the safety of critical devices. Formal verification techniques can be utilized in assuring the correctness of critical systems. In this thesis, first, we offered a methodology that synthesis formal specifications from natural language requirements. The new methodology is successfully applied on safety requirements of insulin pump. 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. In this section we offer some directions for future researches:

1. The synthesising procedure of formal specification is presented in Chapter 2. This novel approach is based on the proposed extraction rules (APER1,2, and 3 are for functional requirements, while APTCER is for timing requirements). This approach can be extended to include more atomic proposition extraction rules, also, an umbrella rule can be proposed that works on the resulting AP lists of all rules.

2. Our work is successfully applied on safety requirements of insulin pump, so it can be applied on other infusion pumps or even any critical device that has safety requirements. This work also can be build as a frame work where new options can be added or modified.

3. Automation procedure of refinement maps construction and some generic refinement map templates are presented in Chapter 3. As our work can be applied on any critical device that has safety requirements, more generic refinement map templates can be identified, in addition, the level of automation can be increased by improving the synthesis procedure.

APPENDIX A. INSULIN PUMP SAFETY REQUIREMENTS

All below stated safety requirements are taken from [1].

- **Requirement 1.1.1:** *"The pump shall suspend all active basal delivery and stop any active bolus during a pump prime or refill. It shall prohibit any insulin administration during the priming process and resume the suspended basal delivery, either a basal profile or a temporary basal, after the prime or refill is successfully completed."*

- **Requirement 1.1.1:** *"The pump shall suspend all active basal delivery and stop any active bolus during a pump prime or refill. It shall prohibit any insulin administration during the priming process and resume the suspended basal delivery, either a basal profile or a temporary basal, after the prime or refill is successfully completed."*

- **Requirement 1.1.3:** *"If the pump allows administering multiple types of insulin, changing drug types and concentrations shall stop any active infusion, remind the user to validate the basal profiles and related parameters, and force the reservoir time and volume to be recomputed."*

- **Requirement 1.1.3:** *"If the pump allows administering multiple types of insulin, changing drug types and concentrations shall stop any active infusion, remind the user to validate the basal profiles and related parameters, and force the reservoir time and volume to be recomputed."*

- **Requirement 1.2.4:** *"The pump shall allow the user to temporarily override the current basal delivery with a temporary basal without changing existing basal profiles, provided that no normal bolus or other temporary basal is in progress. The user shall be required to specify the duration and rate of the temporary basal being programmed."*
- **Requirement 1.2.6:** *"The pump shall start to administer a temporary basal immediately after the user confirms it, and resume the previously active basal profile after the temporary basal is finished."*

- **Requirement 1.2.6:** *"The pump shall start to administer a temporary basal immediately after the user confirms it, and resume the previously active basal profile after the temporary basal is finished."*

- **Requirement 1.2.7:** *"The pump shall allow the user to stop a temporary basal while it is being administered. When the user chooses to stop a temporary basal, the pump shall either resume the active basal profile prior to the temporary basal or require the user to activate a predefined basal profile."*

- **Requirement 1.3.5:** *"The pump shall not allow a normal bolus to start when another normal bolus is in progress. If the user requests a normal bolus when another normal bolus is in progress, the pump shall issue an alert and deny the request."*

- **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."*

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

- **Requirement 2.2.2:** *"Clearing, changing or resetting the pump settings shall require the user's confirmation."*

- **Requirement 2.2.3:** *"Setting and changing the concentration and activity duration of the currently loaded insulin shall require the user's confirmation."*

- **Requirement 3.1.1:** *"The pump and its accessories shall be designed to maintain a fail-safe state in the presence of a single fault condition that results in the inability of the pump to ensure the integrity of the pump's operation. When in a fail-safe state, the pump shall neither deliver insulin nor generate energy or substances that could affect the user's safety."*

- **Requirement 3.2.5:** *"When the user inputs a BG reading, target BG level, insulin-to-carbohydrate ratio, or correction factor that is out of manufacture- or user-defined ranges, the pump shall generate a warning and require the user to confirm or change the input."*

- **Requirement 3.2.7:** *"The pump shall issue a warning whenever there is a failure in event logging or log retrieving."*

- **Timing Requirement 1.2.8:** *"If the currently activated basal profile or the currently ongoing temporary basal has been paused for more than x minutes, it shall signal an audible alarm every y minutes up to z hours."*

- **Timing 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 L are detected, and all insulin administrations shall be stopped."*

- **Timing Requirement 1.8.4:** *"If the pump has been put in a non-delivery mode for more than x minutes, it shall signal an audible alarm for every x minutes up to y hours."*

- **Timing Requirement 2.2.1:** *"If the pump is in a state in which user input is required, e.g., setting time and date, setting drug type, and concentration after reloading the drug reservoir, the pump shall issue periodic alerts/indications every x minutes until the required input is provided."*

APPENDIX B. PARSED TREES OF INSULIN PUMP SAFETY REQUIREMENTS

This appendix illustrates the resulting parsed trees from applying Enju parser on insulin pump safety requirements. Enju 2.4 online demo has been used to get the trees, Table[1] B.1 shows some symbols of syntactic categories used in forming the trees.

Table B.1. The main syntactic categories used by Enju trees

ADJ	Adjective
ADV	Adverb
CONJ	Coordination conjunction
C	Complementizer
D	Determiner
N	Noun
P	Preposition
SC	Subordination conjunction
V	Verb

[1]This table is taken from the main web page of Enju "https://mynlp.is.s.u-tokyo.ac.jp/enju/".

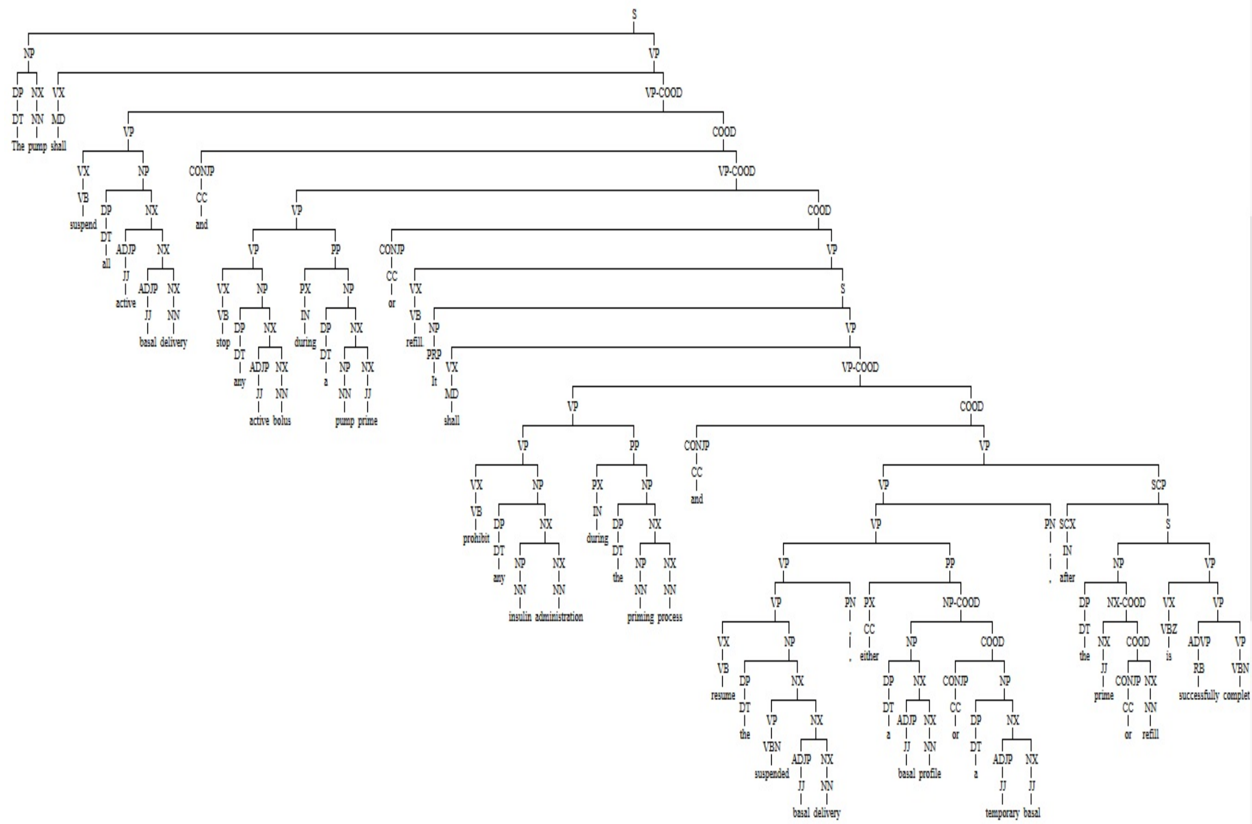


Figure B.1. Enju parsed tree for requirement 1.1.1.

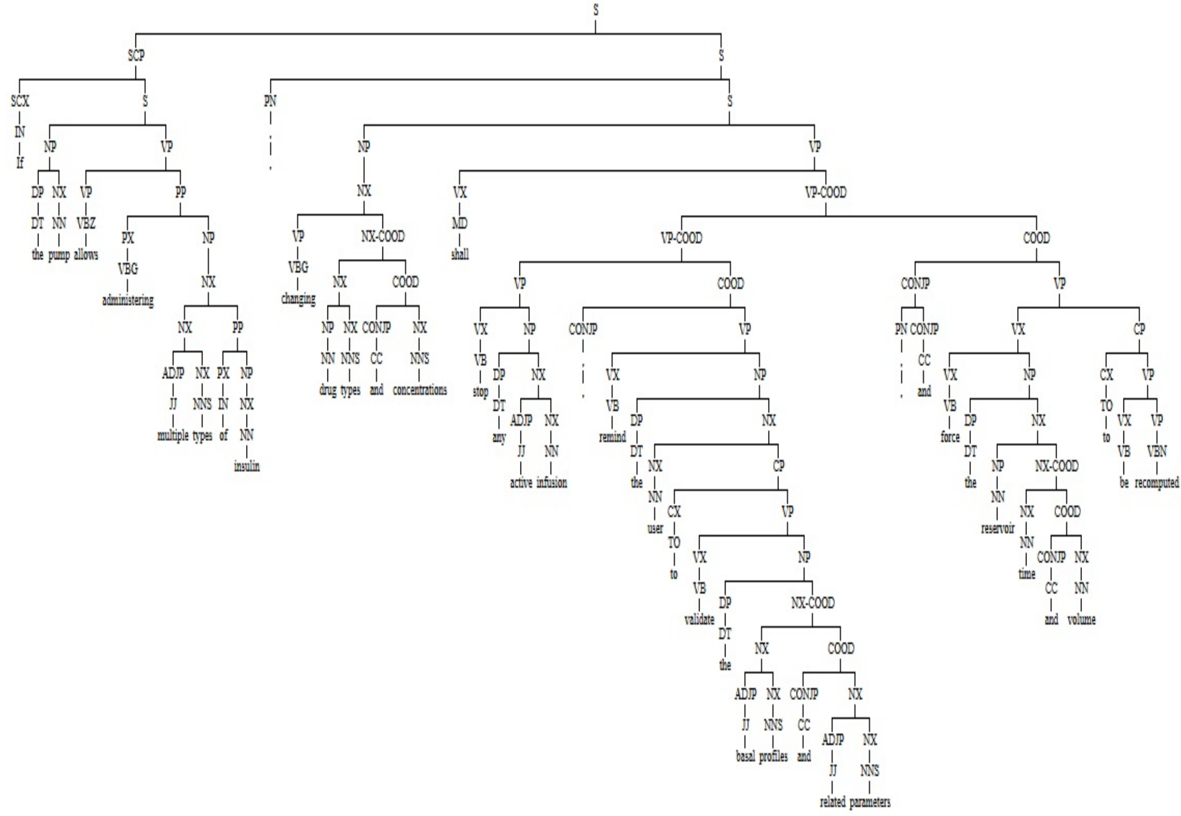


Figure B.2. Enju parsed tree for requirement 1.1.3.

Figure B.3. Enju parsed tree for requirement 1.2.4.

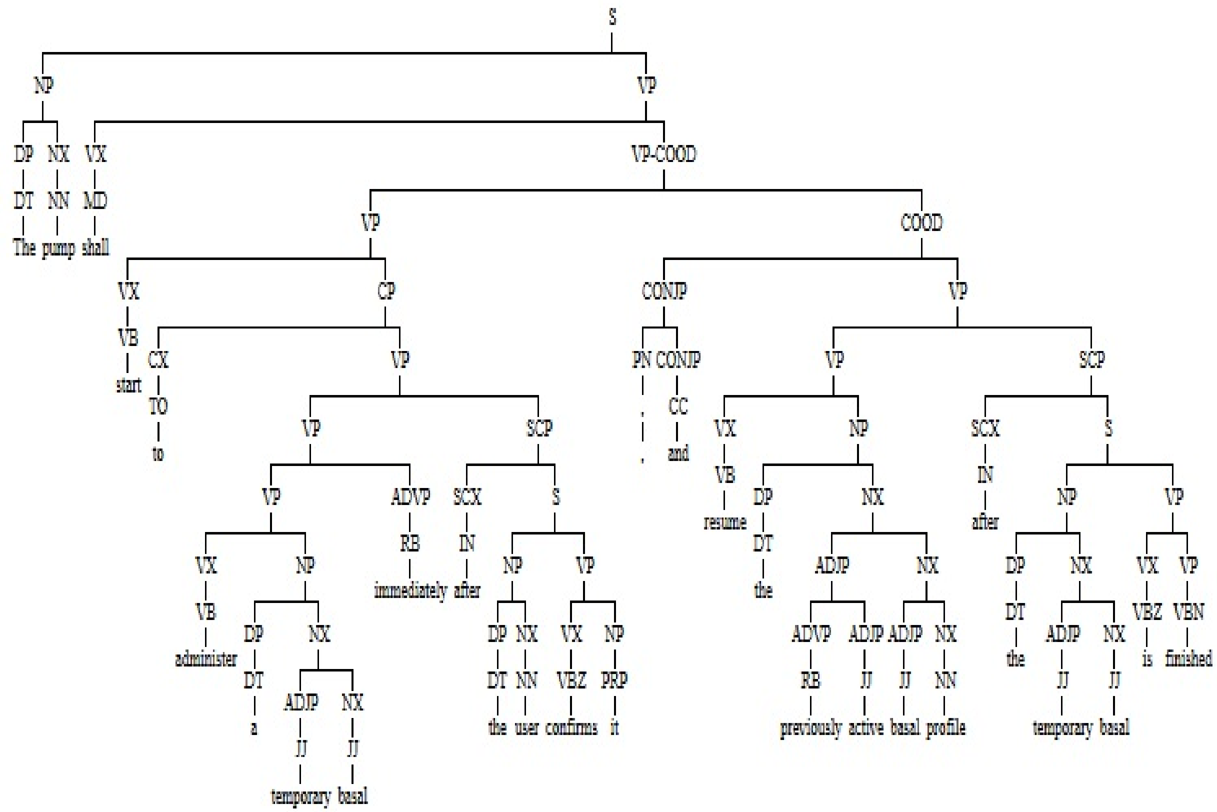


Figure B.4. Enju parsed tree for requirement 1.2.6.

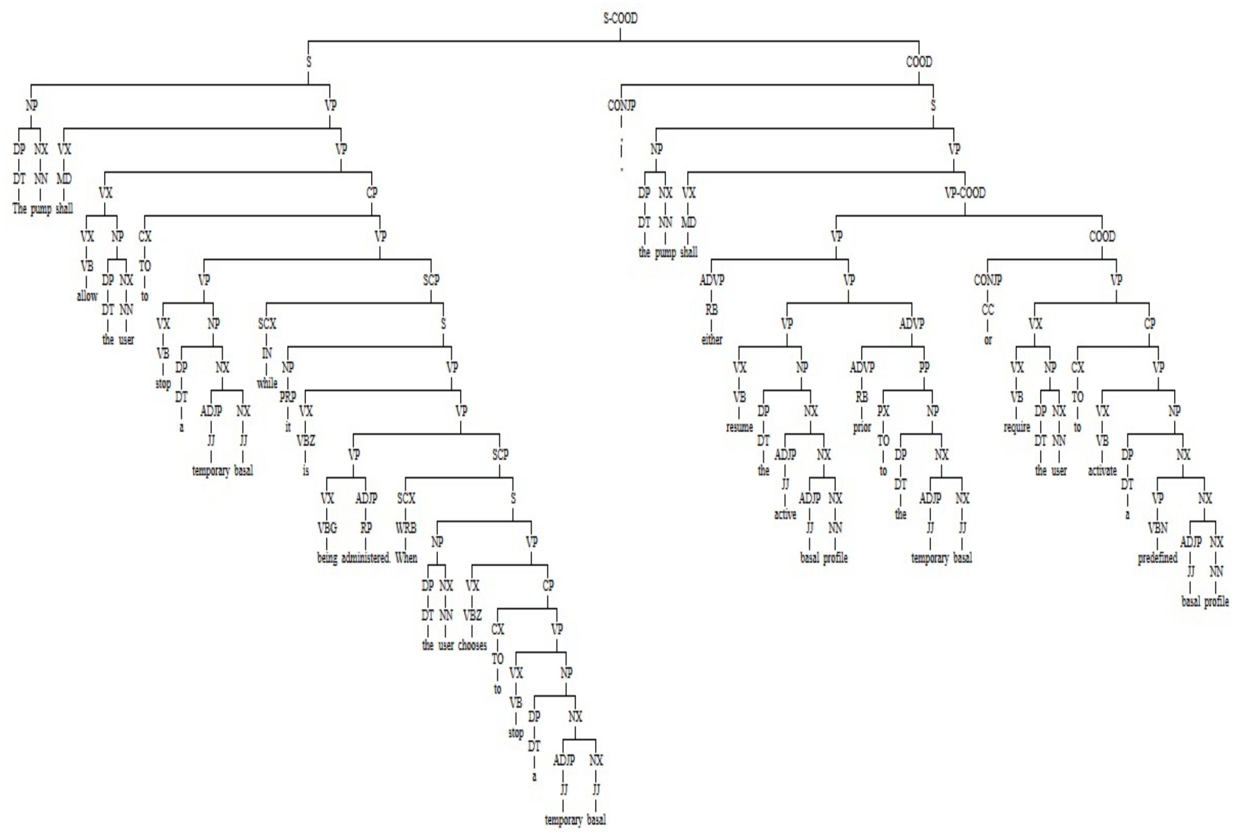


Figure B.5. Enju parsed tree for requirement 1.2.7.

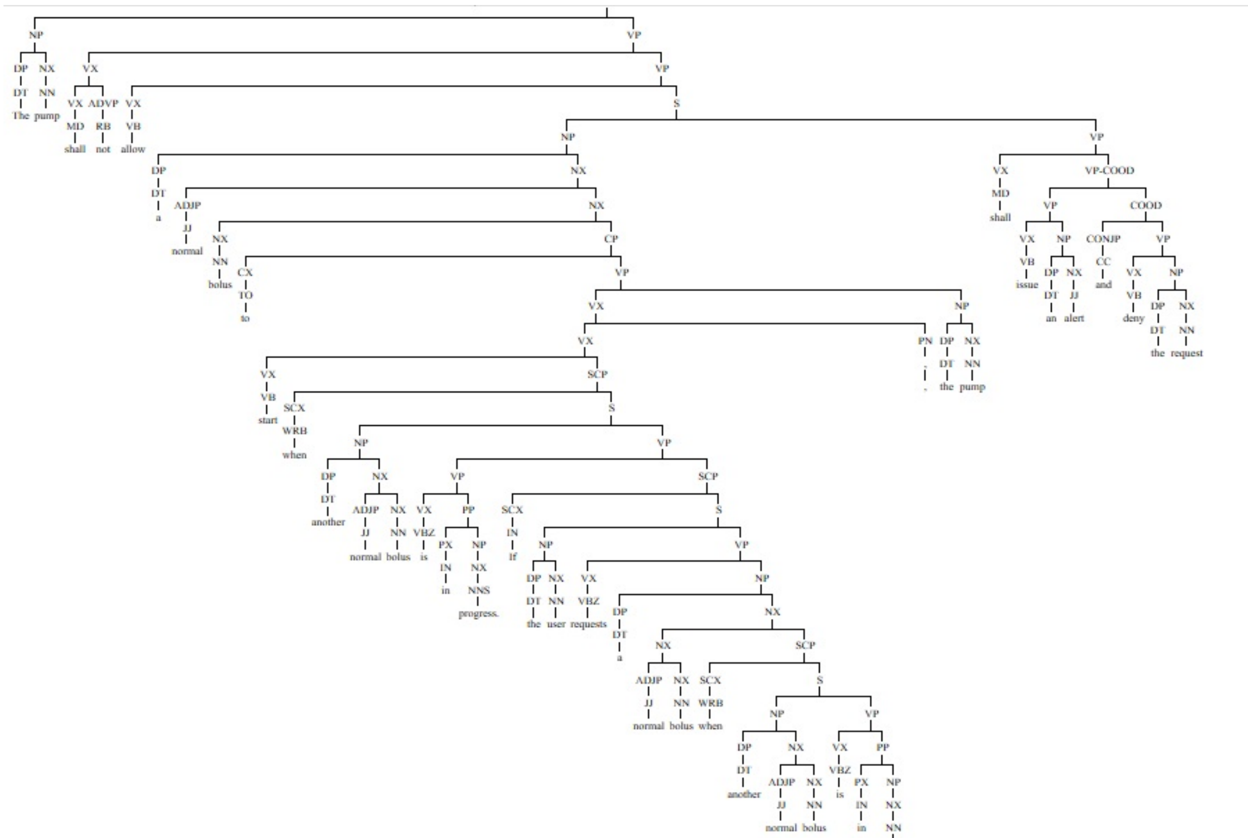


Figure B.6. Enju parsed tree for requirement 1.3.5.

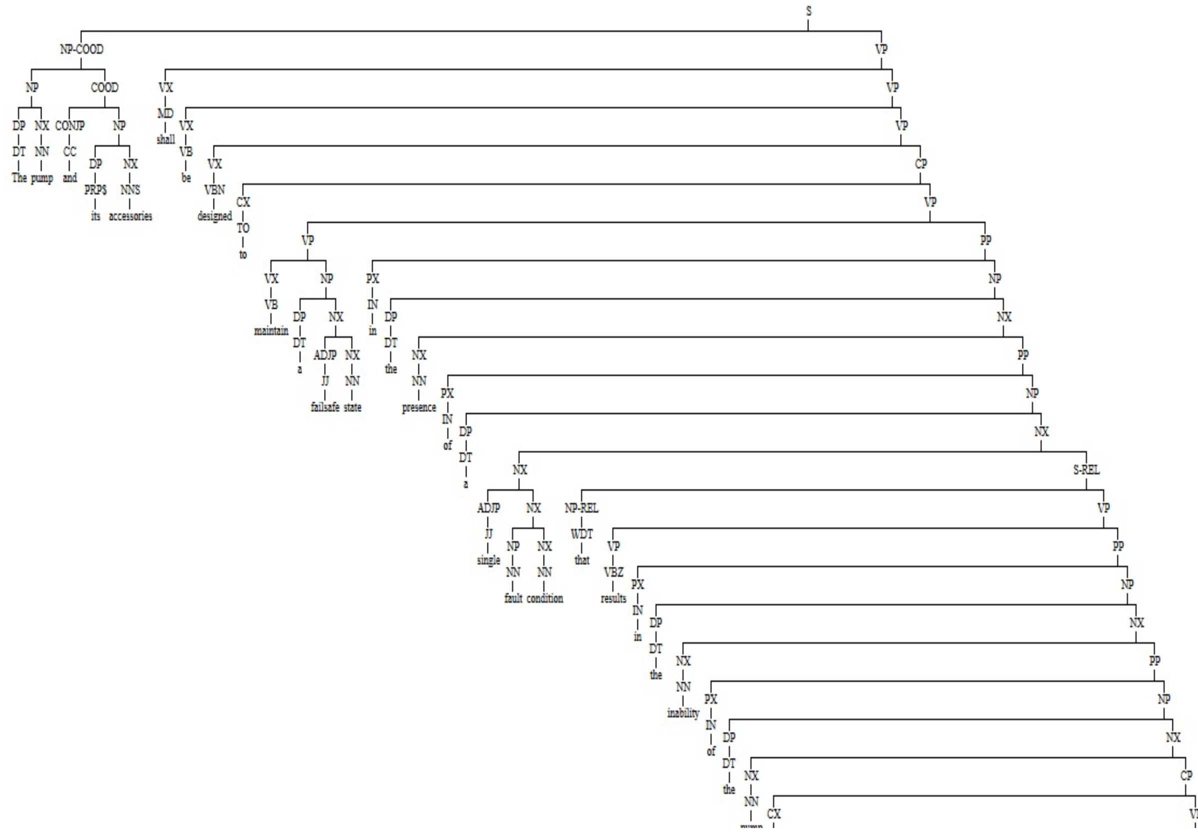


Figure B.7. Enju parsed tree for requirement 3.1.1 (part one).

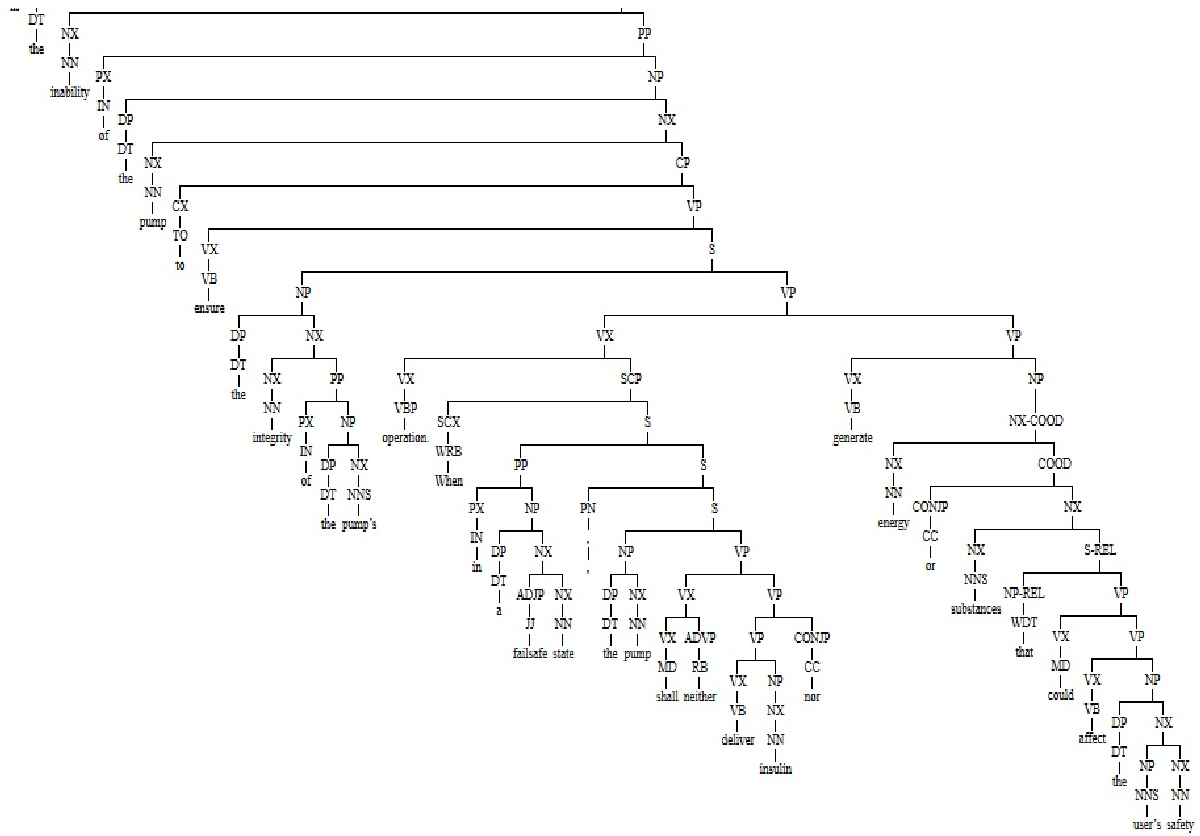


Figure B.8. Enju parsed tree for requirement 3.1.1 (part two).

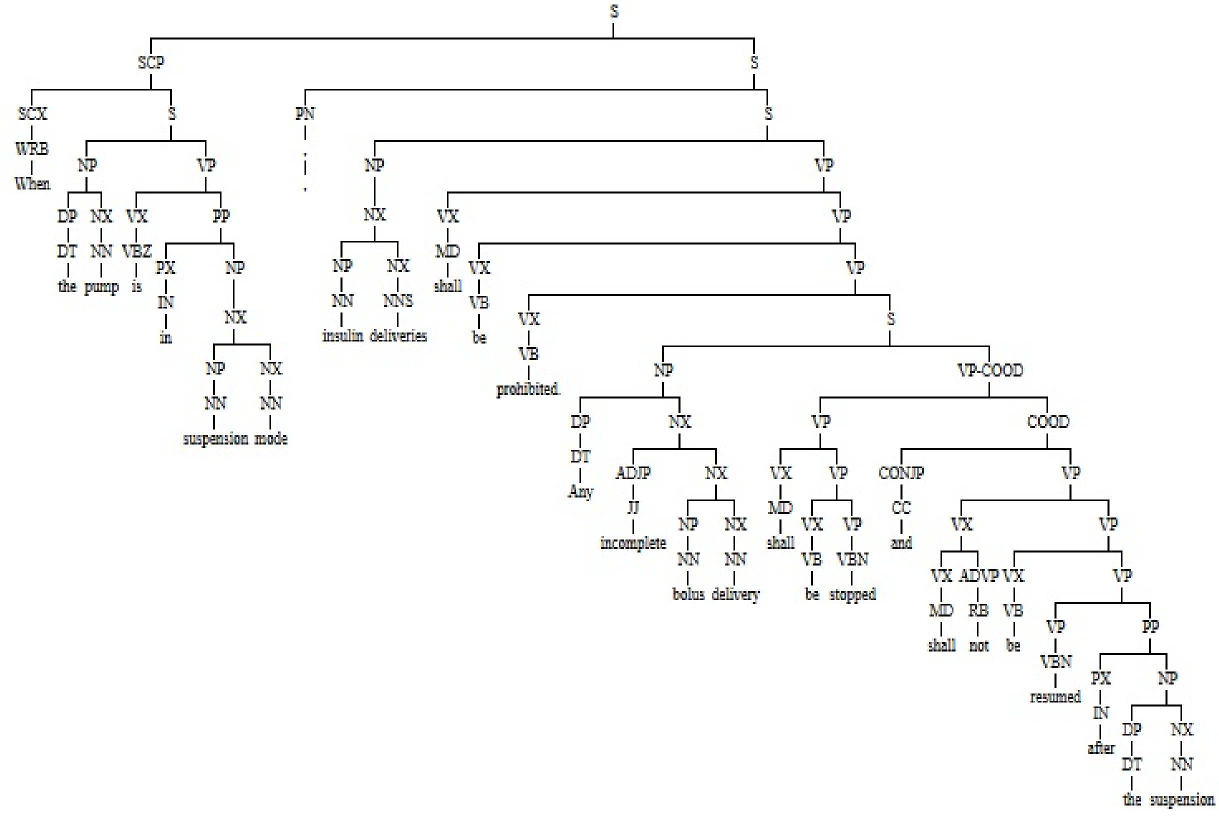


Figure B.9. Enju parsed tree for requirement 1.8.2.

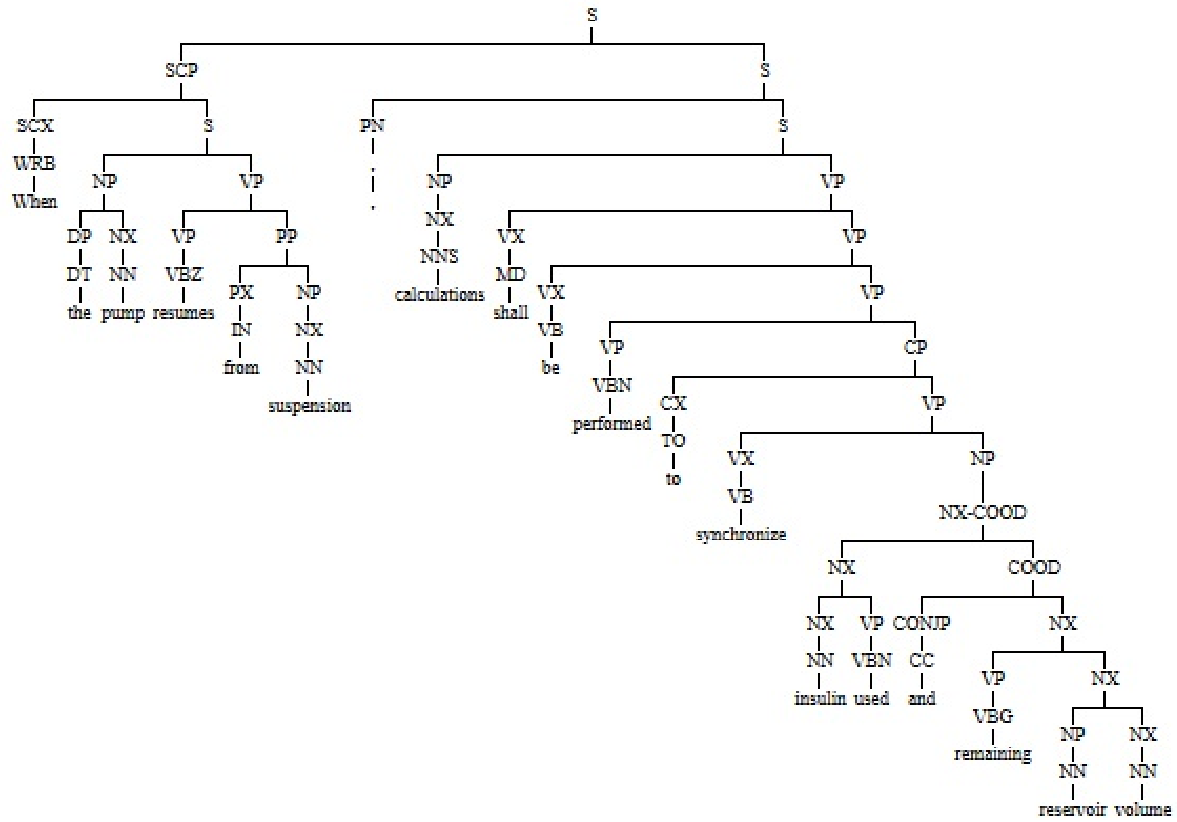


Figure B.10. Enju parsed tree for requirement 1.8.5.

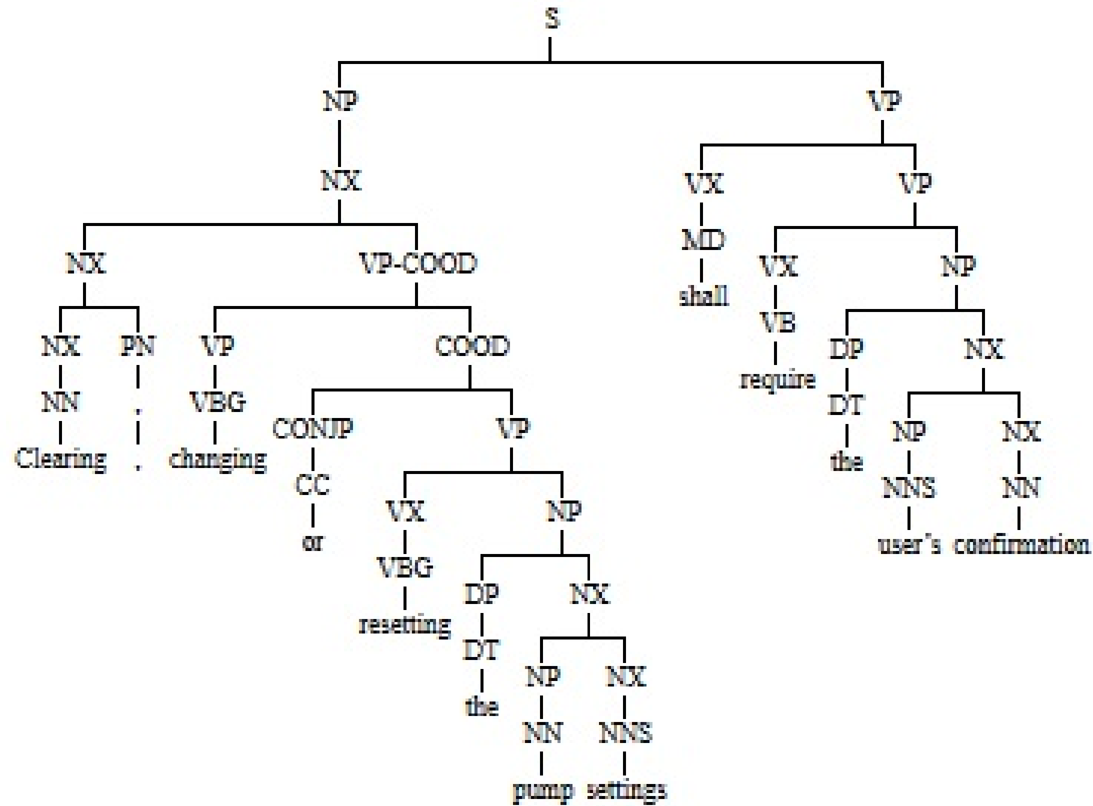


Figure B.11. Enju parsed tree for requirement 2.2.2.

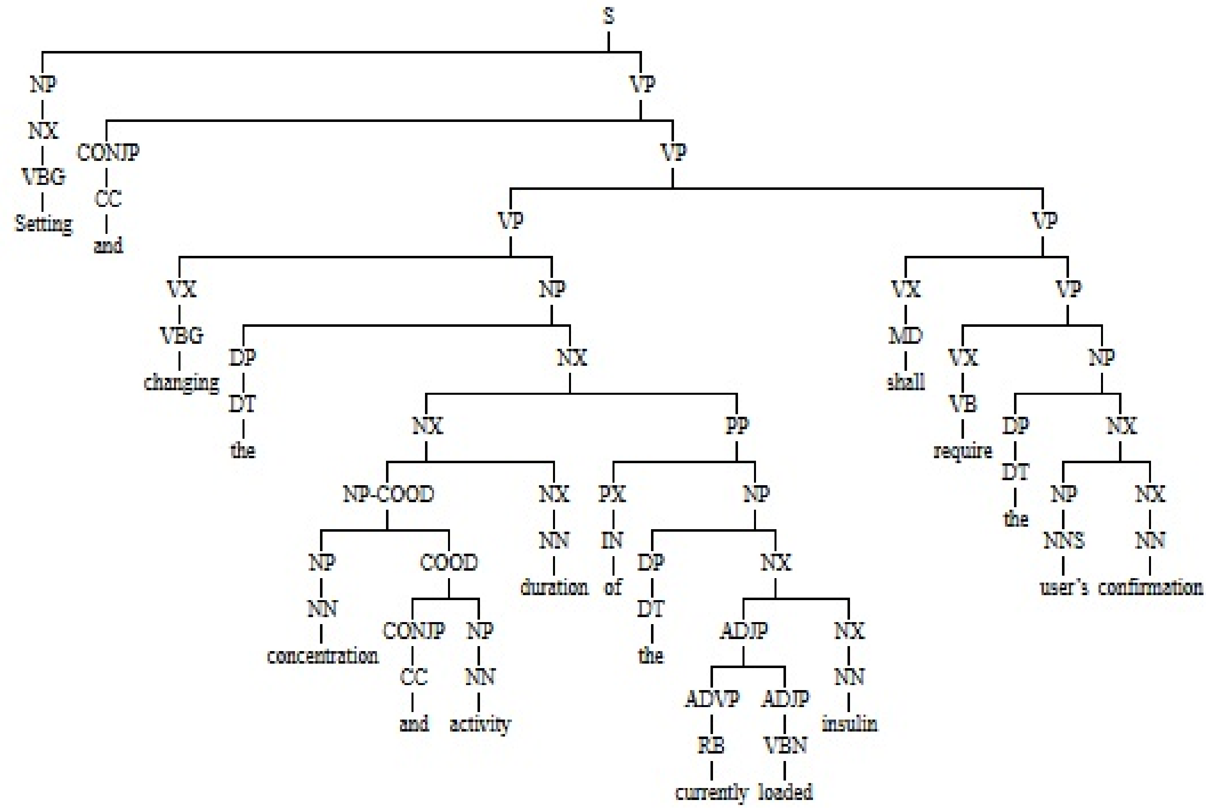


Figure B.12. Enju parsed tree for requirement 2.2.3.

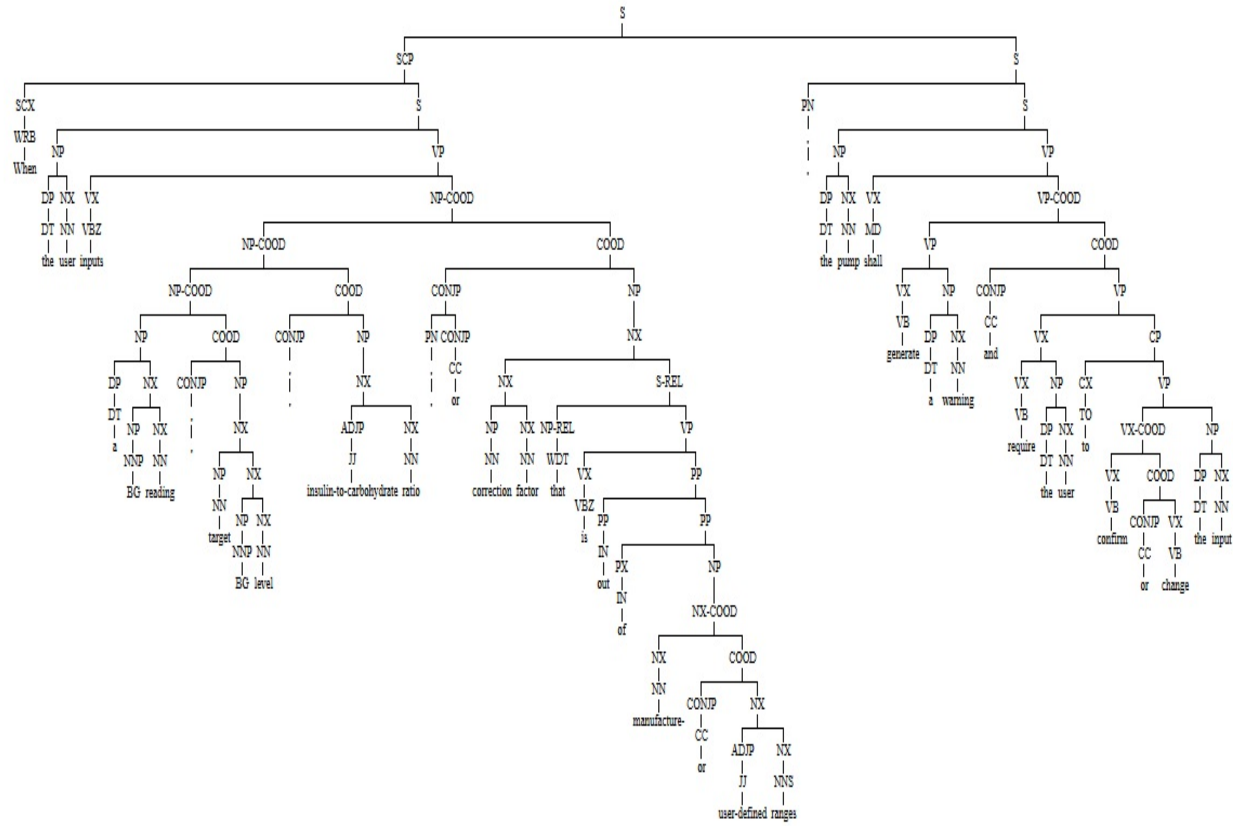


Figure B.13. Enju parsed tree for requirement 3.2.5.

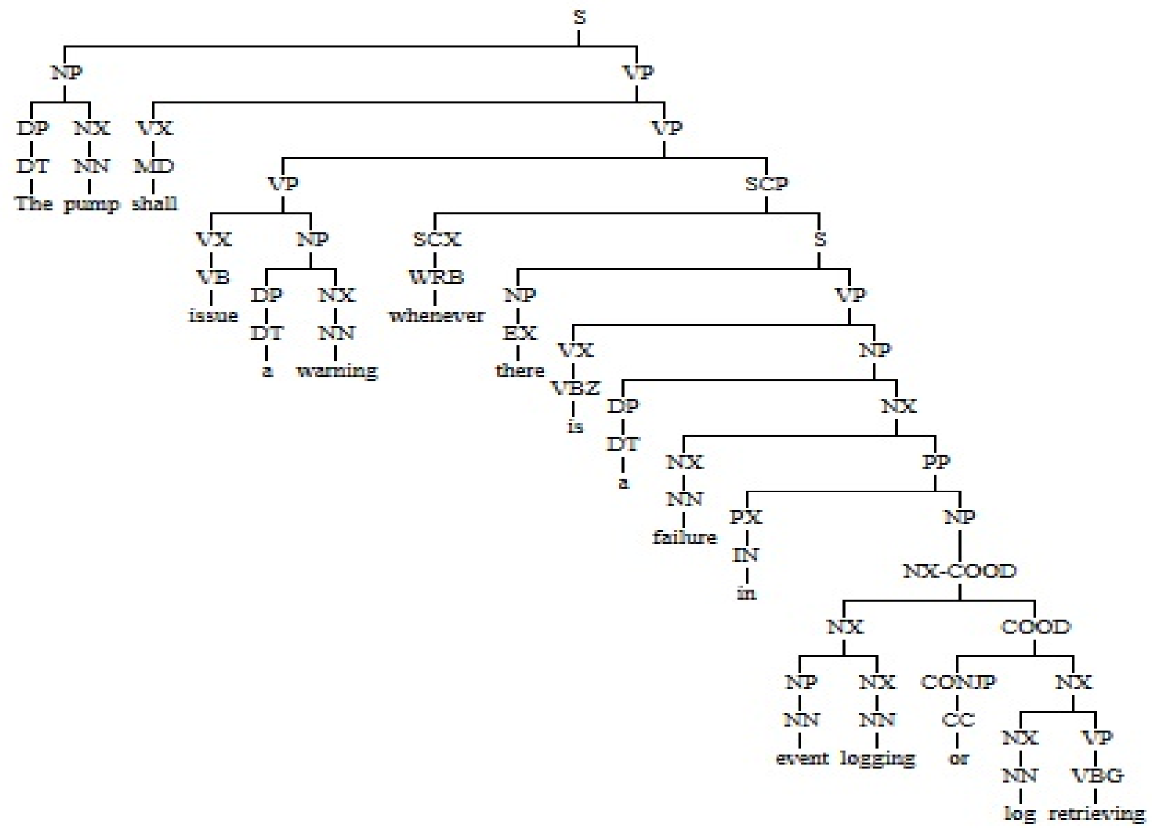


Figure B.14. Enju parsed tree for requirement 3.2.7.

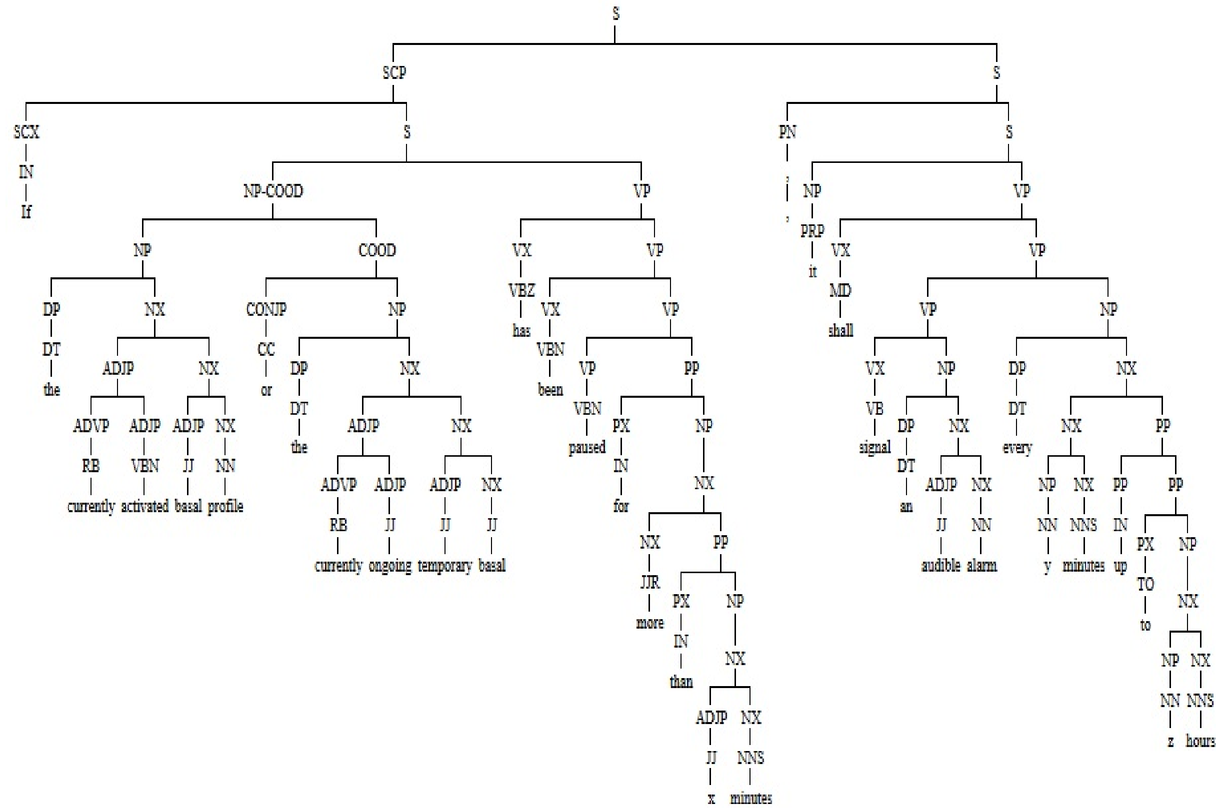


Figure B.15. Enju parsed tree for requirement 1.2.8.

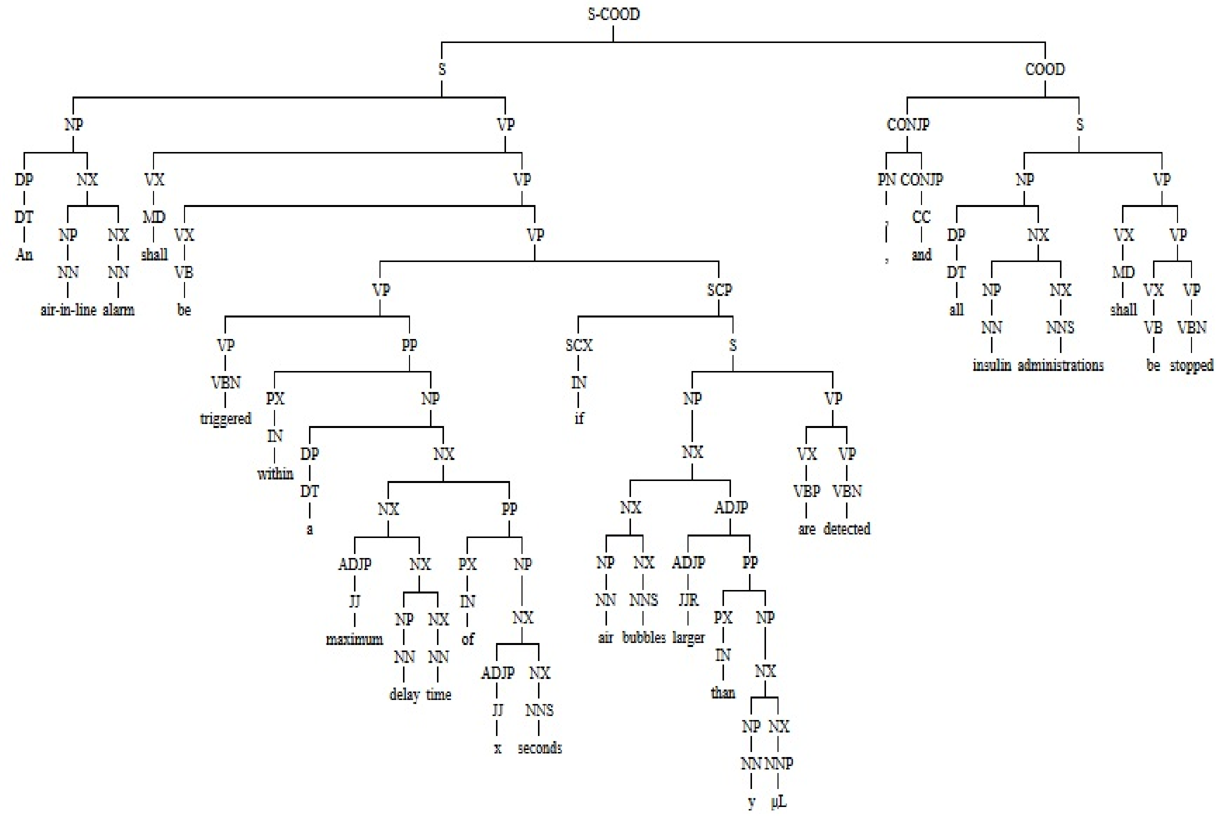


Figure B.16. Enju parsed tree for requirement 1.6.1.

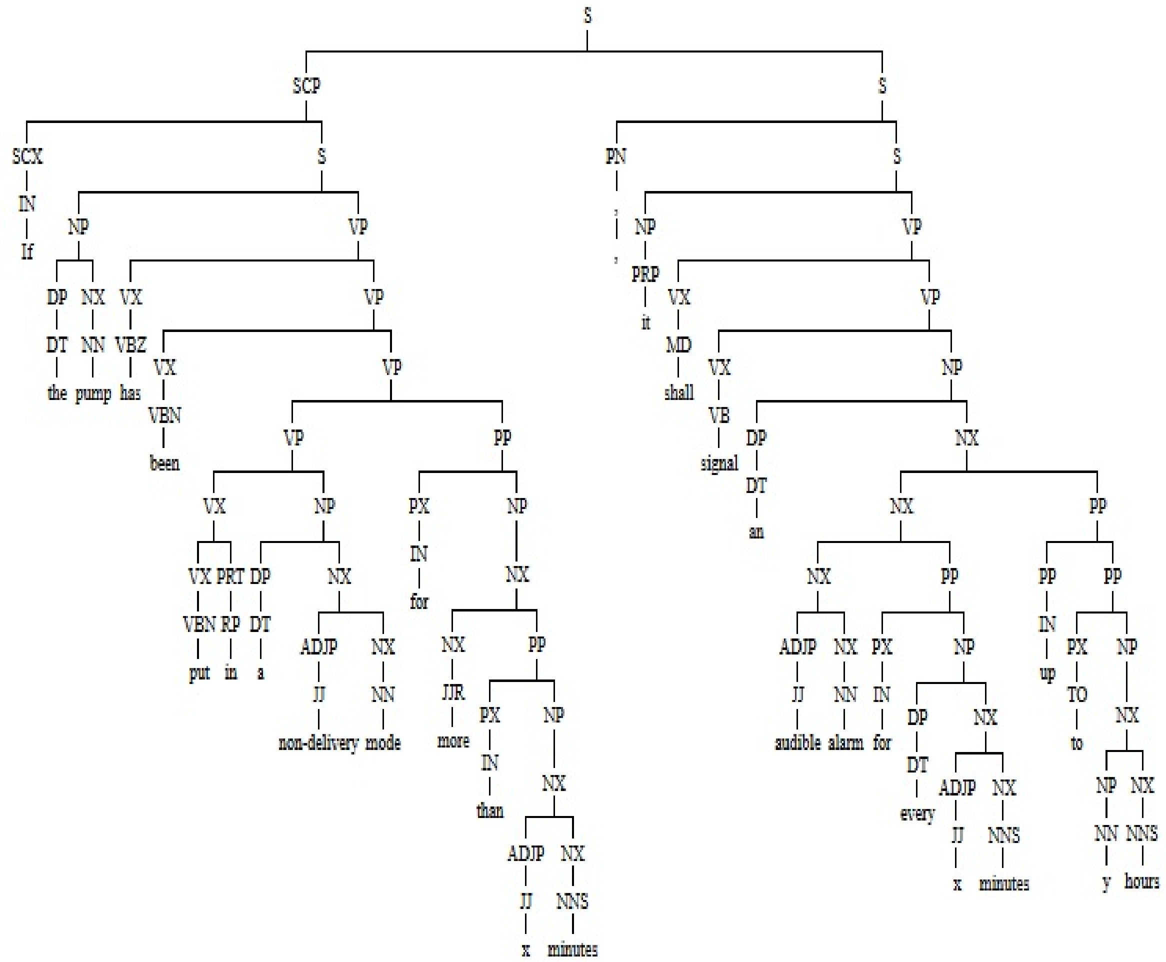


Figure B.17. Enju parsed tree for requirement 1.8.4.

Figure B.18. Enju parsed tree for requirement 2.2.1.

APPENDIX C. LIST OF PUBLICATIONS

[1] **Eman M. Al-Qtiemat**, Sudarshan K Srinivasan, Zeyad A. Al-Odat, Mohana Asha Latha Dubasi, and Sana Shuja. “Synthesis of Formal Specifications From Requirements for Refinement-based Real Time Object Code Verification”. In: *International Journal On Advances in Security* 12.1 & 2 (2019).

[2] **Eman M. Al-Qtiemat**, Sudarshan K Srinivasan, Mohana Asha Latha Dubasi, and Sana 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.

[3] **Eman M. Al-Qtiemat**, Sudarshan K. Srinivasan, Zeyad A. Al-Odat, and Sana Shuja. “Refinement Maps for Insulin Pump Control Software Safety Verification”. In: *The Eleventh International Conference on Advances in System Testing and Validation Lifecycle VALID 2019*. IARIA **(Accepted for publication)**.

[4] **Eman M. Al-Qtiemat**, Sudarshan K. Srinivasan, Zeyad A. Al-Odat, and Sana Shuja. “Synthesis of Refinement Maps for Real-Time Object Code Verification”. In: *International Journal On Advances in Life Sciences*. IARIA **(Submitted on 1st of March 2020)**.

[5] Zeyad A. Al-Odat, **Eman M. Al-Qtiemat**, and Samee Khan. “A Big Data Storage Scheme Based on Distributed Storage Locations and Multiple Authorizations”. In: *2019 IEEE 5th International Conference on Big Data Security and Privacy (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing, (HPSC) and IEEE International Conference on Intelligent Data and Security*. Vol. 5. 978-1-7281-0006-7/19. 2019, pp. 13–18.

[6] Zeyad A. Al-Odat, Sudarshan K. Srinivasan, Sana Shuja, **Eman M. Al-Qtiemat**, and Mohana Asha Latha Dubasi. “IoT-Based Secure Embedded Scheme for Insulin Pump Data Acquisition and Monitoring”. In: *CYBER 2018, The Third International Conference on Cyber-Technologies and Cyber-Systems*. Vol. 3. Cyper-2018. thinkmind.org. 2018, pp. 90–93.

[7] Zeyad A. Al-Odat, Sudarshan K. Srinivasan, **Eman M. Al-Qtiemat**, and Sana Shuja. "A Reliable IoT-Based Embedded Health Care System for Diabetic Patients". In: *International Journal On Advances in Internet Technology* 12.1&2 (2019).

[8] Zeyad A. Al-Odat, Sudarshan K Srinivasan, **Eman M. Al-Qtiemat**, and Sana Shuha. "A Secure Storage Scheme for Healthcare Data Over the Cloud Based on Multiple Authorizations". In: *The Fourth International Conference on Cyber-Technologies and Cyber-Systems (Cyber 2019)*. IARIA. 2019.

[9] Zeyad A. Al-Odat, **Eman M. Al-Qtiemat**, and Samee U. Khan. "An Efficient Lightweight Cryptography Hash Function for Big Data and IoT Applications". In: *29th Annual IEEE STC 2020 Software Technology Conference*. IEEE **(Submitted on 20th of March 2020)**.