Generation of Formal Specifications for Avionic Software Systems

Master Thesis

Submitted in Fulfillment of the Requirements for the Academic Degree M.Sc. Automotive Software Engineering

Dept. of Computer Science
Chair of Computer Engineering

Submitted by: Pranav Gulati
Student ID: 481911
Date: September 26, 2019

Supervising tutor: Dipl.-Math. (Univ.) Marina Reich
Examiner: Prof. Dr. Wolfram Hardt
Abstract

Development of software for electronic systems in the aviation industry is strongly regulated by pre-defined standards. The aviation industry spends significant costs of development in ensuring flight safety and showing conformance to these standards. Some safety requirements can be satisfied by performing formal verification. Formal verification is seen as a way to reduce costs of showing conformance of software with the requirements or formal specifications. Therefore, the correctness of formal specifications is critical. Writing formal specifications is at least as difficult as developing software [36]. This work proposes an approach to generate formal specifications from example data. This example data illustrates the natural language requirements and represents the ground truth about the system. This work eases the task of an engineer who has to write formal specifications by allowing the engineer to specify the example data instead. The use of a relationship model and a marking syntax and semantics are proposed that make the creation of formal specifications goal oriented. The evaluation of the approach shows that the proposed syntax and semantics capture more information than is strictly needed to generate formal specifications. The relationship model reduces the computational load and only produces formal specifications that are interesting for the engineer.

Keywords: Formal Specification, Example data, Avionic Software Systems, First Order Logic, Kripke Structures
Acknowledgments

I would like to take this opportunity to thank all the people who have contributed in some way or the other to the success of this work. I would like to thank my Supervising tutor Dipl.-Math. (Univ.) Marina Reich at Airbus Defence and Space. Her support throughout the thesis has been invaluable. She made me aware of the real problems of the industry and helped me gain valuable insights into the domain of formal verification. I would also like to thank Dr. Andreas Schweiger from Airbus Defence and Space for reviewing the thesis and providing valuable feedback.

Lastly I would like to thank my family and friends for believing in me and supporting me through this journey.
# Contents

<table>
<thead>
<tr>
<th>Contents</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>7</td>
</tr>
<tr>
<td>List of Tables</td>
<td>9</td>
</tr>
<tr>
<td>List of Abbreviations</td>
<td>10</td>
</tr>
</tbody>
</table>

## 1 Introduction

1.1 Motivation                                  | 13 |
1.2 Scope and Objective                         | 14 |
1.3 Outline                                     | 15 |

## 2 Background

2.1 Avionic Software Systems                    | 16 |
2.2 Development Process                         | 18 |
2.2.1 V-Model                                   | 19 |
2.2.2 Regulations                               | 21 |
2.3 Logics: Propositional and First Order       | 24 |
2.3.1 Propositional Logic                       | 24 |
2.3.2 First Order Logic                         | 25 |
2.4 Kripke Structure                             | 28 |
2.5 Linear Temporal Logic                       | 29 |
2.6 Formal Specification and Verification       | 30 |
2.6.1 What is Formal Specification and Verification | 30 |
2.6.2 Model Checking                            | 31 |
2.6.3 Theorem Proving                           | 32 |
2.6.4 Need of Formal Verification               | 33 |

## 3 Related Work

3.1 Improving Quality of Requirements           | 35 |
3.2 Formalizing Requirements                    | 36 |
3.3 Software Development Strategies             | 38 |
3.3.1 Test Driven Development                  | 39 |
3.3.2 Behavior Driven Development               | 39 |
3.3.3 Specification-by-example                  | 40 |
CONTENTS

3.4 Generation of Formal Specifications from Example Data .................. 41

4 Running Example: Queue Timeout ........................................... 43
  4.1 Requirements ................................................................. 43
  4.2 Available Information .................................................... 46
  4.3 Formal Specifications ..................................................... 46

5 Automated Property Formation .................................................. 48
  5.1 System Architecture ...................................................... 49
  5.2 Inputs ......................................................................... 50
    5.2.1 Background Knowledge .............................................. 51
    5.2.2 Marked Scenario Data ............................................. 53
    5.2.3 Relationship Model ............................................... 55
  5.3 Creation of candidate properties ....................................... 57
  5.4 Marking ................................................................. 58
    5.4.1 Marking Syntax ..................................................... 59
  5.5 Algorithm for First Order Logic (FOL) formula creation ............... 62
    5.5.1 Marking Generalization .......................................... 63
    5.5.2 Quantifier Assignment ............................................ 64
  5.6 Algorithm for Kripke Structure formation ................................ 66
    5.6.1 Kripke Structure from Marking Semantics ....................... 66
  5.7 Outputs ................................................................. 68
    5.7.1 First Order Logic (FOL) Formulae ................................ 68
    5.7.2 Kripke Structures .................................................. 69

6 Implementation Details .......................................................... 71
  6.1 Parsing the Inputs ....................................................... 71
    6.1.1 Background Knowledge Parsing .................................... 71
    6.1.2 Relationship Model Parsing ...................................... 72
    6.1.3 Scenario Data File Parsing ...................................... 73
  6.2 Software Implementation .................................................. 73
    6.2.1 Architecture ........................................................ 74
  6.3 Visualizing the Outputs .................................................. 76

7 Evaluation ................................................................. 78
  7.1 Performance Evaluation .................................................. 78
    7.1.1 Effect of Number of Variables ................................... 79
    7.1.2 Effect of Number of Predicates .................................. 80
    7.1.3 Effect of Arity of Predicates .................................... 82
    7.1.4 Effect of Number of Relationship Models ....................... 83
    7.1.5 Effect of Number of Data Rows ................................... 85
  7.2 Feasibility of FOL ....................................................... 86
  7.3 Feasibility of Kripke Structure ........................................ 88
## List of Figures

2.1 V Model according to ARP4754A [78] ................................. 20
2.2 Propositional Logic illustrated ........................................ 25
2.3 First Order Logic illustrated ........................................ 26
2.4 Comparison between Propositional Logic and FOL .................. 27
  2.4a Propositional Logic comparison example ....................... 27
  2.4b First Order Logic comparison example ....................... 27
2.5 Visual Example of a Kripke Structure [40] .......................... 28

3.1 Specification by example ............................................. 41
  3.1a Door can be locked only when closed .......................... 41
  3.1b Gherkin syntax specification ................................. 41

4.1 A Kripke Structure for the running example ........................ 47

5.1 Structure of the proposed system .................................. 50
5.2 Data flow diagram of the proposed system ......................... 51
5.3 Example: Background knowledge ................................... 52
5.4 Example: Marked Scenario Data .................................... 54
5.5 Relationship model template ........................................ 55
5.6 Example: Relationship models ..................................... 57
5.7 Backus-Naur Form (BNF) grammar for marking syntax shown in GOLD parser builder .................................................. 59
5.8 Argument tree for a predicate with 3 arguments .................. 63
5.9 Informal representation of propGroup + stepVal generalization output .................................................. 64
5.10 Example: Generated first order logic formula .................... 65
5.11 Illustrating Marking semantics with generated Kripke structure ................................. 68

6.1 BNF grammar for parsing the background knowledge file .......... 72
6.2 Grammar for relationship model file parsing ....................... 73
6.3 Scenario data file shown as a Comma Separated Values (CSV) .... 74
6.4 Class diagram for the complete software .......................... 75
6.5 Activity diagram for the system ................................... 77

7.1 Effect of number of variables ..................................... 80
  7.1a Observed Data .................................................. 80
  7.1b Plotted Data .................................................. 80
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2</td>
<td>Effect of number of predicates of arity 2</td>
<td>82</td>
</tr>
<tr>
<td>7.2a</td>
<td>Observed Data</td>
<td>82</td>
</tr>
<tr>
<td>7.2b</td>
<td>Plotted Data</td>
<td>82</td>
</tr>
<tr>
<td>7.3</td>
<td>Effect of number of predicates of arity 3</td>
<td>82</td>
</tr>
<tr>
<td>7.3a</td>
<td>Observed Data</td>
<td>82</td>
</tr>
<tr>
<td>7.3b</td>
<td>Plotted Data</td>
<td>82</td>
</tr>
<tr>
<td>7.4</td>
<td>Effect of arity of predicates</td>
<td>83</td>
</tr>
<tr>
<td>7.4a</td>
<td>Observed Data</td>
<td>83</td>
</tr>
<tr>
<td>7.4b</td>
<td>Plotted Data</td>
<td>83</td>
</tr>
<tr>
<td>7.5</td>
<td>Effect of number of relationship models</td>
<td>84</td>
</tr>
<tr>
<td>7.5a</td>
<td>Observed Data</td>
<td>84</td>
</tr>
<tr>
<td>7.5b</td>
<td>Plotted Data</td>
<td>84</td>
</tr>
<tr>
<td>7.6</td>
<td>Effect of number of data rows</td>
<td>86</td>
</tr>
<tr>
<td>7.6a</td>
<td>Observed Data</td>
<td>86</td>
</tr>
<tr>
<td>7.6b</td>
<td>Plotted Data</td>
<td>86</td>
</tr>
<tr>
<td>7.7</td>
<td>Kripke structure elements illustrated on scenario data table</td>
<td>88</td>
</tr>
<tr>
<td>7.8</td>
<td>Generated first order logic formulae for the queue timeout example</td>
<td>92</td>
</tr>
<tr>
<td>7.9</td>
<td>Generated Kripke structure illustrated with properties</td>
<td>93</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Types of software in Aircraft systems ........................................... 17
4.1 Example data to illustrate requirements ........................................... 45
7.1 Evaluating 2 different FOL creation algorithms ................................. 87
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LTL</td>
<td>Linear Temporal Logic</td>
<td>CSV</td>
<td>Comma Separated Values</td>
</tr>
<tr>
<td>CTL</td>
<td>Computation Tree Logic</td>
<td>POJO</td>
<td>Plain Old Java Object</td>
</tr>
<tr>
<td>FOL</td>
<td>First Order Logic</td>
<td>PNG</td>
<td>Portable Network Graphics</td>
</tr>
<tr>
<td>HOL</td>
<td>Higher Order Logic</td>
<td>MFD</td>
<td>Multi Function Display</td>
</tr>
<tr>
<td>TDD</td>
<td>Test Driven Development</td>
<td>VerITAS</td>
<td>Verified Indoor Tactical Avionics Squad</td>
</tr>
<tr>
<td>ATDD</td>
<td>Acceptance Test Driven Development</td>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>BDD</td>
<td>Behavior Driven Development</td>
<td>EUROCAE</td>
<td>European Organisation for Civil Aviation Equipment</td>
</tr>
<tr>
<td>RE</td>
<td>Requirements Engineering</td>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>NLBR</td>
<td>Natural Language Based Requirements</td>
<td>GPS</td>
<td>Global Positioning System</td>
</tr>
<tr>
<td>SRE</td>
<td>Software Requirements Engineer</td>
<td>NLP</td>
<td>Natural Language Processing</td>
</tr>
<tr>
<td>NLP</td>
<td>Natural Language Processing</td>
<td>ILP</td>
<td>Inductive Logic Programming</td>
</tr>
<tr>
<td>RTCA</td>
<td>Radio Technical Commission for Aeronautics</td>
<td>SDLC</td>
<td>Software Development Life Cycle</td>
</tr>
<tr>
<td>RDL-2</td>
<td>Requirements Definition Language</td>
<td>OCL</td>
<td>Object Constraint Language</td>
</tr>
<tr>
<td>BNF</td>
<td>Backus-Naur Form</td>
<td>DSBK</td>
<td>Domain Specific Background Knowledge</td>
</tr>
</tbody>
</table>
1 Introduction

Software is omnipresent. It has become increasingly difficult to imagine all our technological advancement without software. Software drives the pacemakers and carries humans over seas. The importance of the quality and reliability of software is increasing as more and more lives depend on it. The avionics industry has been using software in fly-by-wire systems from a long time [47]. Since the state of flying directly depends on the software, the software doing this is highly flight critical. According to [76] [27], to ensure reliability and correctness of software, 30-40 percent of the costs of development of a critical software system are allocated towards verification and validation efforts. A big portion of this cost goes towards simulation, modeling and testing.

In recent years, the industry has become increasingly interested in the application of formal methods because of the promise of automating verification activities to a great extent if applied properly. This is especially enticing because the regulations that the airplanes have to conform to, increase the costs of development. If processes and technologies could help control the cost of development while upholding to rigorous standards of safety then it makes sense to invest in these technologies.

Formal methods are a set of techniques that have a mathematical basis which could be used for specification, development and verification of systems[67]. These techniques range from using models of a system to the verification of the actual developed system against certain formalized specifications[13]. These specifications express requirements to the system and any assumptions about the environment of the system in an unambiguous and objective form using a formal specification language. Even though the specification language used to express these might be completely unnatural for an untrained eye, it is still a formal specification which has a very exact meaning in its formally defined language semantics.

For example, a sentence in a specific natural language (English) means nothing to the person who does not know that language but the same sentence informs a person, with prior knowledge of that language, a specific meaning. Although in this case the human can interpret the meaning of that sentence based on his understanding of the language and context, in case of a formal specification in a formal language the formal statement has one and only one meaning because of well defined logical and mathematical semantics.

The formal specifications constructed using formal languages are used in the formal verification of systems. Temporal logics are one type of these formal languages, for example Computation Tree Logic (CTL) [24] and its numerous extensions [21][18]. These temporal logic languages express into what states a system can go and in what order.
Another type of formal language is first order logic \[9\] which expresses what properties will the system ensure and under what conditions. Such formal languages help in specifying what the system must do and can later be used to prove that the system obeys those properties using model-checking \[23\] or theorem proving \[32\].

The formal verification process relies on models and formal specifications. Making sure that the model under verification is a correct and close representation of the properties of the underlying system is a huge challenge in itself as explained in \[79\]. It is widely accepted that “... it makes much more practical sense to require that the model faithfully capture the system behavior only to the extent demanded by the objectives of the simulation study” \[79\]. Current research effort focuses on creating different views of a model, each based on different objectives that are needed to be analyzed/studied. These views are created from the same system model. The details that are relevant to a view are retained and the details that are not relevant to the objectives of that view are abstracted away \[49\]. These views then allow to focus and organize the analysis effort. It could also be used to facilitate model checking in the future because it could reduce the required resources for a model checking problem by reducing the model under analysis.

The formal specifications are similar to models with respect to the question of correctness. Is the formal specification correct with respect to the intended system? Is it expressed correctly by the engineer? These questions are typically difficult to answer. The engineer must be experienced enough with the formalism and must have a clear understanding of all textual requirements (in natural language) of the system to form a correct formal specification. The engineer must think about all situations the system can be in and then write a formal specification that correctly describes expected system behavior in all these situations. This is a lot to think about for an engineer. For small systems it is easy to formalize correct specifications because the understanding of the complete system is easy to acquire. For realistic systems it is difficult to imagine that an engineer can describe a component’s behavior correctly with respect to all operating conditions.

A program (in a programming language) describes what steps a system takes to accomplish a goal. A programmer therefore thinks about every small step that the system must take to accomplish that goal in some context. This context is typically implicit because of the order of operations that the program expects to have occurred before a certain point. This allows the programmer to only think about the next correct step to take in that context. This step-wise thinking, that is available in a programming language, to write a program is difficult to see in the writing of a formal specification, for example in FOL or Temporal logics. Every statement that is written in FOL must take the complete system into account; if a FOL formula is to be valid only in a certain context then this context must be made explicit within that formula. With temporal logics the statements are typically made about the system as a whole anyway.

This shows that formal specifications typically comment on the global view of a system. One reason for this could be that a formal specification is supposed to describe what the system shall do. A complete answer to this question must always take the complete system and all its operating conditions into consideration.
Pi-Calculus is a process algebra that allows modeling communicating and mobile\(^1\) systems [84]. Process algebra is an approach used to model concurrent systems. Pi-calculus has found multiple applications in the analysis of critical embedded systems [85][83]. Despite this, some researchers opine that “process algebras have not achieved a breakthrough into the embedded system domain. This may be due to the fact that the algebraic character of process calculi is not so appealing to engineers” [56]. While process calculi are a small part of the domain of formal methods, statements like these highlight that the engineers need something that is simple to apply and understand and at the same time provides all the benefits expected from a formal verification technique.

The goal of this work is to demonstrate that it is possible to generate formal specifications in an intuitive manner from more natural user inputs than formal languages themselves. As a result of this work it must be possible for an engineer to think in small steps to specify a system. The specifications generated should be in popular formal languages such that they could eventually be used in formal verification tools already available in the industry.

1.1 Motivation

The Radio Technical Commission for Aeronautics (RTCA) and European Organisation for Civil Aviation Equipment (EUROCAE) publishes documents that are used to guide development processes in the aerospace industry for airworthiness certification by government regulatory authorities. As a supplement to DO-178C [68], DO-333 [67] was published by the RTCA and EUROCAE that focuses on using formal methods as a source of software verification evidence. It provides the guidelines and extensions to the software life cycle and the artifacts that can be derived from the formal verification effort to suffice as verification evidence in the software life cycle.

This acknowledgment of formal methods by regulatory authorities shows the industry a clear path for integrating formal methods as verification evidence but the barrier in integrating formal methods in the development processes is the initial increase in development costs due to training of people and the actual development of formal specifications as shown in [36]. There is no clear incentive in the industry to increase costs of development without having a definite guarantee of reduced total costs or increased safety. Formal methods therefore have to compete with an already established process of airworthiness certification.

A lucrative way forward would be to reduce the initial costs of integrating formal methods to a level that makes formal methods as competitive as industry standards like software testing so that both testing and formal methods can be used in conjunction to ensure safer software. One technique can be used to fill the gaps left by the other. The key motivating challenges can be summarized by:

\(^{1}\)In this context, mobility refers to the re-configurability of a system
1 Introduction

- Formal specifications are costly to write because of costs of training and costs related to changing a well established process.
- Formal languages are unnatural for engineers, difficult to understand at first glance and backed by significantly complex semantics.

Since a lot of formal verification hinges on the existence of good formal specifications, this work attempts to make the formal specification effort much easier to accomplish. It shall reduce the effort required to make formal specifications from natural language requirements. The key guiding principles of all of this work would be:

- Keep the needs of an engineer in focus. Any human effort that is required to create formal specifications should be small and intuitive to understand.
- The validity of the output is crucial. During the generation process, only those formal specifications are generated that can be proven to be true within the system even if it may be possible to generate more.

Keeping these challenges and principles in mind the scope and objective of this work will be discussed.

1.2 Scope and Objective

This work is intended to be applicable in the requirements specification phase of a development process where the actual system being developed is not available. It is expected that during the requirements specification phase an engineer uses the system presented in this work. The engineer provides the system the required inputs to generate formal specifications that represent the system requirements.

The proposed system can be used even when the system (software or hardware) to be specified is available. This opens up another set of possibilities but these are not discussed and will not be the focus of this work.

The objective of this work is to propose a system that allows the engineer to think in small steps to specify a system. All small specification steps are then used to form a complete picture of the system to be developed (or specified) called the target system from here on. The proposed system then generates specifications that describe the target system in formal specification languages that can be used for verification efforts.

In this work, the type of inputs to the proposed system and an algorithm that uses these inputs to generate formal specifications in FOL and as Kripke structures is proposed. Within the scope of this work, it is attempted to answer the following research questions:

- How can the formal specifications be generated from example data?

This work does not attempt to answer if the example data expresses all relevant properties of a system. It also does not aim to generate all possible formal
1 Introduction

specifications that are valid for this data. It will be made clear that this is not always valuable to an engineer.

- Is it possible to generate finitely many valid formal specifications for a system?

This work does not comment on the completeness of the generated formal specifications. It only comments on the correctness of the created formal specifications. It may be possible to generate infinitely many formal specifications for a system but this work focuses on generating a finite number of formal specifications that provide value to the requirements engineer.

- What information must be included within the example data to generate formal specifications?

This work does not examine the relationship between the actual system under development and the example data. It does not attempt to answer the question if all relevant information about the system is included in the example data. This work attempts to answer the question that if formal specifications are to be generated from example data, what information about that data is needed to generate formal specifications.

1.3 Outline

This thesis is organized as follows. Chapter 2 provides the necessary background information as context for this work. It describes the special challenges of the avionics domain, the safety regulations and the development processes relevant for this work. It also provides the fundamentals of formal specification and verification and presents the formal languages used as a part of this work. Chapter 3 presents the relevant state of the art. It discusses the quality of requirements and ways to formalize them. It also focuses on the development strategies relevant for this work. It then presents the work done to generate formal specifications from data. Chapter 4 presents a running example that provides the context for the rest of the figures and examples. Chapter 5 presents the proposed approach. It discusses the inputs to the proposed system and the information that is needed in the inputs to generate formal specifications. It presents the algorithms used by the proposed system to create formal specifications. It then discusses the quality of the produced outputs and any limitations thereof. Chapter 6 presents strictly relevant implementation details. Most importantly it presents a class diagram of the system and an activity diagram that explains the working of the proposed system. Chapter 7 evaluates the performance of the proposed system and the effectiveness of the proposed approach. It evaluates the feasibility of generation of formal specifications from the proposed inputs. Chapter 8 concludes the thesis by providing a summary. Chapter 9 presents various scope of improvements and extensions to the proposed approach.
2 Background

Building software for avionic systems is a big challenge. A lot of lives depend on the correct functioning of an avionic software system. There are strict regulations in place all over the world to ensure airworthiness of a commercial aircraft. These regulations establish a certification process for every avionic system and also provide strong guidelines on how the development process must be carried out and the artifacts that must be produced as a result of the development effort. These artifacts also include “verification evidence”[68]. Verification evidence are artifacts that demonstrate that certain aspects of the system are verified to be correct with respect to the requirements. The requirements themselves have to be demonstrated to be correct with respect to the intended functionality of the system. The production of verification evidence at every stage of the development process is regulated and this is briefly discussed in section 2.2.2.

Formal specification and verification is one method of providing verification evidence [67]. This has gained popularity in recent times due to an expected automation of the verification tasks. Formal verification requires formal specifications as an input. To understand the significance of formal specifications we must understand how they are used and what do they look like. The following sections explain formal specification and verification techniques and two types of formal languages. A formal language, unlike natural language has a restricted set of symbols and a well defined syntax. Mathematical logic and temporal logic are two different types of formal systems. Each type can have multiple formal languages. Propositional logic and FOL can be classified as mathematical logic. Linear Temporal Logic (LTL) and Kripke structures can be classified as temporal logic. All of these will be discussed in this chapter.

FOL and Kripke structures are especially interesting because they will be the outputs of the proposed system. FOL can be used to describe mathematical properties of a system and Kripke structures can be used to describe stateful properties of a system. These formalisms were chosen to target both mathematical and stateful attributes of an avionic software system. These will be discussed further.

2.1 Avionic Software Systems

Avionic software systems are simply embedded software systems whose development is strictly regulated to ensure safe operation. All program code that goes on the microcontroller unit or the computer of an avionic system must undergo extensive testing to
ensure safe operation under all operational conditions. All sub-routines created as a part of the development process must always trace back to a requirement and a reason for doing it a specific way must be documented. These extra steps that increase the safety of an avionic software system also contribute to the significant costs of development.

A very key aspect of avionic software systems is that the software produced must be built to last for the complete service life of an aircraft. This could also go up to 50 years and even more. This means that some of the software running on such an aircraft could actually have been written 60 years before its last flight. This aspect of avionic software is very unique because when people think about software, people imagine something that could be updated easily but this is not the case with avionic software systems. This does not mean that software is never updated, it just means that the updates are difficult to do unlike conventional consumer software systems.

In terms of functionality, avionic software systems are similar to embedded systems. In a general sense, they can function as an electronic control system, a data transformation system or a state machine. Any software is a combination of one or more of these basic functionalities. These categories and examples of them are shown in Table 2.1. These categories are only meant to provide an overview of the types of software and the avionic systems that fall under them. This shall not be considered a complete categorization of all avionic systems.

<table>
<thead>
<tr>
<th>Category of Software</th>
<th>Aircraft Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Electronic Control System</td>
<td>Flight Control, Temperature Control</td>
</tr>
<tr>
<td>Data Transformation and Representation</td>
<td>Fuel Gauging, Sensor displays</td>
</tr>
<tr>
<td>Stateful Systems</td>
<td>Emergency Systems</td>
</tr>
</tbody>
</table>

Table 2.1: Types of software in Aircraft systems

There are multiple control systems on an aircraft. The fly-by-wire systems completely decouple the control action of the pilot from the force applied on the actuator using electrical signals. This allows better aircraft control at higher speeds because the actual effort of moving the control surfaces of an aircraft is not anymore done by the pilot but by an actuator. The flight physics and the control laws of an aircraft define how the aircraft will react to the control inputs as well as to the changes in the environment. These changes in the environment can also be mitigated by control software to allow the crew to have less things to worry about[61].

The data transformation and representation systems refers to software responsible of displaying sensor values and errors/warnings. This requires careful interpretation of a combination of sensor values before issuing a warning. Multi Function Displays (MFDs) are now common to display aircraft data and areas on the display can be reserved for warning messages[61]. A combination of audio and visual messages are also commonly used to provide critical information to the pilot immediately.

Most software is composed of stateful systems. The command to descend the landing
2 Background

gear or the enabling or disabling of the autopilot are examples of stateful behavior. These are triggered by the pilot. Emergency systems that detect and respond to critical events are also examples of stateful systems. Here the software can use a sensor value to change the operational state of the aircraft. For example, low oxygen concentrations in the cabin can issue the descending of oxygen masks and enabling of oxygen flow through them.

This basic classification of avionic software systems allows this work to focus the effort in and around these categories of software.

Predictability is also a very key aspect of avionic software systems. In [86], a real time decision making system is proposed. This, on the basis of rules, provides decisions in a predictable time. This real-time behavior is critical to ensure that a system responds to the changes in its environment in a predictable time. Real-time requirements are very common in the flight control computer of an avionic system. This is because a control action is flight critical and must obey its deadline for a safe flight. Rapid prototyping of real-time control laws is proposed in [82] using a layered approach. It allows refining the control laws using a Hardware in the loop simulation. This lowers the time needed for development because the developers need not wait for the availability of actual hardware.

There are several interesting projects being executed in the professorship of computer engineering at TU Chemnitz. The OPIRA (Open Innovation for RPAS\(^1\)) project aims to “enhance safety and operation of RPAS”. One of the sub-project used high-resolution image sensors to identify the landing strip at an airport to assist the pilot during the landing phase [88]. Field Programmable Gate Array (FPGA) based image processing technique to detect straight lines was presented in [81] as a part of this project. These approaches significantly enhance the performance of image processing applications. These approaches can also be used to enable automated landing.

The APOLI (Automated Power Line Inspection) project aims to develop highly automated vision-based inspection systems for electric power distribution systems [89]. A blob detector based burn mark detection approach was presented in [90]. It is used to identify damaged insulators which are employed in high voltage transmission lines.

2.2 Development Process

The development process for avionics is traditionally the V-Model or a variant thereof. The reason for this could be that in some countries the V-model is mandated for federal projects [51] by the government. Another reason could be that the DO-178C standard that is used for regulating software development in the avionics industry provides concrete objectives that must be fulfilled for airworthiness certification. The timing and production of the artifacts corresponding to these objectives fits well with the V-Model and this might also contribute to the avionic industry choosing V-Model as a development process[41]. It must be made clear that the standard by itself does not state that the V-Model must

\(^1\)Remotely Piloted Aircraft Systems
be used as a development process, but its noteworthy that the development process/life cycle is motivated by the requirements of the regulations. In comparison other Software Development Life Cycle (SDLC) models exist (e.g. Agile, Iterative, Spiral) which are gaining popularity not only in the critical embedded systems domain[59] but also in the avionics industry[41].

Since currently V-Model is still the most prevalent, this and the regulations in the avionics industry will be discussed to get an insight of what are the challenges faced by the industry in a development process context.

### 2.2.1 V-Model

The V-Model [37] is a development process that separates the pre-development, development and post-development phase of a software life cycle. The left side of the V is the pre-development phase, the right is the post-development phase and the tip of the V is the development phase. An immediate observation that can be made is that the development phase happens only once in the software life cycle. This means that this development process is not so suitable if there are a lot of changes in requirements or errors come up during the post-development phase because this would mean that all the steps of the left side of the V will have to be repeated and, more importantly, in doing this it must be ensured that the changes are consistent with the previous effort.

The V-Model is illustrated in Figure 2.1. The blue boxes on the top summarize the regions in a V-Model. The pre-development phase has activities related to the safety analysis of the aircraft and its subsystems. As a result of these activities, subsystems are added and the requirements are detailed. During this phase a lot of requirements are also allocated to hardware, and software is also allocated to hardware and assigned its own set of software requirements. Also, during this phase itself, the verification activities are defined. This is demonstrated by the blue arrows directed from the left side of the V to the right side of the V.

The “Item Design” phase is illustrated in the top centre of Figure 2.1. During this phase the hardware and software development takes place. Once the development is completed, the previously defined verification activities, along with system integration is done to reach the end of the V-Model. At this point the end-product shall be ready to be delivered.

The interesting thing to note here is that during this complete process there is nothing that goes wrong, there is no phase that is specifically dedicated to change in requirements or change in the needs of the customer. This means that change is not a part of the development life cycle of the product. It can also be seen that “maintenance” is also not a part of the development life cycle of the product. This does not mean change or maintenance is impossible to handle, it just means that to do this one has to go back to the left side of the V and make sure that while changing something, nothing else gets affected or else there would be a lot of rework.
Advantages in Avionics

To summarize, following are the major advantages of the V-Model in context of avionics:

- The model supports systematic planning which is crucial for safety analysis of a system. This plan is also requested by airworthiness certification authorities.
- Verification tests are planned during the requirements phase which ensures consistency between the tests and the requirements automatically.
- Well organized and structured process gives assurance that everything is planned properly and in steps.
- The step-wise nature allows to monitor progress with milestones and set up realistic timelines.
- Relies on technology being known and stable throughout the process. In case of avionics, where development life cycles could be as long as 5-10 years with service life of several decades, the development process must be suited to have a rigid schedule and development in a step-wise manner and this is exactly what the V-Model provides.

Disadvantages in Avionics

To summarize, following are the major disadvantages of the V-model in context of avionics.
2 Background

- Having a very rigid schedule means that new technologies that are discovered during the development life cycle cannot be used easily. This means when a product is released it is very likely that it has old technology.

- Changes can mess up consistency. Change management is made to be a separate topic along with the V-Model because changes are not handled within the model itself. A change may or may not affect already generated artifacts and therefore extreme care must be taken when changing something.

- Limited parallelism. The only way to have parallelism is to have two V-Models simultaneously. The model in itself doesn’t support proper parallelism. The way to achieve this is to create two different V-models for different features of a big product development. These are then carefully coordinated.

- The V-Model does not have a phase for exploratory testing but only for planned testing. The V-Model talks about verification tests which are definitely important but exploratory testing is also important in discovering bugs[46][72][4]. An iterative life cycle allows exploration opportunity multiple times which is a lot more beneficial than doing it only once at the end as a separate addition to the process.

2.2.2 Regulations

The RTCA and EUROCAE publishes documents that serve as guidance and recommendations to the avionics industry for the development of all avionic systems. Of these set of documents the DO-178C refers to “Software Considerations in Airborne Systems and Equipment Certification”. This document specifically discusses the software life cycle processes and its objectives, activities and evidences of satisfaction of these objectives. This serves as a guideline to airplane system manufacturers to ensure that during the life cycle of the development of software, proper evidence is generated in line with the objectives of this document.

The DO-333 is a formal methods supplement to the DO-178C which takes into account the use of formal methods as a part of the development process. This document presents the changes or extensions that would be made to DO-178C objectives and activities such that formal methods can be used to provide verification evidence.

This work targets the creation of formal specifications therefore the key points of both the documents in context of this work and its application will be discussed.

DO-178C

The DO-178C [68] talks about software development life cycle and the processes in it. The processes are needed to produce life cycle data that is used for the airworthiness certification. The following points attempt to summarize the basic direction of the guidelines provided in the document. The key takeaways from this document are discussed later.
2 Background

- **Software Level Definition**: All software sub-systems have to be classified based on a Safety Assessment Process into levels from A-E. This classification is based on the impact of failure of the software module on the operation of the aircraft. The 5 failure categories (Software level classification) are Catastrophic (A), Hazardous (B), Major (C), Minor (D) and No Safety Effect (E). The definitions of these failure categories can be found in the glossary.

- **Software Life Cycle Processes**: The software life cycle processes can be categorized into 3 major categories: Planning, Development and Integral processes. The planning process is concerned with defining the activities that will be carried out in the development and the integral processes. The development process is formed of the requirements, design, coding, integration processes. The integral process has the software verification, configuration management, quality assurance and the certification liaison processes.

- The document does not prescribe any software development life cycle model like the V-Model or otherwise. In fact in section 3.2 of DO-178C, the possibility of an iterative\(^1\) development process is also shown.

- The section on requirements process states that an analysis of the created system requirements must be done for “ambiguities, inconsistencies and undefined conditions”. In the document it is mentioned that this analysis can be completed through reviews of the requirements.

- The section on software verification process shows testing and reviews as the way to produce verification evidence.

The key influence of this document on the work of this thesis is discussed in the following points:

- The document clearly defines the different development phases and this work in particular is relevant for the requirements process.

- The development processes highlighted in this document are really flexible. The document itself does not require a use of a specific development process. The industry on the other hand favors the V-Model significantly. This work shall not focus on changing the development process. This work rather, focuses on the requirements phase that is always present in any development process and suggests ways to improve this phase as a whole.

- The primary way to validate a requirement is through a review. This means that textual requirements are expected to be discussed for “ambiguities, inconsistencies and undefined conditions”. Formalization of these requirements can allow for the analysis of ambiguities, inconsistencies and undefined conditions to be automated.

---

\(^1\)A cyclic development process that allows the sequence of phases of Requirements, Design & Implement, Test, Evaluate to repeat. After the evaluation phase new requirements can be created and another iteration of development can take place.
This work contributes in the formalization of the requirements using example data. The formal specifications generated through this work can be used to achieve the above stated requirement validation objectives.

- Verification evidence that needs to be produced as a part of the verification and validation phases can be assisted through formal methods and this is discussed in the section on DO-333.

DO-333

The DO-333 [67] is the formal methods supplement for DO-178C and DO-278A. This document lays the foundation on the basis of which the avionics industry can include formal methods in their development process. It provides clear guidelines on what and where can formal methods be used in the development process and what verification evidence is considered sufficient. Some key observations are listed below:

- During most activities in the verification process, formal methods can produce verification evidence of consistency between high and lower level requirements but it cannot produce the reasoning and rational of the existence of low level or derived requirements from the high level requirements. This must still be done by review.

- All formal verification activities rely on the existence of a formal model or formal specifications. This is echoed throughout the document and is of course obvious. This simply highlights the importance of formal specifications.

- “As a minimum, testing will still be required to ensure that the Executable Object Code is compatible with the Target Computer”[68]. This statement is the reason why formal methods by themselves do not provide enough value/incentive to the industry. It is because all tests created for the target computer can also be run during development but this is not true for all formal methods. If testing must exist to verify code on the target computer then formal methods will just be an added activity to the development process, thereby having a definite increase in development cost. If a process existed that allowed formal methods to be directly used for testing purposes in the end, then the costs to convert to formal methods will be justified and then it will gain much more traction with the financial and business side of the avionics industry.

- Section FM6.7 in the document discusses the verification of executable object code for certain properties. These are typically run-time properties relating to memory bounds and data bus contentions etc.. Many run-time properties can be safely guaranteed by formal methods when an appropriate model is available. This cannot be done by testing alone. This is a huge selling point for formal methods and it is the reason why formal methods have started gaining traction in the industry.

- The document in section FM12.3.5 states that at the time of writing, approaches that use a combination of formal methods and testing, for one software component
were not available. A coverage analysis must always be made when using other methods of verification along with formal methods.

The takeaway from this document is that the regulatory authorities have paved a way for formal methods to be integrated into the development processes of the avionics industry. The key influences of this document to this work is discussed below:

- Correct formal specifications are key to any formal method. This work makes the creation of the formal specifications easier as compared to writing the specifications directly in a formal language.

- Formal methods can be used in some parts to produce verification evidence but till now using formal methods to verify compatibility with the target computer is not sufficient, until this changes testing and formal methods have to co-exist. Testing of Executable Object Code is a necessary process. This work uses example data to generate formal specifications. This example data can also be used to generate verification and validation test cases. Also there is sufficient research [44] [35] which can help us generate tests from formal specifications which can be used to produce verification evidence in compliance with DO-178C.

We now understand that avionic software systems present extraordinary challenges in comparison with embedded systems. The inclusion of formal methods is gaining popularity in the avionics industry, but for any formal method we need a good formal specification. Different formal specification languages are discussed in the following sections.

### 2.3 Logics: Propositional and First Order

In this section we discuss both Propositional Logic and First Order Logic since both will be used within the scope of this thesis. Propositional logic is sometimes referred to as zeroth-order logic. It is considered weaker in its expressive power when compared to first order logic.

#### 2.3.1 Propositional Logic

Propositional logic\(^1\) statements are built with logical connectives and propositions. Logical connectives are: not\((\neg)\), and\((\wedge)\), or\((\vee)\), implication\((\implies)\), and biconditional\((\iff)\). The exact meaning of each of these connectives is not relevant for this discourse and it can be looked up in the reference mentioned in the footnote. Propositions are statements that can only be assigned a truth or false value. Propositions themselves do not have variables, instead they have uninterpreted symbols. An uninterpreted symbol is assigned

\(^1\)For a more formal analysis of FOL and propositional logic refer to [5]. This is not strictly necessary for the reader but it is recommended that the reader has enough understanding to differentiate between FOL and propositional logic formulae (Discussed in this section).
a value when an interpretation is applied to the complete statement. A variable is a symbol that, even after application of an interpretation, can have more than one value. An interpretation function assigns truth values to propositions by assigning meanings to uninterpreted symbols according to the interpretation theory. The interpretation theory is essentially what gives meanings to a propositional formula and it is simply a set of facts that are stated as axioms in that theory. To illustrate all these definitions, an example is taken in Figure 2.2.

\[
I \models P \land Q \land (a < b)
\]

read as

The right hand side is true and is a semantic consequence of the interpretation ‘I’ on the left hand side. This essentially means if the theory/model ‘I’ is true then the right hand side can be proven to be true.

where

- \(P\) and \(Q\): Atomic propositions, evaluated based on the interpretation
- \(\land\): Logical connective defined by the valuation function
- \(I\): The interpretation theory/model which contains the interpretation function. It also defines the meaning of the symbols ‘(‘, ‘)’, ‘<’, ‘a’, ‘b’ and then defines if ‘(a < b)’ is evaluated to true or false.

Figure 2.2: Propositional Logic illustrated

The right hand side in the formula in figure 2.2 is the propositional formula on which the interpretation of \(I\) is applied. \(P, Q, (,), a, <, b\) are simply uninterpreted symbols that are in the alphabet of the formal language. The connective in this example is \(\land\) which is evaluated by the valuation function. In case of propositional logic, this symbol(\(\land\)) is typically assigned the meaning of a ‘boolean and’.

The most essential thing to understand is that with a propositional logic formula reasoning about a set of objects cannot be done using one single interpretation because that one interpretation cannot assign multiple values to a symbol. As will be noted with first order logic, one symbol can be interpreted as an element of a set and therefore can be interpreted as having multiple values within one interpretation.

### 2.3.2 First Order Logic

First Order Logic (FOL) \(^2\) is a logical formalism which builds upon propositional logic. It allows to introduce predicates to describe relationships between different objects and allows to reason about sets of objects. Predicates can be understood as functions that

\(^2\)For a more formal analysis of FOL and propositional logic refer to [5]. This is not strictly necessary for the reader but it is recommended that the reader has enough understanding to differentiate between FOL and propositional logic formulae (Discussed in this section).
2 Background

can take arbitrary number of inputs and output a boolean value: true or false. The reasoning on a set of objects is done using quantifiers. This is called quantification of a variable over its domain of discourse. Quantification is to state (or quantify) what number of objects from the domain, when assigned to the variable in the formula, would result in the formula being true. This is illustrated in figure 2.3. It must be observed here that the meaning of the FOL statement cannot be interpreted until a domain of discourse for \( x \) and \( y \) is defined and the meaning of ‘lessThan’ is defined. This is done using a structure\(^1\) and an interpretation\(^2\).

\[
\forall x. \exists y. \text{lessThan}(x, y)
\]

read as

for all \( x \) in the domain of \( x \) there exists a \( y \) in the domain of \( y \) such that the predicate lessThan is true when the tuple \((x,y)\) is assigned to it.

where

\( x, y \) : The variables in the formula

lessThan : Predicate whose meaning depends on the interpretation

\( \forall, \exists \) : quantifiers that can be applied on variable names that immediately follow the quantifier. e.g. ‘x’ in \( \forall x \).

Figure 2.3: First Order Logic illustrated

FOL has quantifiers \( \forall \) (for-all) and \( \exists \) (exists) which quantify the variable that follows them. The \( \forall \) quantifier means that the FOL formula will be true for all values of the for-all quantified variable in its domain. The \( \exists \) quantifier means that, in the domain of the quantified variable there exists at least one value for which the FOL formula is true.

The main differences that emerge when compared to propositional logic are:

- A domain of discourse is talked about in first order logic. With propositional logic the ‘domain’ is implicitly defined by the interpretation function because the uninterpreted symbols take on one single value according to the interpretation function. By changing the interpretation function multiple times, multiple valuations can be done for a propositional logic formula.

- First Order Logic evaluation requires a ‘structure’, ‘interpretation model/theory’ and a ‘valuation function’. With propositional logic interpretation and valuation functions are enough to evaluate a true or false value of a formula.

- The presence of a domain of discourse for variables means that these variables can be quantified over that domain. This is done using quantifiers. Propositional logic does not have quantifiers because it does not reason over a set of objects.

\(^1\)defines the sets of the domain of discourse for all variables. It also defines what operations are valid on this set.

\(^2\)similar to the description in propositional logic. It interprets the non-logical symbols i.e. recognizes predicates, variables, functions and constants and assigns them meaning accordingly.
To differentiate between a propositional logic statement and a FOL statement the example in Figure 2.4 is shown. Figure 2.4a illustrates the following natural language statement ‘b is equal to 3 implies 3 is less than 5 and b is less than 10’. In a logical sense if b is not equal to 3 even then this propositional logic formula will be evaluated to true because of the implies operator ¹.

A different and comparable first order logic statement can also be made. Figure 2.4b states that ‘there exists a b where b is less than 5 and the predicate lessThan is true for the assignments (b, 10)’. The structure defines the variable b to have the domain of natural numbers and an interpretation similar to propositional logic can be used along with the definition of the predicate.

It can be seen that this FOL formula reasons about the complete set of values the variable b can take. It can be imagined that if b is replaced by any of {1,2,3,4}, which are in the set of natural numbers, both constituent expressions (separated by ∧) will be evaluated to true. Therefore the FOL formula is true that there exists at least one assignment of b such that the expressions are true.

\[
\begin{align*}
\text{Propositional Logic comparison example} & \quad \text{First Order Logic comparison example} \\
\begin{array}{c}
b = 3 \implies 3 < 5 \land b < 10 \\
\land : \text{logical and} \\
b : \text{b can have any interpretation and} \\
even then this formula will be true \\
=,<,3,5,10 : \text{These are defined by the interpretation}
\end{array} & \quad \begin{array}{c}
\exists b. b < 5 \land \text{lessThan}(b, 10) \\
\exists : \text{The exists quantifier} \\
\text{lessThan}(c, d) : \text{Predicate which is true if } c < d
\end{array}
\end{align*}
\]

Figure 2.4: Comparison between Propositional Logic and FOL

Usage

FOL can be used to express different types of properties. It can also be used to formalize software requirements. In tools like SPARK ² first order logic formulae can be written in the SPARK language which is a subset of the Ada programming language. A brief overview of using SPARK in alignment with objectives of the DO-178C is shown in [69]. The GNAT toolsuite ³ is a set of tools that can be used for development and verification activities. It supports the analysis of SPARK contracts with respect to the code. The SPARK contracts are proven over the semantics of the program and the programming language (ref section 2.6.3). If a proof cannot be found or cannot be completed then the programmer has to assist the tool in proving a contract by changing either the source

¹\( A \implies B \) is logically equivalent to \( \neg A \lor B \)
²https://www.adacore.com/about-spark (accessed on 22.08.2019)
³https://www.adacore.com/gnatpro/toolsuite (accessed on 22.08.2019)
2 Background

code or the contract. When a proof of a contract is successful then that contract is considered to be proven over the software. This provides a guarantee that the software will obey the contract.

In effect, if a system could generate the first order logic formulae then these could be used to input to the SPARK system. The developer can then write the source code according to the requirements. The properties can then be proven with respect to the source code. When the developer is finished and all the properties have been proven then this can be used as verification evidence for those properties of the system.

2.4 Kripke Structure

Kripke structures [50] are a variation of transition systems as in computer science. A kripke structure can be visualized as a set of states with unguarded transitions between them but every state has a label of the true properties in that state. This is shown in Figure 2.5. In Figure 2.5, 3 states $s_1$, $s_2$ and $s_3$ can be seen where $s_1$ is the only initial state. The arrows indicate the allowed transitions between the states. The $p$ and $q$ in curly brackets '{}' indicate the property names that are true in that state. The properties that are false in that state can also be computed as shown later.

Formally a kripke structure is defined as $K = \langle S, I, R, L \rangle$[25]. The sets $S, I, R, L$ are described as follows:

- set of atomic propositions ‘A’. This set is not directly part of a Kripke Structure but it is needed to define it.
- set of states ‘S’.
- set of initial states ‘I’ where $I \subseteq S$.

![Figure 2.5: Visual Example of a Kripke Structure [40]](image-url)
2 Background

- a transition relation ‘R’ where \( R \subseteq S \times S \). This simply means that all transitions that exist in the Kripke Structure will be a part of set \( R \) as a pair of states.
- a labeling function ‘L’ where \( L : S \rightarrow \mathcal{P}(A) \). Read as “Labeling function ‘L’ maps elements of the set ‘S’ to elements of the power set\(^1\) of ‘A’”. This function maps each state to a Label that carries the information of what atomic propositions are true in that state. This label is in itself stored within the state.

The properties that are false in a specific state \( s \in S \) can be computed by the set difference operator (\( \setminus \)). Set of false properties is then given by \( A \setminus L(s) \)

Usage

A Kripke Structure is used to describe the behavior of a stateful system whose properties change with every change of state. Since most avionic software systems have at least some stateful component, this formal language is a concise way of representing software behavior.

Kripke structures can be converted to Büchi automata \([22]\). Büchi automata are automata which accept infinite input sequences as long they result in a run of the automaton that visits an acceptance state infinitely often. An acceptance state is a state in the automata that is marked to check for valid input sequences to a Büchi automaton. This conversion of a Kripke structure to a Büchi automaton will be used in the later sections.

2.5 Linear Temporal Logic

Temporal logic is a branch of logic that is used to reason about the evolution of a system in its life cycle. This evolution may not be related to the physical notion of time in seconds, typically it only relates to the evolution of system states in a particular order. Temporal logic is used for reasoning about this evolution. A sequence of states for an example system in figure 2.5 could be \( s_1 \rightarrow s_2 \rightarrow s_3 \). This is only one of the many possible sequences that this system could go through. For example another sequence could be \( s_1 \rightarrow s_2 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \). This simple existence of more than one sequence of states for one single system model shows the existence of two different notions of evolution. These are categorized as linear time and branching time. With linear temporal logic \([65]\), the logical reasoning is restricted to only one such sequence of states at a time. With branching time logic, the logical reasoning takes into consideration all possible sequences of states at the same time.

LTL provides the following operators to reason about such infinitely long state sequences:

- **Globally**: denoted by \( G \) - A formula like \( G(p) \) states that the property ‘p’ will be true in all states in the sequence of states.

\(^1\)A Power set of a set contains all possible subsets of that set including the empty set
2 Background

- **Finally:** denoted by $F$ - A formula like $F(p)$ states that the property ‘$p$’ will eventually be true in the sequence of states.

- **Next:** denoted by $X$ - A formula like $X(p)$ states that the property ‘$p$’ will be true in the next state. This is where the notion of the current state becomes important.

- **Until:** denoted by $U$ - This is a binary operator. A formula like $pUq$ states that the property ‘$p$’ will remain true in all states preceding the state where ‘$q$’ is true.

Originally in [65] Pnueli proposed the $G$ and the $F$ operators. The others were added as extensions. Presently, the operators stated above are the most popular ones although many extensions or variations might exist.

As can be quickly understood a sequential program fits very well into the LTL style of reasoning and this was the original idea of Pnueli. A complete formal description of LTL is omitted because it is not relevant to this discourse. This description of LTL shall form the basis of the discussions that will follow in section 2.6.2.

**Usage**

LTL statements can be used to reason about stateful systems in a concise way. The exact mechanism is not relevant for this work. The interesting property of LTL formulae are that they can be converted to Büchi automaton as well just like Kripke structures [22]. This interesting property is used by model checkers as will be discussed in the section 2.6.2.

2.6 Formal Specification and Verification

Using formal languages to model the system and its expected behavior can be used to verify this model against its expected behavior. This verification can help in understanding the system design and its validity with respect to the expected behavior. In case of protocol models this verification can sometimes highlight protocol level problems. In some cases it is also possible to use the actual system or parts of the actual system for automatically checking against the desired properties. This checking can be performed by model checkers or theorem provers.

The discussion of regulations and avionic software systems showed that it is possible to have formal verification processes integrated within the software development process. This section shall focus on answering why is formal verification needed and how can one carry out formal verification.

2.6.1 What is Formal Specification and Verification

Simply put, formal verification is (automatic) mathematical verification of a formal model against formal specifications. A formal model is a model that is stated in a formal
language such that the model itself does not have ambiguities or inconsistencies and represents the system under analysis. A formal specification is a statement in a formal language that expresses the expected behavior of the system under analysis. In contrast an informal specification is a statement that is composed of an informal language\(^1\) even if it may convey the intended meaning. An example of an informal specification could be Unified Modelling Language (UML) diagrams\([60]\) or natural language specifications. They are informal because the semantics of the language are not completely defined.

To understand formal verification an example of a software that shall sort a list can be taken. The requirements for that software is

“The software shall sort a list of natural numbers in ascending order”

To conduct formal verification on the software that implements this requirement a formal specification must be created. This requirement can be formalized in FOL as:

\[
\forall \text{myList} : \text{Array. }\forall i : \text{Natural. } \text{read}(\text{myList}, i + 1) \geq \text{read}(\text{myList}, i)
\]

Here \(i\) is of type \text{Natural} and \text{myList} is of type \text{Array}. These types sorts are defined within the structure (ref 2.3.2) associated with the FOL formula. The function ‘\text{read}’ is defined in the interpretation that takes a tuple of type (\text{Array},\text{Natural}) and returns an element of the array of type \text{Natural}. The function ‘\text{+}’ is also defined in the interpretation with some syntactic sugaring. The predicate ‘\text{\geq}’ is also defined in the interpretation and used here with some syntactic sugaring.

This formula essentially states that for all lists and for all indices \(i\) in those lists, the \(i + 1\)\(^{th}\) value of in that list will be greater than the \(i\)\(^{th}\) value.

This formal specification states the expectation from the software that would implement the list sorting algorithm. Now formal verification is when the implemented software and the programming language semantics are used to show/prove that the software will satisfy the formal specification. The formal verification step can be done using model checking or theorem proving both of which will be discussed.

### 2.6.2 Model Checking

Model checking [23] is a technique where a model of the system under analysis is taken and this model is checked against certain expected behavior/properties which are formally specified. Kripke structures can be used as models of a system(ref section 2.4) and Linear Temporal Logic (LTL) statements as desired properties.

One way to accomplish model checking is to use Kripke structures as models of the system under analysis and LTL statements to describe the properties to be checked. The model checking tool can then convert Kripke structures and LTL statements into Büchi automata. The Büchi automaton for the LTL statements is negated and then the cross product of the negated Büchi automaton is taken with the automaton from the

\(^1\)A language with no restricted syntax or grammar and with no concretely defined semantics
2 Background

Kripke structure to produce another Büchi automaton. If there exists an acceptance input sequence for this Büchi automaton then that input sequence shows a way in which the negated LTL property can be satisfied. This is a violation of the desired property that was stated by the LTL formula.

Currently one popular tool that is used for model checking of systems is the SPIN model checker [43]. It is used to construct a model of the system in a language called Promela\textsuperscript{1} and the desired properties can be expressed in LTL formulae. The LTL formulae are then checked on the model constructed in Promela and a counter example is also given when a property is violated. As discussed earlier both the model and the LTL properties are converted to a Büchi automaton and then checked to find an example where the LTL property is violated.

An interesting possibility that is exploited in this work is that, if in the end two Büchi automata are used for the model checking process then it does not matter if the Büchi automata is coming from a Kripke structure or an LTL formula. Therefore in this work, the specification effort is considered to be successful when a Kripke structure of desired properties is generated.

The biggest challenge with model checking is that the model that is being checked must be an accurate representation of the real target system. This was already discussed as a part of the introduction (ref Chapter 1). This challenge itself allows the application of model checking only for verification of algorithms and protocols which can be easily modeled. It limits the application of model checking to actual system software. Nevertheless, multiple research teams along with industrial collaboration have shown that model checking can be instrumental in finding protocol inaccuracies[57]. With this work, the objective is to be able to generate the formal specification against which a model will be checked. This work excludes the possibility to create formal models of systems directly from example data.

2.6.3 Theorem Proving

Theorem Proving comes from the mathematical domain where some given facts (axioms) or theories are used to prove or disprove a hypothesis. The theorem prover is supposed to provide a proof of support or contradiction. The theorem proving process can be automated, semi-automated or manual. In the case of formal verification the automated or semi-automated theorem provers can provide the most value because of lesser human effort required as compared to manual proofs.

One way of using theorem provers in the industry is through the GNAT toolsuite\textsuperscript{2}. The idea there is to translate the software to be verified into a formal model in the language of the theorem proving tool based on the syntax and semantics of the programming language and then use the desired properties as a hypothesis. The theorem prover shall

\textsuperscript{1}Process or Protocol Meta Language
\textsuperscript{2}https://www.adacore.com/gnatpro/toolsuite (accessed on 22.08.2019)
now attempt to prove or disprove the hypothesis and as a result provide a proof. First order logic statements can be used to create both, the formal model and the hypothesis. The GNAT toolsuite from AdaCore allows programmers to write software code in the SPARK language (a subset of the Ada language) which is internally converted to a Why3 [20] model of the software. The programmers can now also write the desired properties in SPARK and the toolchain then attempts to prove the desired properties over the model of the software.

This work targets the creation of FOL properties because a property in any formal language can generally be translated into any other formal language. The expectation here is that an FOL property generated by the system created as a part of this work could directly be converted to a desired property in SPARK (called a SPARK contract). Therefore the generation of an FOL property is considered a successful completion of the specification effort.

### 2.6.4 Need of Formal Verification

The most common argument that is given in favor of formal verification is that formal verification allows us to detect problems early on in the development process and therefore it saves costs of rework later[42]. This argument is weak because it is very difficult to prove that by doing formal verification some rework was avoided which would have happened if formal verification was not done. To conclusively provide proof of this argument 2 teams will have to perform the same project with and without formal verification to prove this. Even then the metrics of benefit vs costs may not be easy to compare objectively. For example it is likely that both techniques will identify different bugs and then it is a matter of opinion about which bug was more critical and which bug took more time to solve. This argument is not enough to justify the case for formal verification.

Instead, a different rational should be presented. Formal verification should be compared with testing and it should be shown that formal verification is just like another testing technique in terms of its outcome of detecting bugs if not better. Just like testing, where there are certain tests that are run on developed software and software is changed until all those tests pass, formal verification is also carried out over formal specifications and software, and the software is changed until the specifications are proven over the software. By doing formal verification we can generate the same verification evidence as with testing but faster because with a formal verification effort we can ‘test’ the software with classes of test cases grouped together by mathematical logic. The use of this mathematical grouping for a big group of test cases should be convincing enough to say that they are at least equivalent. Formal verification produces at least the same quality of output and the time that is taken to formalize is earned back because one specification statement can verify multiple classes of test cases. Another argument that goes along the same lines is highlighted in [17], which is that formal verification leads to a better exploration of the test space. This saves valuable time and should justify the use of formal methods.
The final argument that must also be presented is that functional safety conditions are a lot more natural to express in a formal language than in a test vector. For example a test vector that simulates a car being detected in the front of the car being tested would be fixed in the test case like “if distance sensor value is 19.8 meters then car should slow down until reading of distance sensor is 25”. If a formal specification of it was written it would read somewhat like “If the distance sensor gives a reading of less than 20 meters then slow down the car until the reading of the distance sensor is 25 meters”. The formal specification is more close to the real requirement and represents the requirement more closely. It would take multiple test cases to convey the same meaning as the formal specification. Please note that both the above examples were informal translations of what a test case vs a formal specification would read like, but they convey the point.
3 Related Work

This work presents an approach to generate formal specifications from natural language requirements that are supplemented with example data illustrating the requirement. This work touches the requirements creation process, usage of example data within the requirements and generation of formal specifications from example data. To understand all the areas that are touched upon by this work, first, the approaches that propose methods of improving the requirements during the requirements process are discussed. Then development strategies that use example data are discussed. These strategies are then used to demonstrate that it is possible to apply the work proposed here within an industrial setting. Finally, the approaches where formal specifications can be generated from example data in different contexts are discussed.

3.1 Improving Quality of Requirements

Many scientific documents that talk about the quality of software requirements state a variation of the statement that “most software requirements are written in natural language”[33][34][16][15][64]. This notion is also quite common in the industry. Natural language requirements are one of the easiest ways of expressing software (even system) requirements, even though this allows ambiguities, incompleteness or inconsistencies to exist in the requirements. The general idea discussed in this section is that natural language requirements can be analyzed and improvements can be suggested to the requirements engineer writing the requirements. The advantage of this approach is that the requirement engineers are not severely constrained in expressing themselves as the suggestions do not require them to change the language of the requirement itself, but to make the language more clear.

Génova et al. in [38] proposed various quality metrics for natural language requirements. These metrics should guide the requirements engineer to produce clearer and better quality natural language requirements. Kummier in [52] beautifully summarizes the work around requirement quality metrics that can be used to improve the quality of natural language requirements. Quality metrics for natural language do not introduce any formalism within the requirements. These metrics improve the natural language itself. On the other hand, this work focuses on getting formal specifications from requirements thereby formalizing the requirement.

In [14], Berry et al. categorize Natural Language Processing (NLP) tools into four categories and present an analysis comparing these tools. These tools accomplish different
things like checking of weak phrases and ambiguous requirements or generating models from natural language descriptions, generating traces between requirements and other artifacts in the project. They conclude that NLP tools can at times be disadvantageous to use for high dependability systems. This is because “the fundamental limitations of NLP means that 100% recall is impossible” [14]. This shows that improving the quality of requirements using NLP approaches is unreliable. Therefore the work presented here focuses on improving ways to formalize the requirements.

Another way of improving quality of requirements is by forcing the requirements engineer to think in patterns of pre-defined type and structure. These patterns can be in terms of sentence structure and in terms of templates of models in SysML/UML. Denger et al. in [33] propose various patterns that are very close to natural language but are structured with a meta-model to ensure removal of ambiguities in a nearly natural language requirement sentence. The meta-models are domain specific and each model element results in at least one pattern. In an ideal scenario the requirements engineer can choose a pattern based on what s/he wants to describe and then that pattern tells what all questions must be answered by the requirement sentence. In the case illustrated in the paper, already available natural language requirements were analyzed based on certain rules and when violations were found, the requirement was re-written based on the pattern that the requirement fit in. These if implemented in a tool may force the Software Requirements Engineer (SRE) to think in patterns and thereby answer all ambiguities that would have remain unanswered if the engineer was not guided. This minor restriction could improve the quality of requirements as long as it can be guaranteed that patterns for all possible requirements in a domain exist. This approach although valuable, might not be easily useful in all domains due to unavailability of patterns for those domains. In contrast, the approach presented in this work can be applied without restriction in multiple domains, although the discussion of this is not within the scope of this work.

Mavin et al. with EARS (Easy Approach to Requirements Syntax) [58] also do the same but without a clearly defined meta-model for each domain. They propose five patterns that are used to classify all requirements. It is the job of the requirements author to identify what pattern is applicable to what requirement. As compared to the previous work, it feels easier for the SRE because there are lesser amount of things that the author has to keep in mind. This may improve the overall quality of requirements it still does not provide a way to get formal specifications from the requirements process which is one of the goals of this work. Getting formal specifications from the requirements process is explored in the next section.

3.2 Formalizing Requirements

Developing requirement models has often been proposed in literature as will be seen in this section. An immediate drawback of any such framework is that it is a completely new skill that a requirements engineer must learn to be able to effectively use it. Also,
any such requirement model in a way shifts the work from a developer, who is more suited develop a formal model, to a requirements engineer who, traditionally, has the responsibility of understanding the customer requirements and documenting them. This drawback is common to the approaches presented in this section and will therefore not be pointed out again. A way to minimize this drawback is to make sure that the proposed framework is easy to use and can be quickly understood by the SRE. This work also proposes creating a model of the requirements using example data that illustrates the requirement, and in principle it is along the same lines as the work presented in this section. A difference that could easily be seen is that this work uses example data to generate the specifications which is not seen in the other approaches.

In [66] a requirement modeling framework KAOS, is used to generate formal specification in B (a formal language). The framework describes four sub-models that are used to model the requirements of a system. The four sub models are: goal, operation, agent and object. The sub models for example, model the goals of the system using a tree like structure where one goal is decomposed into sub goals. This way multiple models can be created for one requirement. These four types of sub-models can then be used to create a B machine for every agent which is a formal specification of the agent.

Similarly in [80], researchers propose that textual requirements be supplemented with scenario descriptions in Requirements Definition Language (RDL-2). These descriptions are then used to generate requirement models within RDL-2 and eventually generate formal specifications in the Z specification language[1]. The reason it is similar to the previous approach is because this approach also uses a custom language to partially formalize the requirement through the effort of the engineer. In comparison to both the above approaches, this work uses a smaller language to describe scenario data instead of the requirement. This is then used to generate formal specifications.

In [8], Autili et al. restrict the English language and grammar in a way such that they can express temporal logics in structured sentences. The requirements could be expressed in numerous patterns which then provide a structured english sentence which can be translated to temporal logics. The author has to make consecutive decisions to reach a pattern that is suitable for the requirement to be specified. The author can then specify the events or states in a system. These can then be selected to fit correctly in the previously selected pattern. As an advantage of this approach, various temporal logic formulae can be generated to represent the requirements. The author of the requirement is therefore restricted to use the plethora of patterns available within the system. This at times can be a very challenging task, much more difficult than simply describing the requirement. In comparison, with this work the engineer can write the natural language requirement and start specifying the requirement using example data almost immediately.

Da Silva in [73] proposes a Requirements Specification Language (RSL) that is rigorous and organized into a set of views and abstraction levels. This language was designed for the target audience of “business analysts or requirements engineers” to have a common language for sharing the system description. The authors recommend that analysts should read the textual requirement and then specify the requirement in the Requirement
3 Related Work

Specification Language. In doing so the analysts have to decide on the following things[73]:

- the events that may occur in the scope of the business
- type and participants for each event
- identify the main (top-level) processes, and then decompose them into simpler (low-level) processes
- type for each process, and the participant responsible for it
- specify the business behavior of the system in RSL
- define the respective type, stakeholder and priority for each business goal
- ... and others

The critical analysis of the requirements text that is expected from a business analyst and the number of decisions that they must take to formalize the requirements makes this task prone to mistakes. This approach relies on an analyst’s deep understanding of the domain of application, system requirements and the Requirements Specification Language itself. This essentially makes the task of an analyst very difficult. Essentially an analyst is responsible to express all requirements in a formal language. In comparison, this thesis proposes a smaller formal language that is can be used to illustrate the requirement through example data. This can then be used to generate formal specifications.

In this work, the primary goal is to make the creation of formal specifications fit within the requirements process. The engineer can write the requirements and illustrate those requirements with example data. This not only adds value to the informal requirements but at the same time can be a source of generation of formal specifications. This principle was also echoed in various development strategies that are discussed in the next section.

3.3 Software Development Strategies

Software development strategies refer to the process followed by developers to develop software. Test Driven Development (TDD) and Behavior Driven Development (BDD) are considered in this section because both these strategies advocate the creation of test cases before starting to write software code. This work also resonates with these strategies and proposes the illustration of the requirements with example data. This example data can be used to derive test cases.

It must be noted that with this discussion, the use of TDD or BDD is not being advocated for the avionic or embedded systems domain. Only the benefits of the creation of test cases before starting development and using those test cases at some point during the development process are pointed out. These benefits are also in line with the DO-178C recommendations and the V-Model. The discussion of the development strategies only presents an alternate way of representing test cases in a table along with the requirements.
This representation structure is also used in this work. With this section, the strategies of TDD and BDD are demonstrated and a connection between them and the proposed work is established.

3.3.1 Test Driven Development

TDD [12] is based on the basic principle of executing tests on the software system before the software actually exists. This means that the tests are developed first and the software afterwards. These tests fail because the software does not exist. Therefore, TDD as the name suggests drives the development of software using failing tests that the programmer has to turn to passing tests by writing correct program code. The mantra of TDD is “red/green/refactor” [12], which essentially means that you start from a test that fails (red), after some programming the test finally passes (green) and finally the refactor step is crucial to remove all the “sins” committed to make the test pass. V-Model in its traditional sense of having a testing and verification phase after the development phase may not allow including TDD within the V-Model. In theory, TDD can also be used in the embedded world as suggested by the available literature [75][39] and tools [10] that support it. TDD implies using unit tests to test software functionalities during the development cycle. The tests may or may not represent desired behavior of the system.

Although TDD does not directly contribute to the work presented here, it was discussed briefly because it inspired the creation of BDD which is discussed in the following subsection.

3.3.2 Behavior Driven Development

BDD was first coined by Dan North [63] which was born out of the need to properly implement TDD in an agile development process [11]. BDD is a process where developers start writing test cases in a language that is easily understood by business stakeholders. It allows describing a behavior of what the software shall do in a nearly natural language instead of writing a test suite in a programming language. It allows business stakeholders to validate what the software will do and allows the developers to derive test cases directly from the behavior description and test the software developed.

BDD advocates on writing tests that only describe the desired behavior of the system as opposed to TDD. This means that with BDD, tests are created to test the functionality of the system, this implies robustness testing\(^1\) is not included. BDD also focuses more on the harmony between business stakeholders and developers. By writing tests for desired behavior in a structured natural language, it allows business stakeholders to get involved in the testing process and validate the tests early on in the development process. TDD on the other hand, was more meant for developers only, which discouraged any

\(^1\)a testing technique where the unexpected or invalid inputs are provided to a software or the software is operated in a stressful environment
involvement from the customers or business stakeholders. This was because the tests
created as a part of this process were in programming languages. Therefore the business
stakeholders could not really validate the various design decisions that were taken during
the creation of the test cases. The communication with the stakeholders did not take
place till the validation phase and this was too late. This increased the time required to
deliver satisfactory software. BDD tries to solve exactly this problem.

Behavior Driven Development (BDD) is discussed here because like BDD this work
adopts the workflow in a Behavior Driven Development (BDD) process. An added
advantage of BDD and the tools around it is that they are focused on deriving a test
suite out of a behavior description but it is important to note that deriving test cases was
not the primary intention of BDD but was a possible outcome of the tools that realized it e.g Cucumber[29]. A BDD feature file is shown in Figure 3.1b. This feature file is used
to describe a small functionality of software. It also establishes, using the “Scenario”
description, the exact steps that should be taken to accomplish the objectives of the
described feature. As can be seen the scenario description is some form of structured
English where there are keywords like Given, And, When and Then. The Given word
is used to establish pre-conditions for a scenario. The When keyword in combination
with the Then are used to describe an event and a response respectively. The Examples
section of a feature file can be used to generate test cases.

3.3.3 Specification-by-example

“Specification by example” [3] is essentially a BDD strategy with the only addition that a
requirement specification will have some examples concerning the values that a variable
takes in the context of that requirement. The objective here is to show how do the
variables of a system change in the context of a requirement. An example in figure 3.1a
illustrates this.

Although all information presented in Figure 3.1a is informal it clearly conveys what we
want the door to do. This concept can be extended to make the same example more
formal like in Figure 3.1b.

Figure 3.1b shows the Gherkin [30] syntax which has started seeing applications in the
software industry. A key feature of Gherkin is that the specification feels like a natural
language which is extremely important to ensure adoption. Along with this, Gherkin
allows the specification of examples within a feature file which can then be used to
generate a full test suite [29].

The work in this thesis builds upon the concept of having examples along with the
requirements as with specification-by-example. This work would require the person
writing the requirements to add example data that illustrates those requirements. Every
example data table will contain one column extra other than the variables described. This
column will provide some information that is needed to generate formal specifications
from the data table.
### 3.4 Generation of Formal Specifications from Example Data

Generating formal specifications from example data has already been explored in different contexts in the literature. One essential feature of this technique is that the example data is used as a ground truth and a formal specification that is true for that ground truth is generated.

One interesting example of this approach is seen in [19]. Here the user was expected to perform computations on numbers on a computer system which recorded these computational steps. The system then generated a computer program that would do the same or optimized computations to produce the same output. This work was done in 1976 and it shows that generating programs from examples has existed in the literature for a long time. Another similar example was [77] where input and output lists were provided to a software system which was used to generate a LISP\(^1\) program that implemented the algorithm that the input-output examples represented. In [6] researchers used inductive inference along with data metamorphosis rules to produce a Prolog program. Conceptually it is similar to [77] except that here they used formalized semantics of the Prolog language to construct the Prolog program. In comparison, this work aims to generate formal specifications from data in formal specification languages instead of programming languages.

---

\(^1\)LISP is one of the oldest programming languages. The name is derived from LISt Processor. The major data structure in the language is lists. Read more at https://lisp-lang.org/
3 Related Work

A different direction of research was done in [26] where the objective was to get program specification from multiple views of a database. A View of a database is a presentation of only certain column headings in the database based on the conditions specified in the query for that view. This work constructed a specification for each view of a database. This work is especially relevant to the work presented here because they use relationships between the data items in the view to generate specifications of how that view was created. On similar lines, this work also uses the relationships between column headings to generate formal specifications. One key difference that can be easily noticed is that the objective of this work would be to generate temporal properties\(^1\). Specifications of database views will not capture an evolution of a system and therefore the temporal properties for a database view would not be relevant.

Specification Mining [55] is another direction of research which deals with using program code or execution traces of program code to formulate formal specifications of the program behavior. This area is widely explored in [53] [54] [7] to find a good way to generate formal specifications that are temporal or related to data flow. This validates the work presented here because if execution traces can be used to generate formal specifications then an example data table given by the requirements or tests engineer can also be used to generate formal specifications.

\(^1\)properties which deal with changes in the state of a system and the order of these changes
4 Running Example: Queue Timeout

In this chapter, a situation of a requirements engineer is described to understand the steps that are typically taken for assigning software requirements to software components\(^1\).

At this point, the software requirements already exist in natural language. The requirements engineer then chooses to provide certain examples to illustrate the natural language text. This is done such that the software developer can understand what is intended by the requirements. Giving examples, quickly clarifies what is expected of the software component. It is also possible that the developer derives (or develops) test cases from the requirements. The developer then starts developing the software component.

This process is quite similar to the process supported by the Cucumber tool (ref section 3.3.3). Within this process it is envisioned that, the formal specifications can be derived from the example data that illustrates the requirements. This is the main goal of this work.

This chapter discusses the above process with respect to a specific example that is relevant for both embedded systems and the avionics domain. This example was inspired from the work done as a part of the Verified Indoor Tactical Avionics Squad (VerITAS) project [2]. The VerITAS project is used to demonstrate internal research and technology developments at Airbus. It uses multiple hexacopters and is intended to support flight in indoor or Global Positioning System (GPS) denied environments. Every hexacopter has a mission computer which runs software responsible to issue flight commands to the flight control computer. The example taken in this chapter is inspired from the requirements of a software component that runs on this mission computer.

4.1 Requirements

The software component that is discussed here is a health monitoring component that receives heartbeat\(^2\) messages from other software components. If no heartbeat messages are received on the message queue, for 10 seconds then it is assumed that the message queue is faulty. A reset must be done where the message queue is closed and re-opened.

---

\(^1\)software components are defined as “Reusable software components are self-contained, clearly identifiable artifacts that describe and/or perform specific functions and have clear interfaces, appropriate documentation and a defined reuse status” [70]

\(^2\)alive components send periodic messages to convey health status. An absence of a heartbeat message for a fixed period of time may indicate failure of a component.
This software component is also responsible of doing certain other computations that must not be done if the message queue is closed because this indicates a problem.

A set of informal natural language requirements can be created for such a software component. It could be noted that the set of textual requirements presented here are incomplete. One reason for this is that often times requirements are incomplete in the industry. Another reason is to keep the example short. The concepts presented as a part of this thesis can be applied to an extended set of requirements as well. Therefore only a small set of requirements are taken for this example. The proposed process will be applied to only this set of requirements. The set of requirements are listed below:

1. The software component shall open a message queue called ‘healthQ’ and receive messages of a pre-specified format\(^1\).

2. The software component shall not do any computations if the message queue is closed, since this indicates a reset state and results of the computations may be faulty at this time.

3. If a message is received within 10 seconds then the message queue must not be closed.

4. If no message is received for 10 seconds then the software component shall close the message queue and then re-open the message queue so as to reset the buffers of the message queue.

5. The message format only contains one byte. The software component shall process the message by copying the message byte to the ‘dataValue’ variable in memory only if the 10 second timeout was not elapsed.

A question that can be raised for the third and fourth requirement could be: how is the 10 seconds interval defined? Is it 10 seconds from the start of the system? Is it from the last received message? This is called ambiguity. The natural language requirements have this ambiguity when the requirements are poorly written. As will be seen, this ambiguity gets cleared up when an example data for these requirements is provided. The example data, in some way, captures the exact thinking of the requirements engineer rather than the abstract idea that is conveyed in a set of natural language requirements.

Example data depicting these requirements is shown in Table 4.1. This data is called scenario data because it depicts the scenarios of how the system shall behave according to the requirements.

It can be observed that the ambiguity is resolved. The two cases illustrate that the message queue must be reset only if no messages have been received for 10 seconds from the time of the last received message. The headings of each column are explained as follows:

\(^1\)The format and capacity can be described in other requirements. These requirements are not considered simply to shorten the scope of this example.
4 Running Example: Queue Timeout

<table>
<thead>
<tr>
<th>healthQopen</th>
<th>healthQdata</th>
<th>dataValue</th>
<th>time</th>
<th>canProcess</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>89</td>
<td>89</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
<td>45</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>NULL</td>
<td>NULL</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>NULL</td>
<td>NULL</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>NULL</td>
<td>NULL</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>39</td>
<td>39</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.1: Example data to illustrate requirements

- **healthQopen**: 0 if closed and 1 if open.
- **healthQdata**: The message data that arrives at the message queue.
- **dataValue**: The software simply copies the data received to this variable only if **canProcess** is 1.
- **time**: Denotes the passing of time in seconds. Resets to zero when message queue is reset and if a message is received.
- **canProcess**: Only 1 if **time** is less than 10, 0 otherwise\(^1\).

With a quick glance the software developer understands requirements and the intended behavior clearly. The table can also serve as a source of acceptance tests if required.

---

\(^1\)There is no strong rational about why the requirements engineer chose to create this variable. This variable is retained simply to demonstrate the flexibility of the process.
4.2 Available Information

To understand what can and cannot be included in a specification we must understand what information is available to the requirements engineer at this point. Following is the information that is available to the requirements engineer at this point:

- This set of software requirements are low-level and come from a higher level of system requirements. The exact traceability of these requirements is available and can be known if needed.
- The requirements engineer knows the names assigned to message queues and other data fields since this is documented using an ‘Interface Control Document’. This document specifies the names and ranges of data items that will be shared between different components of a system. It also specifies the format of this data if applicable.
- From requirement 3 and 4, the engineer knows that 10 seconds is an important value and around that value actions must happen to the message queue.
- From requirement 2, S/he also knows that no processing should happen if the healthQ is closed.
- From requirement 2 and 4, S/he also knows that some ‘reset of buffers’ operation must be done. A result of this operation should be to close and then open the queue. This could be a specific state of the system.

This knowledge that is already known can be used for specifying the component and the expectations from it. This work will focus on using this available information that the requirements engineer already has and express it within the requirements document so that it becomes a part of the ground truth.

4.3 Formal Specifications

To carry out formal verification of the developed software component, formal specifications are needed. For this example the following formal specifications can be created.

The following formal specifications in FOL can be created for this example:

\[ \exists \text{time}. \exists \text{canProcess}. \text{time} = 10 \land \text{canProcess} = 0 \]
\[ \forall \text{time}. \forall \text{canProcess}. \text{time} < 10 \implies \text{canProcess} = 1 \]
\[ \forall \text{healthQopen}. \forall \text{canProcess}. \text{canProcess} = \text{healthQopen} \]

The first FOL statement states that, there exists a value of time for which there exists a value of canProcess for which the value of time is equal to 10 and the value of canProcess is equal to 0.
The second FOL statement states that, for all values of the variables \( \text{time} \) and \( \text{canProcess} \), if the value of \( \text{time} \) is less than 10 then it implies that the value of \( \text{canProcess} \) is 1. This also means that if the value of \( \text{time} \) is not less than 10 then \( \text{canProcess} \) may or may not be 1.

The third FOL statement states that, for all values of \( \text{healthQopen} \) and \( \text{canProcess} \), the values of \( \text{canProcess} \) are equal to the values of \( \text{healthQopen} \).

The following Kripke structure could also be created that demonstrates a ‘reset’ state and a ‘normal’ state. The normal state would mean that the system is either receiving the messages as expected or it is not receiving the messages but the timeout period has not elapsed. The ‘reset’ state could mean that the system has now closed the message queue because the timeout elapsed and no message was received within the timeout. The Kripke structure describes this behavior. The red arrow indicates the initial state for the Kripke structure (in this case **Normal**).

![Kripke Structure](image)

Figure 4.1: A Kripke Structure for the running example

This work shall focus on expressing the scenario data table 4.1 in an intuitive format and then generate formal specifications in first order logic and Kripke structures. This work shall be evaluated using this example by assessing the outputs that are produced by the proposed system.
5 Automated Property Formation

The requirements allocated to software are the only source of ground truth about what the system should do. The problem with this ground truth is that it is subject to interpretation because it is in natural language. Sometimes the interpretation of the software developer may not align with the requirements engineer.

To strengthen this ground truth various approaches were considered in chapter 3. These approaches attempted to remove the ambiguities in the requirements by asking the requirements engineer to provide clarifying text. In another set of approaches, the requirements engineer or some analyst would formalize the requirements in a requirements model. By carrying out this process, ambiguities, inconsistencies and incompleteness could be checked automatically. As a summary, the approaches added more information to the requirements to clarify the requirements and thereby strengthening the requirements.

On similar lines, one approach, to strengthen the ground truth of requirements, could be to add example data to the requirements that illustrates the meaning of the requirement. This example data, when provided by the requirements engineer, captures the exact interests and intentions of the requirements engineer. This strengthens the ground truth by giving a concrete representation to the natural language requirement. If a system would exist that could use the example data to generate formal specifications, then the source of the formal specifications could be rooted in the requirements document. This would make the formal specification represent the ground truth much more closely as compared to a formal specification that is written by a human after interpreting the natural language requirement. This approach is presented in this work and the system is developed as a part of this work.

It was shown in section 3.3.2 that providing example data is a common practice in certain development strategies. The requirements engineer can write the requirement in natural language and then supplement it with example data that will illustrate the requirement with certain variables. This example data is now a part of the ground truth and it can be used to generate formal specifications that can help in the formal verification effort at later stages.

In this work, a systematic way of using example data as a source of ground truth is presented. This work describes how should the example data be created. It also describes the additional input from a requirements engineer that could be used to guide the generation of formal specifications. This additional input is called the “relationship model”. It will be shown why this additional input is needed. Within the scope of this work is also the implementation of the “property generation system”. This system uses
the example data and the additional inputs to create formal specifications. These formal specifications are generalized and referred to as properties of the system.

The formal specifications generated by this system should be in formal languages that can be used in the industry for formal verification efforts. These languages are first order logic and Kripke structures. In section 2.6, the application of these languages is discussed in the context of the industry.

In this chapter the proposed approach is described. This covers the inputs and the outputs to the proposed system. At first the system architecture is described to provide a high level overview of the system. The inputs to the system are then discussed in detail. Algorithms of creation of the candidate properties and formal specifications as FOL formulae and Kripke structures are then presented. Finally, observations about the generated outputs and the limitations of them are then discussed.

5.1 System Architecture

This section presents a high level system architecture. It is used to introduce various components of the approach. In figure 5.1, The INPUTS on the left and the OUTPUTS on the right can be seen. The green box in the middle labeled as “Automated Property Formation” simply converts its INPUTS to its OUTPUTS.

Two types of INPUTS exist.

The blue denotes the background knowledge which is pre-existing information. This means that it is assumed that the background knowledge will exist before the example data is being inputted by the user/requirements engineer. The background knowledge can be of two types, general mathematics and domain specific. This is discussed in the section on inputs.

The other type of INPUTS are the user-inputs. The user inputs are the “Marked Scenario Data” and the “Relationship Model”. The marked scenario data is the example data (ref section 4.1). As the name suggests this example data is “marked” and represents a “scenario”. The relationship model, models the relationships between two or more data variables that are used to illustrate the requirement in the scenario data. These will be discussed further in the section on inputs.

The only OUTPUT that is expected from the system are the formal specifications. These will be in the form of first order logic formulae and Kripke structures.

A UML data flow diagram is shown in figure 5.2. This figure shows the transformation of the inputs to the system to the outputs. At first glance we can see the process “Automated Property Formation”. The user creates the marked scenario data and the relationship model. The background knowledge is also input data but it is imported as it is into the property formation process. The detailed description of these inputs is done in the following sections. The process 1.1 uses all the inputs to produce propositional logic
5 Automated Property Formation

5.1 Formulation

Figure 5.1: Structure of the proposed system

formulae that are true for the marked scenario data. This is done by first generating all possible propositional logic formulae and then validating them with the scenario data to obtain the true set of propositional logic formulae.

This set is used for generating Kripke structures and first order logic formulae. The true propositional logic formulae are processed by process 1.2 to create Kripke structures. This process uses a “marking syntax and semantics” along with propositional logic formulae. The syntax and semantics of markings will be discussed in the following sections. The true properties are processed by process 1.3, which performs 3 types of generalizations on the propositional logic formulae to generate first order logic formulae.

This data flow diagram was only included to show the coarse flow of data within the system. This provides a basic overview of the system. The inputs to the system are discussed in the next section.

5.2 Inputs

The inputs to the proposed system are briefly described below:

- **Background Knowledge** - It is a set of predicate definitions that are used to represent some mathematical or domain specific operator. They can be used to represent a condition of the system. Each predicate can be evaluated to true or false based on the assignments to it.

- **Marked Scenario Data** - This is a small data set that is used to express the requirements in terms of changes in values of variables of a system. The variables of the system are arranged as the columns of the scenario data table and are therefore referred to as column headings or data headings in all the future text.

- **Relationship Model** - It is used to express the relationship between the data headings in the scenario data using predicates from the background knowledge. Each relationship model only states about what predicates may be related to what
all column headings. It does not describe the exact nature of the relationship. Each of these inputs will be discussed in detail.

5.2.1 Background Knowledge

The background knowledge is a set of predicate definitions. Examples are shown in Figure 5.3. This is the actual file of background knowledge that was used to conduct tests with the system. The syntax of the file was formulated in Backus-Naur Form. Since the syntax itself is only an implementation detail it is not described in this chapter. As can be seen, the predicate name comes first followed by the argument list in round brackets separated by commas. This is then followed by a number which denotes the number of arguments. This number was put there to have it as a sanity check for the system. This number serves no higher purpose. It can be removed in a future version of the system. After this, two colons are used as a separator to start the predicate definition. The predicate definitions can only consist of atomic boolean formulae that are connected using boolean operators AND(&&), OR(||) and NOT(!).

For example, in figure 5.3, the predicate isPositiveResult in line 9 is used to check if the result of a subtraction operation is positive or not. According to the definition, the predicate has 3 arguments just like the minus predicate in line 2. The first 2 arguments are subtracted and the result is compared with the third argument. If the result of the subtraction is equal to the third argument and the result is greater than zero, then the
predicate evaluation returns true.

![Background Knowledge](image)

**Figure 5.3: Example: Background knowledge**

For every input file certain rules are checked for that file. These rules are checked by the software implementation as a part of this work. The following rules are checked for the background knowledge file and errors are reported if these are violated:

- The predicate definition can only contain the variable names defined in the argument list of the predicate.

- The number of arguments in the arguments list and the number between the ‘,’ and the ‘::’ should be the same. This is a sanity check.

The predicate definitions can be compared to mathematical functions that only return a boolean value. Currently, previously defined predicates cannot be used to define new predicates but this is natural next step of improvement. Another improvement could be to include types in the predicate definitions. These types could also be data structures like stacks or queues. Currently, the predicate arguments can only be assigned decimal arguments.

**Domain Specific Background Knowledge**

The background knowledge can be divided into Domain Specific Background Knowledge (DSBK) and mathematical background knowledge. This distinction is conceptual and the example of the *isFlying* predicate in Figure 5.3 illustrates this. Typically DSBK predicates are expected to be more complex than general mathematics because they are expected to use multiple concepts (addition, subtraction, comparison etc.) from general mathematics and combine them to form a predicate. The combination of these concepts is illustrated by the *isFlying* predicate in line 10 but the combination of multiple predicates to form a complex predicate is currently not supported by the system.

DSBK predicates can either be created by a domain expert who needs certain predicates for the specification effort or they can be created as a result of previous specification
effort done by this system. The formal properties generated in one specification effort can be converted to DSBK for other runs of the specification effort. Supporting this is out of scope for this work.

5.2.2 Marked Scenario Data

Marked scenario data is scenario data (ref section 4.1) that has an additional “Marking” column to it. The scenario data is associated with the requirements that it describes. The engineer creates certain variables that represent relevant information of the requirement. These variables are then arranged as in table 4.1. The only additional step is the addition of the marking column and entering the marking for each data row.

An example of marked scenario data is shown in figure 5.4. The markings are present in column A and the data headings are in cells B1-F1.

The figure illustrates all requirements in Chapter 4 through example data. It uses the marking syntax and semantics that will be proposed as a part of this work. The key observations that can be seen in the marked scenario data are:

- The row 13 separates two cases, as were shown in table 4.1. The bottom set of data refers to the case where a reset occurs and the top data refers to the case where the reset does not occur. Each set of data separated by ‘_____’ is called a data sequence.

- healthQopen is zero in the 19th row because the time has reached the value 10 and no message is received according to healthQdata.

- healthQopen remains one in the 12th row as a message is received before 10 time units have elapsed without a message.

For every input file certain rules are checked for that file. These rules are checked by the software implementation as a part of this work. The following rules are checked on the scenario data file and violations are reported:

- The cell A1 should always have the title “Marking”.

- All cells in the first row (B1-F1) should be non-numeric i.e. the data headings should not be numeric. This means that they can start or end with numerals as long as they have some non-numerals. Example: legal: 9gag, var1; illegal: 9678,223.

- Marking column should be according to marking syntax and semantics defined in section 5.4.

Marking

The “Marking” column contains the additional information about the data that is needed to generate formal specifications from it. This additional information is called the
Automated Property Formation

Figure 5.4: Example: Marked Scenario Data

"Marking". Every marking marks a data row with relevant information about that data row.

The marking for one data row captures the context, order, and the grouping of this data row with respect to the other data rows. The well defined syntax and semantic rules for an entry in the marking column allows the engineer to capture this information in the marking. A major contribution of this work would be the definition of the syntax of the markings and the semantic rules that restrict an entry into the marking column.

The three aspects of a data row that are captured by every marking are described as:

- **Ordering**: The scenario data when presented in a table is implicitly ordered from the first row to the last row. This ordering is implicit if no marking is provided. The marking makes this ordering explicit and at the same time allows representing a cyclic order. The marking therefore captures the ordering of data and extends the possibilities of this ordering.

For example. In figure 5.4, the cell A6 A7 and A8 illustrate an ordering from `normal — rest — normal`. This is referred to as a cyclic ordering.
5 Automated Property Formation

- **Grouping**: In a scenario data, more than one row can represent a similar condition or property that is obeyed by a system. This grouping of more than one data rows that represent a similar condition/property of a system can be made explicit by the marking. A “name” can be used to denote this group of markings. This is called a property group.

  For example, in figure 5.4, the cells A3-A6 show the grouping of a “normal” condition of the system.

- **Context**: Since every data row is ordered and can be grouped in some way, there is a context to the existence of each data row. The rows that come before and after may or may not provide context to the values of a specific data row. This context is made explicit by the property group and the step value in a marking. A step value carries the information of the number of data steps this data row is away from the 0th data step of its corresponding property group. This is the context that is made explicit by the marking.

  For example, in figure 5.4, the cell A6 carries the context for the cell A7. The reset only happens if immediately before it the time value is 9 and no message is received.

The above stated information should be explicitly stated in the inputs to generate formal specifications from it. This is done by the marking column and a specific marking syntax and semantics. These will be discussed in section 5.4

### 5.2.3 Relationship Model

The relationship model describes the relationships between data headings that the engineer found interesting for specification. This is the place where the interest of the engineer is expressed. A template for a relationship model is shown in figure 5.5.

\[
\langle \text{dataHeading/constant} \rangle, \langle \text{dataHeading/constant} \rangle \rightarrow \langle \text{predName} \rangle, \langle \text{predName} \rangle;
\]

where

\[
\langle \text{dataHeading/constant} \rangle : \text{any column heading of the marked scenario data table except for the marking column or a constant value like 0,1 or 3.14}
\]

\[
\langle \text{predName} \rangle : \text{any name of the predicate defined in the background knowledge}
\]

\[
[ \text{} ] : \text{denotes the 0 or more occurrence of the enclosed format}
\]

Figure 5.5: Relationship model template

A relationship model not only expresses the interest of the engineer it also reduces the formal specification generation task by a significant magnitude. This reduction will be
discussed during the chapter on evaluation (ref Chapter 7). The relationship model therefore makes the formal specification effort focused towards the goals of a requirements engineer and also reduces the time to generate formal specifications. This two-fold benefit justifies the use of a relationship model in this work.

The relationship model is a very weak model, in the sense that it does not give a lot of precise information. It only expresses that some data headings are related to each other by a set of predicates. Since it is a weak model it can have many sources, some of the possible sources for a relationship model are:

- State charts - can be used to extract information about the operations that are occurring on data items within the state chart to construct the relationship model.
- Activity Diagrams and Sequence Diagrams - same as state chart. If there are certain operations being carried out in conditionals then this information can be used to create the relationship model.
- Real system code – Actual software code can be used to extract the operations carried out on the variables in the software code and these can be used to create the relationship model.
- Simulations – Essentially code that models a system - Same as system code.
- Simulink Specification models – These express the exact relationship and computations carried out on variables or data items, therefore they can also be used to generate a relationship model.
- If none of the above exist then a relationship model can be made by the human itself because it does not have a lot of rules and has extremely simple semantics as described earlier.

It must be noted that the exact creation or extraction of a relationship model from the above mentioned sources in not in the scope of this work. Therefore it is assumed that the relationship model is currently provided by the requirements engineer.

The relationship model file is also checked for consistency with the background knowledge and the scenario data file. This is needed because the relationship model relates the data headings in the data file to the predicates in the background knowledge file. The following rules are checked in software and errors are reported if these rules are violated:

- All predicates referred in the relationship model should exist in the background knowledge.
- All data headings referred in the relationship model should exist in the scenario data file.
- The number of data headings in a relationship model are not enough for at least one predicate in the relationship model. Example: a predicate *plus* requires at least 3 arguments but in the relationship model there are only 2 data headings provided.

56
Two examples of relationship models are shown in figure 5.6. Each line is called a relationship model. The first line of the top example states that `healthQopen`, 0, `canProcess` and 1 are related to each other by the predicate `equal3`. The last line in the bottom example states that data headings `major`, `minor` and numbers 5000 and 1 are related to each other by the predicates `plus` and `greaterThan`. For every predicate in a relationship model multiple assignments of the selected data headings is possible. All possible combinations of these assignments are generated and considered for creating the set of candidate properties.

**Candidate properties**

Candidate properties refer to the propositional logic formulae that are not yet evaluated on the scenario data for a truth value.

All three files are used to create a set of candidate properties. As can be seen in figure 5.4 every row for a data heading has its individual marking. This means, for every data heading, multiple markings exists. For each data heading + marking combination, more than one value in the scenario data table may exist. When a candidate property is created it must consider all possible markings for each data heading.

**5.3 Creation of candidate properties**

Considering the relationship model in the first line of the top example in figure 5.6, the predicate `equal3` requires three arguments. `healthQopen`, `canProcess` and 0 can be chosen as these arguments. Since not all arguments are data headings we therefore refer to these as “tokens” from here on. A candidate property generated from these tokens will not be `equal3(healthQopen, canProcess, 0)`. Instead every token will be “instantiated” for all the markings that exist in the scenario data for that token. Since for every data heading multiple markings exist, therefore each data heading can be instantiated.
with a marking. For constant value arguments (like 0) no instantiation is needed. An
instantiated data heading is denoted by <dataHeading>.<marking>. Therefore one of
the candidate properties will be equal3(healthQopen.Start+0, canProcess.normal+2, 0).
The Start+0 and normal+2 are the chosen markings for this example candidate property.
Simply by changing the marking part of an instantiated token for each chosen data
heading, multiple candidate properties can be constructed for the chosen data headings.

The order and choice of tokens can be changed to create different candidate properties.
All possible combinations are created to form the complete set of candidate properties.
For example, equal3(healthQopen.Start+0, 0, canProcess.normal+2) will be a different
candidate property than the previous one because the constant 0 is the second argument
instead of the third.

The instantiation with a marking is done because it is of interest for the engineer to
know if a predicate is true at all markings for a set of data headings or only true at
some markings. Every instantiated token can be assigned a concrete set of values from
the scenario data table. These values can be more than one in case an exact same
marking occurs more than once in a scenario data table. For example, the marking
normal+2 occurs 3 times in figure 5.4, therefore each data heading will have 3 values
when instantiated with the marking normal+2.

As can be observed this creates a property bank of propositional logic formulae that can be
simply evaluated on the scenario data by substituting the value of the instantiated token
in the predicate and then evaluating the predicate according to its predicate definition.
Hence a true or false evaluation of each candidate property is obtained and this forms
the set of true properties that will be used for the generation of formal specifications. It
must be noted that a true property must be true for all values of the instantiated token
according to the scenario data table.

With the above creation process, it is observed that the creation of propositional logic
formulæ is a finite process. The algorithms used to create FOL formulæ must also focus
on creating finitely many formulæ from this set. It will also be seen that the algorithms
of creation of a Kripke structure, by its very nature, will be finite since they are based
on the syntax and semantics of the marking column.

5.4 Marking

As defined earlier a marking marks every data row in the scenario data table. It is used
to capture the implicit context, grouping and ordering of data rows. The definition of
the marking syntax and semantics is a key contribution of this work. It provides a clear
syntax, that, along with a few defined semantic rules allows the engineer to describe
system behavior.

The marking syntax and semantics are described in this section.
5 Automated Property Formation

5.4.1 Marking Syntax

The marking syntax was defined in a Backus-Naur Form grammar. This grammar is shown in figure 5.7. It was created with the tool GOLD parser builder [28]. This tool also generates parsing files that allow to parse a specific syntax in almost any programming language. GOLD engine for Java [45] was used as the parsing engine for all input files. This parsing engine parses the text according to a defined grammar and informs the user if syntactical errors are encountered. For the scenario data file, the marking column is extracted by the software implemented as a part of this work. The complete marking column is then parsed according to the grammar file presented in figure 5.7. The grammar file starts at line 9 and goes till line 50.

Figure 5.7: BNF grammar for marking syntax shown in GOLD parser builder

Syntax description

In this section, the marking syntax in figure 5.7 is described.

On lines 9-13 various tokens are defined, an example of each in order(9-13) is: lambda, 5, Start, end, ?.

On lines 15-16 the structure of any ‘Undefined Marking’ is defined. An undefined marking must always start with a ‘?’ symbol. The terminating symbol ‘;’ is needed for the parser to parse the markings but this is added in the software implementation before the parsing step and therefore is not needed in the scenario data file.

On lines 18-21 a ‘PropGroup Marking’ is defined. A propGroup marking can either be a numeral or must start with an ‘Id’.

On lines 23-24 a ‘PropGroup Marking Set’ is defined. A propGroup marking set is either made up of one propGroup marking and nothing more OR it is made up of one propGroup marking followed by another propGroup marking set. Evidently, this is a

\[\text{Syntax description} \]  
In this section, the marking syntax in figure 5.7 is described.

On lines 9-13 various tokens are defined, an example of each in order(9-13) is: lambda, 5, Start, end, ?.

On lines 15-16 the structure of any ‘Undefined Marking’ is defined. An undefined marking must always start with a ‘?’ symbol. The terminating symbol ‘;’ is needed for the parser to parse the markings but this is added in the software implementation before the parsing step and therefore is not needed in the scenario data file.

On lines 18-21 a ‘PropGroup Marking’ is defined. A propGroup marking can either be a numeral or must start with an ‘Id’.

On lines 23-24 a ‘PropGroup Marking Set’ is defined. A propGroup marking set is either made up of one propGroup marking and nothing more OR it is made up of one propGroup marking followed by another propGroup marking set. Evidently, this is a

\[\text{1This is different than the tokens considered as arguments to predicates. A token in the parsing context is a syntactical unit of symbols.} \]
recursive definition and allows us to have a series of propGroup markings one after the other in a propGroup marking set.

On lines 25-26 an ‘Undefined Marking Set’ is defined. Similar to propGroup marking set it can be made up of one or more consecutive undefined markings.

On lines 32-50 an ‘Ordered Data Set’ is defined. This is defined under 4 cases based on if the data sequence contains a start or an end symbol. As the name suggests, the definition defines the order of the expected sets. This is discussed case wise:

1. No start, No end: when neither a start nor an end are present in the complete data sequence then the data sequence can only have the undefined marking set before the propGroup marking set. The reason of this is in the semantics of the sets which will be discussed later.

2. Start only, No end: when no end symbol exists then the undefined marking set can only occur before the start symbol and the propGroup marking set can only exist after the start symbol.

3. No Start, only end: when no start symbol exists then the undefined marking set can only occur before the propGroup marking set and the end symbol can occur after both.

4. Start and End both present: when both start and end symbols are present then an undefined marking set can only occur before a start symbol. The propGroup marking set can occur only between the start and the end symbol.

On lines 28-29 a ‘Data File’ is defined. This allows one scenario data file to have multiple ordered data sets (data sequences) separated by ‘___’ (triple underscores)

Semantics description

To understand the semantics of the markings, the type of markings that can exist are discussed.

In the marking syntax in figure 5.7, in lines 31-49, the tokens Start and end are used to separate the undefined marking set from a propGroup marking set. This is done because each set has a different semantic interpretation and it is intuitive to separate them during the creation of the scenario data itself. The start and end symbols are not given any other special meaning apart from being used to separate the two types of marking sets.

The meaning of both these sets are described as follows:

• PropGroup Marking Set : A set of propGroup markings is used to describe ordered behavior. This can be understood as describing a system which changes its states in a very specific order. This order can be expressed by the propGroup marking set.
• **Undefined Marking Set**: An undefined marking set is used to describe events in a system. Since events can occur any time during the operation of the system, the undefined marking set is used to state how can these events occur. This can be compared to interrupts in embedded software programming.

**propGroup Marking**

A propGroup marking is made up of a 'propGroup' part and a 'stepVal' part. A marking looks like: `<propGroup> + <stepVal>`. The propGroup part is either a 'Numeral' or begins with an 'Id' as shown in the syntax. The propGroup part is used to categorize a data row as belonging to a group of data rows with the same propGroup value. This is used to express grouping between the data rows.

The stepVal part denotes how many data steps is this data row away from the data row with the same propGroup part but zero stepVal. For example, in the marking `b + 3` 'b' is the propGroup and '3' is the stepVal. The data row that is marked with this marking is 3 data steps away from the row that is marked with `b + 0`. A data step simply indicates a change in state of the system, it must not be compared to a time step or a concrete notion of timestamps or absolute time values. Another consequence of the above is that, a marking like `b + 3` cannot exist without a marking `b + 0` preceding it. This is because the marking `b + 0` defines the reference point for any numerically greater stepVal marking.

**Rule 1**  
A marking with a non-zero stepVal is always preceded by a marking of the same propGroup but with a numerically lower stepVal.

The markings that do not have a propGroup i.e. purely *Numeral* markings cannot have more than one occurrence of the same value in a data sequence. This is because the meaning of a *Numeral* marking is that it represents a definite data step in the data sequence in reference to the start of the data sequence. A *Numeral* marking, therefore, tells how many definite data steps occurred from the start of the data sequence. Since this is an absolute measurement, the same definite data step cannot occur twice in a data sequence. An error is emitted by the software implementation (as a part of this work) when such a duplication is encountered. One use case of this marking is when the engineer wants to define absolute deadlines that the system must obey.

**Rule 2**  
The same *Numeral* marking cannot occur twice in a scenario data sequence.

*stepVal* is used to define an ordering between the data rows and within the propGroup. A stepVal can either be a *Numeral* or can be a *Numeral + Ques*. A *Numeral* stepVal defines that this data row occurs exactly after *Numeral* number of data steps in the group. A *Numeral + Ques* stepVal defines that this data row can occur at any time after *Numeral* number of steps in the group. The *Ques* symbol essentially denotes uncertainty. So, with a *Numeral + Ques* stepVal, it is only guaranteed that this condition occurs after *Numeral* steps but when exactly is not guaranteed. This is typically used to denote a behavior in the system which is guaranteed to happen after some time but
the time taken is not known. These semantics are checked by software and errors are reported if violations are found.

**Rule 3** A Numeral + Ques stepVal denotes an event that belongs to a propGroup. This provides two guarantees:

1. This state of the system can only occur on or after the Numeral value not before it.
2. This state will definitely occur before the change of the propGroup but exactly when after the Numeral value is not guaranteed.

**Undefined Marking**

An undefined marking is used to specify that a data row, which represents certain conditions of data headings, can occur at any point in the data sequence. It can be thought of as an interrupt condition that can occur at any point during the data sequence. The undefined markings can be named by following the ? symbol by an Id. As an example ? and ?cleanup are two different undefined markings that can both happen at any time. All undefined markings are stated before the beginning of a propGroup marking set. This highlights that these are interrupts and can occur at any time during the data sequence.

**Rule 4** An undefined marking can occur at any point in the data sequence.

### 5.5 Algorithm for FOL formula creation

The FOL formula creation algorithm starts with the set of true propositional logic formulae. For each predicate of a relationship model, a tree is constructed. The tree represents all the arguments to the predicate that resulted in an evaluation of true for that assignment. A tree is shown in figure 5.8. The root node is null which does not represent any argument. All child nodes of the null node represent the first argument to the predicate. All child nodes of the node corresponding to the first argument represent the second argument and so on. Each argument is an instantiated token as discussed in section 5.3. It can be observed that the depth of the tree will be the number of arguments or the arity of the predicate.

The depth of an instantiated token from the null node describes its position in the argument order to a predicate. The quantifier assignment will be done in the order from the first argument to the last argument but since it will be done for each argument position in the same way, this order does not matter.

To create first order logic formulae the semantics of quantifiers ∀ and ∃ must be understood, this was described in 2.3.2. In context of the arguments, these quantifiers can be applied to the marking part of an instantiated token. Since the marking has a propGroup part
and a stepVal part, the quantifiers can be applied to both parts. This is done by first generalizing both parts of the marking in three different ways to create sets and then the quantifiers are applied over that set.

This generalization step to construct sets is discussed next.

### 5.5.1 Marking Generalization

The marking generalization constructs the sets for which a quantifier can be assigned. A marking is generalized using the following steps:

1. The algorithm traverses the tree for every depth and generalizes only those sub-trees that are at the same depth. This is equivalent to saying that the first argument cannot be generalized with the second argument. This is done to reduce algorithmic complexity and this does not make the FOL formula any less effective. If a generalization between two arguments is needed then this can be done later by merging the common sets created through this algorithm.

2. If more than one sub-trees at a depth exist that have the same data heading in the instantiated token, then the marking can be generalized to create a set with
more than one element. Else the generalized set will only contain one element and is equivalent to a propositional logic formula. An example in figure 5.8 would be the `healthQopen.Start + 0` and the `healthQopen.reset + 0` sub trees. These would result in a set with `Start` and `reset` as the set members.

3. The argument sub trees (for this depth) are then generalized using the following rules.

   a) **StepVal generalization**: stepVal generalization means that a set of valid stepVals will be created. This means for multiple instantiated tokens that satisfy all the steps above and only differ in their stepVals (implying that propGroups are same) can be generalized. A set of all the different stepVals is constructed while the propGroup is the same.

   b) **propGroup generalization**: propGroup generalization means that a set of valid propGroups will be created. This means for multiple instantiated tokens that satisfy all the steps above and only differ in their propGroups (implying that stepVals are same) can be generalized. A set of all the different propGroups is constructed while the stepVal is the same.

   c) **propGroup + stepVal generalization**: This means that both propGroup and stepVal sets will be created. The instantiated tokens that satisfy all the steps above but differ in their propGroup and stepVals are generalized. Sets of all the different propGroups and stepVals is constructed.

Once the generalizations are done the output would look something like as shown in figure 5.9. The set names are in capital letters and they replace propGroup or stepVal at their respective positions. As can be observed, the generalized markings are created at every depth level of the tree shown in figure 5.8 to get generalized marking for each argument of the predicate.

<table>
<thead>
<tr>
<th>Sets:</th>
<th>A-[reset, Start] B-[0] C-[reset] D-[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicate:</td>
<td>equal3(healthQopen.A+B,canProcess.C+D,0)</td>
</tr>
</tbody>
</table>

Figure 5.9: Informal representation of propGroup + stepVal generalization output

### 5.5.2 Quantifier Assignment

Once the sets are created, a generalized marking is created. This marking looks like $A + B$ as shown in figure 5.9 where $A$ and $B$ both refer to sets. In a generalized marking as shown in figure 5.9, two sets have been generalized for one data heading. One quantifier is generated for one generalized marking, it does not matter if that generalized marking was propGroup, stepVal or propGroup+StepVal generalized. In the case of propGroup+stepVal generalization where 2 sets will be generated, only one quantifier is generated that applies to both sets. This makes sense because in other generalizations
the individual quantifiers for each such set was already generated correctly. So in the
propGroup+stepVal generalization only those quantifiers add value which apply to both.

Multiple instantiated tokens exist for one generalized marking. For each instantiated
token there exists a tree rooted at the instantiated token. The trees for each instantiated
token corresponding to one generalized marking are compared to find the applicable
quantifier. The decision of a $\forall$ vs $\exists$ quantification is taken based on the following rules:

- If all sub-trees rooted at the instantiated tokens (with the same data heading) for
  a generalized marking have the same sub-trees (not just children) only then a $\forall$
  quantification can be given to the generalized marking.

- If there exists even one sub-tree, rooted at the instantiated tokens for a generalized
  marking, that does not have exactly the same sub-tree as all others then an $\exists$
  quantification is given to this generalized marking.

The reason for the above rules can be clarified with the following. Consider a case, where
a for-all quantification is given to a generalized marking where every sub-tree has all the
same children except one tree which even though it has the same children but it has one
extra child. In this case if a for-all quantification is given and the next argument for the
predicate(next sub-tree in the tree) is chosen with this extra child then the formula that
is generated will be false. Therefore not only should all trees have all the same children
but they should not even have more to get a for-all quantification.

Once the quantifiers are assigned a First order logic formula is created, this is shown in
figure 5.10

\[
\forall A \in \{\text{reset}, \text{Start}\}, \forall B \in [0], \forall C \in \{\text{reset}\}, \forall D \in [0]. \\
equal3(\text{healthQopen}.A+B,\text{canProcess}.C+D,0)
\]

Figure 5.10: Example: Generated first order logic formula

It must be noted that during the complete algorithm the information of the propositional
logic formulae is transformed into sets and quantifiers are applied to them. Since no new
information is created the algorithm of FOL creation generates only finitely many FOL
formulae. In contrast, if a system existed that would generate new information based
on the provided information, it can be intuitively understood that, the new information
produced could be added to the provided information and this new set could be used
to produce new information and so on. The process of producing new information and
adding it to existing information could create a cycle that would continue infinitely. This
is avoided in the presented approach.
5.6 Algorithm for Kripke Structure formation

The algorithm for Kripke structure formation is not as complex as for the first order logic formulae generation. The Kripke structure semantics themselves are described in section 2.4. The key aspects of a Kripke structure are the set of all possible states, set of initial states, transitions and the labeling function. The generation of Kripke structures is primarily based on the Marking syntax and semantics.

In this section the generation of the Kripke structures will be discussed and the exact evaluation of the feasibility of the marking syntax and semantics for its generation will be discussed in Chapter 7.

Every scenario data file is allowed to have more than one situation that can be described for the same scenario. This is equivalent to saying that, not all requirements can be described in one situation (data sequence) i.e. a system can have multiple starting configurations for one requirement and can also have different responses and actions based on it. Therefore the proposed approach allows the requirements engineer to describe more than one situation (data sequence) in one scenario data file which describes or corresponds to the same set of requirements.

Each data sequence is therefore separated by ‘___’ (triple underscores) in the Marking column. This indicates that one data sequence ended and another may begin thereafter. Each data sequence can depict a different situation but it must have the same data headings and must correspond to the same requirement. This was already shown and discussed in Figure 5.4 in section 5.2. As a result of this, every data sequence can be analyzed separately and can be used to create its own Kripke structure. Therefore multiple Kripke structures can be used to describe a set of requirements. It is also envisioned that the multiple Kripke structures can be merged to form one Kripke structure that represents the complete set of requirements. This is considered a possible extension and is therefore not within the scope of this work.

With the understanding of the marking syntax and semantics described in section 5.4 and the Kripke structure semantics in section 2.4, the creation of Kripke structures from the marking semantics is discussed in the following section.

5.6.1 Kripke Structure from Marking Semantics

A Kripke structure is created by reading the ‘Marking’ column in order. The markings that are used to generate one Kripke structure only correspond to one data sequence. The following rules are used to produce a Kripke structure corresponding to one data sequence:

1. For every unique marking in a data sequence a state in the Kripke structure is created.

A state is uniquely identified by the marking and data rows with the same markings
correspond to the same state. By using the same marking more than once, the requirements engineer indicates to the system that the same state is expected at that point of the system.

2. The initial state of a Kripke structure is the first state corresponding to the marking in a data sequence which is not an undefined marking (ref section 5.4.1).

Since the undefined markings can occur at any time in a data sequence, the initial state of a system is described by the first marking that is not an undefined marking.

3. While reading the marking column in order of appearance, every change in the marking represents a valid transition.

Since every row represents a state of the system, a change from the first row to the next represents a transition of states. This is directly translated to the Kripke structure transitions.

4. Every state carries the set of properties that are true in that state. This is the labeling function. The propositional logic formulae that were identified to be true on the scenario data are used to attribute each state to the formulae that are valid in that state.

From all the true propositional logic formulae, the formulae with only one marking for all arguments are filtered. Therefore a set of properties with the same markings for every argument in the predicate is obtained. This set of properties are directly attributed to its corresponding state in the Kripke structure.

5. Undefined markings are represented as states with their corresponding properties inside those states but no transitions are visualized even thought they exist.

Since undefined markings can occur at any time during the data sequence all rows in a data sequence can transition to and from an undefined marking. This is not explicitly visualized in figure 5.11 because it would make the Kripke structure look chaotic, instead no transitions to and from the undefined marking states is shown because it must always be assumed that these transitions always exist just like interrupts.

6. The propGroup markings with undefined step value of the form \(Id + Numeral + Ques\) is especially interesting. Since this data row can occur any time step after \(Id + Numeral\) therefore transitions are added to and from this marking state to all marking states with the same propGroup that have a \(Numeral\) value greater than that of this marking.

An example demonstrating the creation process is shown in figure 5.11. As can be seen this Kripke structure is for the data sequence between lines 5-21 of the markings. Also this Kripke structure does not show the true properties but this is done to make the image readable.

In figure 5.11, the undefined markings are ?a, ?b and ?c. These are shown in the top
of the figure. All the unique markings have a state with that name in the figure. The transitions can be verified in the order of the data rows. The marking $b+1?$ corresponds to rule 6, and contributes to the transitions of $(b + 1), (b + 2), (b + 3) \rightarrow (b + 1?)$.

Figure 5.11: Illustrating Marking semantics with generated Kripke structure

5.7 Outputs

The outputs produced by the proposed system are Kripke structures and FOL formulae. In this section we only discuss the quality of the produced outputs and any limitations thereof. The outputs produced for the running example using the presented approach are shown and discussed in chapter 7 section 7.4.

5.7.1 First Order Logic (FOL) Formulae

The FOL formulae are produced using 3 types of generalizations. It was observed that the formulae with both the propGroup and stepVal generalization are the most concise. It was also observed that the repetitions in this type of generalizations were the least. This means that redundancy of information was lower when both parts of a marking were generalized when compared to individual parts. Even though this gives the impression that generalizing the propGroup and stepVal separately will provide little value, it must be noted that this is not the case.

It must be understood that the objective of the proposed system is to find the FOL formulae that provide the most value to the engineer. It must be considered that
there might be cases where simply because of forcing to generalize both the propGroup and stepVal together, some interesting properties, that can only be generalized till the propGroup, are rejected.

Therefore using all 3 types of generalizations is currently considered a good way to provide relevant information to the engineer.

It was observed that the for-all quantification provides the most concise information in a FOL formula because an exists quantification only says that for some value in the set, the formula evaluates to true whereas in case of a for-all the statement is true for all values in the set. While this difference may imply that a system that creates FOL formulae should be greedy to produce more $\forall$ quantified formulae, but this was not done in the proposed system even if it was easily possible to do it.

The objective of the proposed system is to provide correct information and provide an algorithm that preserves the ground truth that is represented by the scenario data. Therefore the presented algorithm assigns an $\exists$ quantification even if only one element in the set does not satisfy the formula. In contrast, an option could be to simply remove the non-satisfying element from the set and then create a $\forall$ quantification for the rest of the set. This was not chosen because it fragments the generalized set. Since the choice of one way over the other depends on the perspective and needs of the engineer, this choice is therefore stated here for the sake of completeness.

**Limitations**

During the formulation of the algorithm for the creation of FOL formulae, various decisions were taken. Some of the limitations of these decisions are discussed here.

The FOL formulae are produced based on the marking generalizations. This means that the sets constructed for quantification will be sets of markings. This fact can be seen as a limitation of the algorithm presented. It can be imagined that FOL formulae that quantify over the variable itself rather than the markings would provide more value to the engineer. This is not achieved with this work.

It must be noted that markings provide an essential way to make statements about the data rows in a formal way. A natural extension to this work could be to create FOL formulae that quantify over the variables. This is still possible to implement while retaining markings and will be discussed as a part of the future work.

**5.7.2 Kripke Structures**

The Kripke structures created by the proposed system are based on the marking syntax and semantics. Kripke structures model the desired system and can be translated into Büchi automata for model checking (ref section 2.6.2).
It was observed that the Kripke structures created were very strict to the definition of a state. Every marking has its corresponding state. Since a marking can occur multiple times in a scenario data file, only those properties are attributed to a state that are valid at all possible values corresponding to the same marking. This often leads to states being very strict with what properties can be attributed to them. This strictness often highlighted the expectations of the engineer which were not fulfilled by the scenario data.

It must be noted that this strictness is needed to preserve correctness of all generated propositional logic formulae under all possible interpretations of that formula. Therefore this strictness was seen as a natural consequence of being correct under all interpretations.

**Limitations**

The algorithm of Kripke structure creation supports expressing “interrupts” in a system. This was done using the undefined marking in the marking syntax and semantics. The undefined marking expresses the interrupt state but there is no direct way to represent the same in a Kripke structure.

This is because a Kripke structure does not have any semantics on which transition can take place i.e. there are no specially marked states except the initial states. Therefore there is no default way to restrict a transition from the interrupt state to a state other than the one at which the interrupt occurred. This is seen as a lack of expressiveness of the Kripke structure. Even though the exact intention of the engineer can be captured within the proposed marking syntax and semantics, an exact translation of this is not available within the Kripke structure semantics.

It must be noted that this limitation does not negatively affect a safety analysis based on the generated Kripke structure. If a system can be shown to be safe with respect to the generated Kripke structure, then it will definitely be safe where the extra transitions are not allowed. At the same time, it can also be understood that this limitation would lead to situations where the safety of a system would be unnecessarily rejected. Therefore this is one limitation on which future work can be carried out.
6 Implementation Details

The implementation of the proposed system was done in Java programming language. Java was chosen because it is planned that sometime in the future the proposed system might be integrated within the Eclipse development platform as a plugin.

The first objective of the implementation was to parse the inputs from 3 different files corresponding to the 3 different inputs discussed earlier. Then the inputs are stored as objects on which the algorithms for FOL and Kripke structure creation are applied. The architecture of the implementation is described in the following sections. The output visualization is essential to produce a tangible output. Therefore the Kripke structure objects that were generated in software were graphically visualized. This is also discussed in the following sections.

6.1 Parsing the Inputs

The inputs to the system as described in section 5.2 are in three files. The first file contains the background knowledge as shown in figure 5.3. The second file contains the marked scenario data as shown in figure 5.4. The third file contains the relationship model which is shown in figure 5.6.

The parsing of the inputs is discussed in the following sections.

6.1.1 Background Knowledge Parsing

The GOLD parser builder [28] was used to create a Backus-Naur Form grammar for the background knowledge files. This grammar can be used with a parsing engine to parse the input files. This grammar is shown in figure 6.1

As can be seen from the grammar file, line 10 shows the syntax of a complete predicate definition. A predicate is made up of a name (Id), argument List (<argumentList>), number of arguments (NumberLiteral) and a boolean expression (<BoolExp>). The boolean expression is the definition of the predicate that must use the arguments in the argument list.

The definition of a boolean expression in lines 12-15 includes all fundamental boolean operators. Other relational operators are defined in lines 17-22. Some of the mathematical operators are covered in the definition of <MathExp> in lines 24-26. It can be seen
6 Implementation Details

that currently the mathematical operators divide (\(\div\)) and multiply (\(\times\)) are not specified but this is only an extension and this does not affect the validity of this work.

The grammar is used to generate a parse tree for the background knowledge file. This parse tree is used to create executable predicate objects within Java for each predicate definition in the background knowledge file. The executable predicate objects can be assigned arguments and can be evaluated to true or false for the assigned arguments. This is needed to evaluate the predicates for the scenario data. Once the GOLD parser builder is provided the grammar in figure 6.1, it generates a grammar table which can be used by any GOLD conformant parsing engine. This work used the GOLD engine for Java [45]. The background knowledge file is supplied to the parsing engine. The parsing engine provides the parse tree for the file which is then used by the software implementation to create executable predicate objects. Any operation can be supported by the background knowledge file as long as a proper grammar and the appropriate mapping to an executable predicate object in Java can be created. This means that any operation that can be implemented in Java can ultimately be supported by the background knowledge file by providing an appropriate grammar.

6.1.2 Relationship Model Parsing

The relationship model was parsed in the same way as the background knowledge file. Since the relationship model itself does not require an evaluation like the predicates in a background knowledge, the relationship model file is parsed by the parsing engine and the parse tree is provided to the software implementation. The parse tree is used to extract the variables and the predicate names and then used to construct candidate properties. The Backus-Naur Form grammar for a relationship model file is shown in figure 6.2

As can be seen the grammar is small and simple. In line 19 it can be see that a relationship model is made up of tokens and predicateName. The tokens are defined in lines 11-14
6 Implementation Details

![Figure 6.2: Grammar for relationship model file parsing](image)

and can be of type \textit{Id} and of type \textit{Numeral}. The predicate names are defined in lines 16-17. These can only be of type \textit{Id}. It can be observed from line 21-22 that one file can be made up of one or more than one models separated by a ';'.

### 6.1.3 Scenario Data File Parsing

The scenario data file is structured as a CSV file. A CSV is like a table and can therefore also be viewed and edited in Microsoft Excel or other CSV viewers. An example of a CSV file is shown in figure 6.3. Each new line refers to a data row as shown by the numbering on the left in this figure. Another view of the same CSV file as a table was shown in figure 5.4 in chapter 5.

A scenario data file must contain the headings row and the ‘Marking’ column. This is verified by the software implementation and this was discussed in section 5.2. To parse the CSV file the OpenCSV [71] java library was used. The OpenCSV library allows binding each row in a CSV to Plain Old Java Objects (POJOs) of a class. The headings of the CSV can be mapped to the attributes of the class. The class must be defined by the developer according to which the data is mapped. The class name and the data file is provided to the OpenCSV library and a list of POJOs corresponding to each row in the data file are obtained. This list will then be used to access the data for evaluating the predicates.

### 6.2 Software Implementation

The complete software implementation comprises of 51 files structured in 10 packages. The project contains 3101 executable lines of code which excludes comments and blank lines. These numbers only correspond to the Java software implementation done as a part of this work. This does not include the OpenCSV library and GOLD engine.
implementations. This also does not include the grammar file definitions that were also done as a part of this work but are not counted as java implementation.

6.2.1 Architecture

A class diagram for the software is shown in figure 6.4. Two sections of the class diagram are marked, Parse Results and Outputs. The ‘Parse Results’ refer to source code that was needed to parse the input files and store the results of the parsing. The ‘Outputs’ refers to the classes whose objects are created as a final result of the software implementation. All the remaining area that is unmarked, corresponds to intermediate classes that are needed to organize the information and are created to allow parallelism of the evaluation of predicates.

The class ‘ExecutableProp’ implements the ‘Runnable’ interface. It is used to parallelize the generation and evaluation of all combinations of arguments for a predicate. For every predicate name in a relationship model a separate ‘ExecutableProp’ object is created. This object can run in parallel to all other objects because it is created in such a way so that it does not share any resources with the other threads. Even though significant improvements in CPU utilization were observed with this parallelism, it can be improved further. The improvements can come by balancing the load on each thread. Implementing this is not within the scope of this work but it must be considered for future improvements.
Figure 6.4: Class diagram for the complete software
In figure 6.5, an activity diagram describing the major activities of every executing class is shown. This shows the step wise processes that are carried out in the current system. The ‘App’ class is the entry point of the software and is responsible to interface between different classes. The ‘App’ passes the file names to individual parser classes and runs them. It waits for all parsers to conclude, fetches the parse results and checks them for consistency (ref section 5.2.3). The consistency checker is designed in a different class but it runs in the ‘main’(App) thread itself. After the consistency check is passed, objects of the class ‘ExecutableProp’ are created based on the relationship model information. These objects are referred to as ‘executable properties’ for the sake of simplicity. The main thread issues execute commands to all executable properties and waits for all of them to conclude. After all executable properties have finished, the FOL properties are already displayed and the propositional logic formulae are stored within each executable property class. The propositional logic formulae are fetched from the executable properties and Kripke structures are created from them and the data table. The creation of the Kripke structure happens in the main thread. Every Kripke structure is then visualized using GraphViz for java [62] by generating Portable Network Graphics (PNG) images for them. These are stored on the hard drive for viewing.

6.3 Visualizing the Outputs

The outputs of first order logic and Kripke structure need to be visualized so that they can be easily read by the user. To do this the dot tool was used and GraphViz for java library [62] was used. Using this library the Kripke nodes could be visualized just as they are stored inside the java objects. This is shown in figure 7.9 in chapter 7. The first order logic formulae are rendered as text and shown on the console as illustrated with figure 5.10 in chapter 5 and in figure 7.8 in chapter 7.
Figure 6.5: Activity diagram for the system
7 Evaluation

To evaluate the effectiveness of the proposed system a performance evaluation of the software is done. This demonstrates how the current software scales with respect to the change in inputs. An example was taken for evaluating the performance and various parameters of that example were changed to analyze the effect of those changes on the outputs and the performance.

To demonstrate the realistic usage of the system, a real world example is taken from the VerITAS project[2] (ref chapter 4). A step wise process to specify the example data with the proposed approach is described. This workflow is then used to generate the outputs for the running example.

The feasibility of generating formal specifications as FOL formulae and Kripke structures through the proposed approach is also discussed. Two variations of the real world example are compared to evaluate the effectiveness of the proposed approach. One variation uses a relationship model and the other does not.

7.1 Performance Evaluation

To evaluate the performance of the software, the effects of the specification problem in itself must be removed. This is needed because real systems have patterns in the occurrence of data and a performance evaluation based on these examples will lead to a very specific performance analysis to the problem being specified. Therefore to conduct all tests in this section, the scenario data table was filled with random values based on the \textsc{RANDBETWEEN(0,50)} function in Excel 2013. The table in itself consists of 10 data variables and 30 data rows with syntactically correct markings. The background knowledge files used were the same as shown in Figure 5.3. The relationship models, which are an input to the software, are used to change the different test parameters. These are changed in the relationship model file.

All tests were conducted on a Windows 10 Home Edition, running on a Core-i5 4200M mobile processor with 6 Gigabytes of RAM\(^1\). The processor has a base clock of 2.5GHz and a boost clock of 3.0 GHz. All tests were conducted with only the Eclipse IDE running and with no other applications running. The time measurements were done

\(^1\)Random Access Memory
7 Evaluation

using the Java Mission Control\(^1\). This was done so that no software changes are made to
the software for measurement purposes.

The software itself is supposed to output all the FOL formulae generated. It also counts
and shows the total number of predicate evaluations(ref section 5.2.1) that were done
to create propositional logic formulae. This number is referred to as the “Predicate
Evaluations” in all the presented data.

The time taken by the software from the start of the application thread to the end of
the application thread is measured and this is referred to as the “Time Taken” in all the
presented data.

All observations were collected by averaging 5 runs with the same configuration. This
was intended to remove any noise in execution time due to the unpredictability of the
operating system.

To evaluate the performance based on inputs to the software, various test parameters are
considered, these are :

- Number of variables in one relationship model
- Number of predicates of arity\(^2\) 2 in one relationship model
- Number of predicates of arity 3 in one relationship model
- Number of relationship models in one specification task
- Number of data rows with one relationship model

The analysis of the software based on each of these input parameters is discussed in
the following sections. Since the objective is to understand the influence of the input
parameters on the performance of the software, only one parameter is changed at a time.
The observations from these can be used to understand which input parameters might
become a bottleneck in the future.

7.1.1 Effect of Number of Variables

To remove the effect of any other parameters on the analysis, the following parameters
are held constant with the following values:

- Number of Predicates = 1
- Arity of Predicate = 2
- Number of Relationship Models = 1
- Number of Data Rows = 30

\(^1\)https://www.oracle.com/technetwork/java/javaseproducts/mission-control/index.html(Accessed on
29.08.2019)

\(^2\)arity is the number of arguments that a predicate can take
7 Evaluation

To conduct the tests, the number of variables in one relationship model were increased one by one (from \( \text{var1} \) to \( \text{var10} \)). The predicate was chosen to be the “equal” predicate. Figure 7.1a shows the observed data and Figure 7.1b shows the plotted data.

<table>
<thead>
<tr>
<th>No. of Variables</th>
<th>Time Taken (ms)</th>
<th>No. of Predicate Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>649</td>
<td>423</td>
</tr>
<tr>
<td>3</td>
<td>639</td>
<td>1307</td>
</tr>
<tr>
<td>4</td>
<td>689</td>
<td>2891</td>
</tr>
<tr>
<td>5</td>
<td>865</td>
<td>4866</td>
</tr>
<tr>
<td>6</td>
<td>955</td>
<td>7198</td>
</tr>
<tr>
<td>7</td>
<td>977</td>
<td>10031</td>
</tr>
<tr>
<td>8</td>
<td>1033</td>
<td>13305</td>
</tr>
<tr>
<td>9</td>
<td>1008</td>
<td>16910</td>
</tr>
<tr>
<td>10</td>
<td>1063</td>
<td>21181</td>
</tr>
</tbody>
</table>

(a) Observed Data  
(b) Plotted Data

Figure 7.1: Effect of number of variables

As can be seen from the plot, the number of predicate evaluations rise sharply with the number of variables. This is because all possible combinations of assignments must be considered. The time taken by the software, on the other hand, does not rise so sharply. This is because the total number of predicate evaluations is still quite low (20K). It must be noted that since all execution times are around 1000 ms (or 1 second), they are susceptible to the scheduling delays by the operating system. To get a reliable trend analysis the number of variables must be increased to a value such that the software takes at least more than a second. This would remove the effect of the initial setup time of the software which is of the order of a few 100 ms.

Nevertheless, it is expected from the number of predicate evaluations that the time taken by the software would increase with the number of variables.

7.1.2 Effect of Number of Predicates

To remove the effect of any other parameters on the analysis, the following parameters are held constant with the following values:

- Number of Variables = 10
- Arity of Predicate = 2 (and 3)
- Number of Relationship Models = 1
- Number of Data Rows = 30
To conduct the tests, the number of predicates in one relationship model were increased one by one. The same predicate was added every time. This was done because changing the predicate would result in a different type of computation which would depend on the data. To avoid any dependence on the data, the same predicate was used. Due to this, a pattern in the number of predicate evaluations can be observed. The difference between each consecutive number of predicate evaluations is constant.

This test was performed for two different arities of 2 and 3. The predicates of choice were “equal” and “plus” respectively. This was done simply to analyze if the change of arity changes the trend of the data. Figure 7.2a shows the observed data and Figure 7.2b shows the plotted data for predicate of arity 2. Figure 7.3a shows the observed data and Figure 7.3b shows the plotted data for predicate of arity 3.

For both the predicate arities, the number of predicate evaluations increase linearly. The same can be observed for the time taken. It can be seen that for arity 3 the time taken is significantly greater than for arity 2. This is discussed later.

It is expected from these observations that the time taken would linearly increase with respect to the number of predicates of the same type. This is because each predicate has a fixed number of possible predicate evaluations for a specific relationship model. Addition of the same type of predicate in a relationship model will only increase the number of predicate evaluations needed by the same fixed amount.
### Evaluation

<table>
<thead>
<tr>
<th>No. of Predicates</th>
<th>Time Taken (ms)</th>
<th>No. of Predicate Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1036</td>
<td>21181</td>
</tr>
<tr>
<td>2</td>
<td>1183</td>
<td>42362</td>
</tr>
<tr>
<td>3</td>
<td>1683</td>
<td>63543</td>
</tr>
<tr>
<td>4</td>
<td>1691</td>
<td>84724</td>
</tr>
<tr>
<td>5</td>
<td>2216</td>
<td>105905</td>
</tr>
<tr>
<td>6</td>
<td>2380</td>
<td>127086</td>
</tr>
<tr>
<td>7</td>
<td>2564</td>
<td>148267</td>
</tr>
<tr>
<td>8</td>
<td>2640</td>
<td>169448</td>
</tr>
<tr>
<td>9</td>
<td>3027</td>
<td>190629</td>
</tr>
<tr>
<td>10</td>
<td>3277</td>
<td>211810</td>
</tr>
</tbody>
</table>

(a) Observed Data

(b) Plotted Data

Figure 7.2: Effect of number of predicates of arity 2

<table>
<thead>
<tr>
<th>No. of Predicates</th>
<th>Time Taken (ms)</th>
<th>No. of Predicate Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23405</td>
<td>726598</td>
</tr>
<tr>
<td>2</td>
<td>26470</td>
<td>1453196</td>
</tr>
<tr>
<td>3</td>
<td>41440</td>
<td>2179794</td>
</tr>
<tr>
<td>4</td>
<td>55603</td>
<td>2906392</td>
</tr>
<tr>
<td>5</td>
<td>70000</td>
<td>3632990</td>
</tr>
<tr>
<td>6</td>
<td>76000</td>
<td>4359588</td>
</tr>
<tr>
<td>7</td>
<td>86000</td>
<td>5086186</td>
</tr>
<tr>
<td>8</td>
<td>147000</td>
<td>5812784</td>
</tr>
<tr>
<td>9</td>
<td>180224</td>
<td>6539382</td>
</tr>
<tr>
<td>10</td>
<td>201000</td>
<td>7265980</td>
</tr>
</tbody>
</table>

(a) Observed Data

(b) Plotted Data

Figure 7.3: Effect of number of predicates of arity 3

#### 7.1.3 Effect of Arity of Predicates

To remove the effect of any other parameters on the analysis, the following parameters are held constant with the following values:

- Number of Variables = 10 (and 5)
- Number of Predicates = 1
- Number of Relationship Models = 1
- Number of Data Rows = 30
To conduct these tests, the arity of predicates was changed from 1 to 5. This was accomplished by introducing new 4 and 5 arity predicates that were not included in the background knowledge earlier. These were defined as the following:

\[
\text{arity4pred}(a, b, c, d), 4:: a == b && c == d;
\]
\[
\text{arity5pred}(a, b, c, d, e), 5:: a <= b && c <= d && e < b;
\]

Figure 7.4a shows the observed data and Figure 7.4b shows the plotted data.

<table>
<thead>
<tr>
<th>No. of Variables</th>
<th>Arity of Predicate</th>
<th>Time Taken (ms)</th>
<th>No. of Predicate Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>593</td>
<td>104</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>1041</td>
<td>9179</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>23978</td>
<td>720396</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>101000</td>
<td>1200352</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>OOM</td>
<td>OOM</td>
</tr>
</tbody>
</table>

It can be observed from the plots that the increase in arity exponentially increases the number of predicate evaluations and the time taken. As a result of this, when the number of variables are 10 and the arity of predicate is 4 then the software heap runs out of memory. This is because there are simply too many combinations to create to be fully correct. The same situation arises with arity 5 predicate with 5 variables. Therefore the tests were not conducted beyond these values of arity.

It may be possible that certain algorithmic and memory management optimizations may reduce the number of combinations that are to be tried. This may lead to an improvement in the performance.

### 7.1.4 Effect of Number of Relationship Models

To remove the effect of any other parameters on the analysis, the following parameters are held constant with the following values:

- Number of Variables = 5
- Number of Predicates = 1
- Arity of Predicate = 2
7 Evaluation

- Number of Data Rows = 30

To conduct the tests the number of relationship models in the file were changed from 1 to 10. Since only the performance of the software was under analysis, the same relationship model was used for all 10 instances.

Figure 7.5a shows the observed data and Figure 7.5b shows the plotted data.

<table>
<thead>
<tr>
<th>Relationship Models</th>
<th>Total Possible Pred. Evaluations</th>
<th>No. of Pred. Evaluations</th>
<th>Time Taken (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>19220</td>
<td>2460</td>
<td>663</td>
</tr>
<tr>
<td>2</td>
<td>38440</td>
<td>4920</td>
<td>681</td>
</tr>
<tr>
<td>3</td>
<td>57660</td>
<td>7380</td>
<td>766</td>
</tr>
<tr>
<td>4</td>
<td>76880</td>
<td>9840</td>
<td>863</td>
</tr>
<tr>
<td>5</td>
<td>96100</td>
<td>12300</td>
<td>919</td>
</tr>
<tr>
<td>6</td>
<td>115320</td>
<td>14760</td>
<td>1078</td>
</tr>
<tr>
<td>7</td>
<td>134540</td>
<td>17220</td>
<td>932</td>
</tr>
<tr>
<td>8</td>
<td>153760</td>
<td>19680</td>
<td>1039</td>
</tr>
<tr>
<td>9</td>
<td>172980</td>
<td>22140</td>
<td>1129</td>
</tr>
<tr>
<td>10</td>
<td>192200</td>
<td>24600</td>
<td>1260</td>
</tr>
</tbody>
</table>

(a) Observed Data  (b) Plotted Data

Figure 7.5: Effect of number of relationship models

As can be seen, in this table another measurement of “Total possible predicate evaluations” was included. This measurement tells the total number of possible assignments that were possible for the complete run of the software. It shall be observed that of the 19,220 possible predicate evaluations only 2,460 are actually evaluated. This is one key optimization that was used to reduce the number of evaluations without affecting the correctness of the generated formulae.

To understand why this can be done, we must first understand that each marking (e.g. \( \text{normal} + 0 \)) can occur more than one time in a scenario data table. This means that even if the the predicate has an argument of \( \text{var1}.\text{normal} + 0 \), this assignment can take more than one value. The propositional logic formula created with this argument must be true for all values of this argument in the table. If the assignment of even one value from the data table makes the predicate evaluation false then it can be safely concluded that the predicate is at least false for one value in this data. This means no further assignments need to be checked. This reduces the total number of evaluations. This simple technique shows a reduction of the number of computations of about 87.2 percent \((= ((19220-2460)/19220) * 100)\) in this artificial test. The actual saving depends on the data and the patterns in the data and therefore this percentage must not be considered.
a guaranteed saving but, a saving is guaranteed.

It can be observed in the plot that the number of predicate evaluations increase linearly with respect to the number of relationship models. Since for all these observations the time taken was around one second, a trend analysis does not give any useful information because of an effect of scheduling delays from the operating system.

It is can be expected to find other optimization techniques without sacrificing correctness of generated properties.

7.1.5 Effect of Number of Data Rows

To remove the effect of any other parameters on the analysis, the following parameters are held constant with the following values:

- Number of Variables = 5
- Number of Predicates = 1
- Arity of the Predicate = 2
- Number of Relationship Models = 1

To conduct these tests, 5 data rows were added to the existing 30 for each test. The data rows were given the markings that already existed in the 30 rows so as to not introduce any change in the number of unique markings. Every data row inserted was then populated by the \texttt{RANDBETWEEN(0,50)} function in Excel 2013, as was done with the other rows. Figure 7.6a shows the observed data and Figure 7.6b shows the plotted data.

As can be seen in the “Number of Predicate Evaluations” column, the number remains constant. This is because of the computation reduction technique discussed in sub-section 7.1.4. It can also be seen that even though the total number of possible evaluations grew to 112,500 the total number of actual evaluations remained constant at 2,460. A saving as high as 97 percent. This is primarily because in this artificial test no new logical change was added. The 30 data rows that first existed already showed that the all possible predicate assignments will result in a false propositional logic formula. Therefore, even with the addition of more data rows, the total number of evaluations that were needed, did not increase.

This shows that the software only carries out those evaluations that will result in a true property. This can be a positive or a negative depending on the objective. If the objective is to only find false properties then a minor software change can favor creation of false properties. In this work the objective was to find true properties and therefore the software was designed this way. It must be noted that the core algorithm shall still remain the same even if false properties are to be discovered.
7 Evaluation

<table>
<thead>
<tr>
<th>Data Rows</th>
<th>Total Possible Predicate Evaluations</th>
<th>No. of Predicate Evaluations</th>
<th>Time Taken (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>18000</td>
<td>2460</td>
<td>682</td>
</tr>
<tr>
<td>35</td>
<td>24500</td>
<td>2460</td>
<td>657</td>
</tr>
<tr>
<td>40</td>
<td>32000</td>
<td>2460</td>
<td>685</td>
</tr>
<tr>
<td>45</td>
<td>40500</td>
<td>2460</td>
<td>705</td>
</tr>
<tr>
<td>50</td>
<td>50000</td>
<td>2460</td>
<td>774</td>
</tr>
<tr>
<td>55</td>
<td>60500</td>
<td>2460</td>
<td>863</td>
</tr>
<tr>
<td>60</td>
<td>72000</td>
<td>2460</td>
<td>853</td>
</tr>
<tr>
<td>65</td>
<td>84500</td>
<td>2460</td>
<td>801</td>
</tr>
<tr>
<td>70</td>
<td>98000</td>
<td>2460</td>
<td>856</td>
</tr>
<tr>
<td>75</td>
<td>112500</td>
<td>2460</td>
<td>877</td>
</tr>
</tbody>
</table>

(a) Observed Data

Figure 7.6: Effect of number of data rows

7.2 Feasibility of FOL

The feasibility to express FOL properties with the proposed approach can be evaluated by checking for correctness of the FOL formulae created. If all FOL formulae are correct then it is shown that the proposed approach of creating specifications is capable to produce correct FOL formulae. No automatic way exists to verify if the algorithm produces correct FOL formulae and the only reasonable way would be to compare it with another trusted implementation that generates FOL formulae for the same files. Since this is not available, manual verification and reviews of the algorithm were undertaken. As a result of the verification process the following results were obtained:

- The FOL creation algorithm is verified to generate correct formulae for 5 test scenarios (all 3 input files for each scenario were created). One of the scenario was specifically made to contain pseudo-random data\(^1\). The FOL properties generated by all scenarios were verified to be correct over the data manually.

- The algorithm was reviewed by one reviewer and the algorithm was understood to be very careful in assigning a ‘∀’ quantifier. This means that the algorithm favors creating correct and concise ‘∃’ formulae rather than creating more number of ‘∀’ formulae. This explicit choice was made to preserve the generalized sets instead of fragmenting them (ref section 5.7).

\(^1\)In contrast to random data like in section 7.1, the data was copied and pasted in chunks manually so that some logical patterns still exist in the data.
7 Evaluation

<table>
<thead>
<tr>
<th>Criteria</th>
<th>With Relationship model</th>
<th>Without Relationship model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total FOL formulae produced</td>
<td>96</td>
<td>2728</td>
</tr>
<tr>
<td>Number of predicate evaluations</td>
<td>869</td>
<td>92730</td>
</tr>
<tr>
<td>Total possible predicate evaluations</td>
<td>4754</td>
<td>678160</td>
</tr>
<tr>
<td>Total time taken (ms)</td>
<td>538</td>
<td>1965</td>
</tr>
</tbody>
</table>

Table 7.1: Evaluating 2 different FOL creation algorithms

**Comparison: With/Without a Relationship Model**

To understand the effectiveness of the proposed system, the system which uses a relationship model was compared with a system that does not use a relationship model. A relationship model is used to guide the system to create candidate properties from which FOL formulae are then generated (ref section 5.2.3). The comparison was done on the basis of the total time taken, total possible predicate evaluations, number of predicate evaluations and the total number of FOL formulae produced. The data presented in Table 7.1 is based on the specification effort of the running example.

It can be seen from Table 7.1 that the algorithm with the relationship model produces much less FOL formulae than the algorithm with a relationship model. This difference of about 96 percent reduces the amount of information presented to the engineer. The engineer gets only those formulae that are interesting for the specification problem and that example. Without a relationship model, numerous mathematically correct formulae would be produced that may not be relevant in the context of the target system. With the proposed approach of using relationship models, the engineer always has the option to generate all possible formulae but in most cases the extra properties generated are only mathematically correct but convey no special meaning in context of the system.

It can also be seen that the number of actual predicate evaluations done without a relationship model are about 99 percent more than with a relationship model. The important point to understand is not the percentage difference in itself because this would vary on a case by case basis. The important point is that with all this extra computation the information produced may not be valuable to the engineer because first, it is too much and second, it may not be contextually relevant.

The results obtained by comparison justify that using a Relationship model reduces the workload and restricts the information produced to be strictly useful for the engineer. This reduction of information is not necessary but extremely valuable because it reduces the number of things the engineer has to focus on.

The proposed approach also supports providing all the information just like the algorithm
without a relationship model. This can be achieved by using all predicates and all variables in one relationship model. This means that the proposed approach favors reduced information but supports providing more information if needed. This is essential in focusing the specification effort to the properties that are of interest to the engineer.

This interesting-ness is difficult to express numerically but by experience it is observed that all mathematically correct properties of a system are not valuable for specification purposes. Therefore an approach which captures the interest of the engineer and produces formal specifications accordingly should reduce the computational effort as well as provide the engineer with the information that s/he is interested in.

### 7.3 Feasibility of Kripke Structure

The feasibility of the marking syntax in creating a Kripke structure can be established if for all the semantics of a Kripke structure there exists a way to express those semantics in the marking syntax. We now discuss all aspects of a Kripke structure that must be supported by the proposed system and also discuss ways of how they are supported.

The Figure 7.7 shows all the elements of a Kripke structure that can be directly mapped to the markings in a scenario data table. Each element is explained individually after the diagram.

![Figure 7.7: Kripke structure elements illustrated on scenario data table](image)

It can be noticed that the figure does not map the labeling function to the scenario data. This is because the labeling function is generated by the system and cannot be directly mapped to the scenario data table.

**States**

The states in a Kripke structure denote the state of a system at a certain point in the life of the system. This notion is exactly captured by the scenario data file itself and each row is considered a statement about some intermediate state of the system. A key addition
to this is that the names of these states are given a meaning and a notation (meaningful name) by the marking. The marking defines or names this state, but not only does a marking name the state, it also provides information about this state. If this state is a part of a bigger group of states then the markings of the complete group have the same propGroup (ref section 5.4.1). The marking can also provide information if this state is intended to come after some other state within the group. This information is provided by the stepVal in the marking.

The states are represented in the Figure 7.7 by the golden color.

Since the scenario data rows already establish the statefulness of the system, the marking syntax only adds more information about these states that is not contained in a node of a Kripke structure. This means the proposed system has more than required information about the states to express them as a Kripke structure node.

**Initial States**

A Kripke structure is allowed to have multiple initial states. This is also allowed in the proposed system within one scenario data file. This means that in one scenario data file multiple situations within the same scenario can be depicted and for each situation/ data sequence (ref section 5.4.1) a different initial state can be chosen. The system also allows specially named initial and termination states named ‘Start’ and ‘End’ respectively. It must be noted that the system assumes the first state in a data sequence is the initial state for that data sequence and so to input multiple initial states, multiple data sequences must be provided. The separation of data sequences allows the engineer to describe different situations based on different initial conditions.

The initial states are shown in the Figure 7.7 by blue color.

**Transitions**

The transitions in a Kripke structure are unguarded transitions. This means that going from one state to the other is not dependent on any condition. The only thing that is guaranteed is that once the system reaches a state then it must satisfy all properties that are labeled to it by the labeling function.

The transition semantics are very easily visible in the scenario data file. Every row succeeding the current row is considered as a transition. This means if in one data sequence one state is encountered multiple times but every time this state is followed by a different state, then this state will have as much outgoing transitions as the number of unique states that follow it.

The transitions are shown in figure 7.7 by the red color.
Labeling function

The labeling function for a Kripke structure gives all the properties that are true for a state. The properties in a Kripke structure are in propositional logic and this is exactly what the proposed system also uses. Every propositional logic statement that is true is stored in a collection and every Kripke node contains a map to the properties that are true in that node. This has nothing to do with the marking syntax but rather with the proposed system.

The labeling function cannot be directly mapped to a scenario data file element. A labeling function is a result of attributing true propositional logic formulae to nodes in a Kripke structure. This can be observed in the output of the running example in figure 7.9.

7.4 Example: Queue Timeout

The running example described in Chapter 4 was used to evaluate the proposed approach. The specification workflow refers to how the specification of the requirements would be done with the available information.

7.4.1 Specification Workflow

To specify the scenario the following steps are taken.

- The requirements and available information is analyzed to identify possible variables that can be used to illustrate the requirements. These variables were mentioned in section 4.1.

- The variables are now put into a table with the first column as the “Marking” column and the other columns with the variable names.

- The engineer can now think about the first step, or initial state that the system should be in. This can be illustrated by the Marking name Start or by a marking name of the engineer’s choice. It could also be named Init or normal or anything else like in figure 7.7.

- After deciding the marking name, the state of all the variables is now defined by assigning values to each variable at that marking.

- Now the engineer can think about the next step. The next step could be named as normal, to denote that the system is under a normal state of operation. The engineer now continues to specify the state of each step after the normal step using normal + 1 or normal + 2+? as marking names. The meaning of these are described in section 5.4.1
• To illustrate a scenario where the system resets the message queue. The engineer can name another marking called \textit{reset}. This could denote a state where the message queue is closed.

• After completely specifying the scenario data according to the requirements. The engineer must construct relationship models with variables for which formal specifications are interesting to the engineer. The engineer can use the variables \textit{time} along with 10 and a predicate ‘lessThan’ to say that \textit{time} should be related to 10 by the predicate lessThan. Another example could be, \textit{healthQopen}, \textit{canProcess} and 0 are related to each other by the predicate ‘equal3’. These are shown in 5.6. These relationship models simply express a possible relationship according to the engineer. It must be noted that some relationship models may yield no formal specifications.

This completes the specification process. The resulting scenario data file was already shown in figure 5.4. The resulting relationship models were shown in figure 5.6. The engineer can now simply generate the formal specifications for the scenario data. The outputs produced are FOL formulae and Kripke structures that are valid on the scenario data. These outputs are discussed in the next section.

### 7.4.2 Evaluating System outputs

In this section we evaluate and understand the outputs produced by the proposed system.

**FOL formulae**

There were 96 FOL formulae produced as a result of this specification effort for this example. Since these 96 formulae were a combined result of all 3 types of generalizations (ref section 5.5.1), it was seen that a lot of them were expressing a common logical formula that could be derived from a combination of all the formulae. These redundancies could be removed in the future by simply generating only those formulae that are logically different.

All FOL formulae were verified to be true for the specific artifacts. Some key examples are shown here and their logical meaning is discussed.

The first FOL formula states that the value of the variable \textit{healthQopen} is zero for the \textit{reset, Start} markings, but for the variable \textit{canProcess} the value is only zero for the \textit{reset} marking. This subtle difference could be overlooked by a human author of a FOL formula. It essentially highlights the states of the system when the message queue is closed.

The second FOL formula states that the value of the variable \textit{time} is less than 10 for all step values of the marking \textit{normal}. This essentially encapsulates the requirement that the system shall stay in the normal state (not reset the message queue) when the time since the last received message is less than 10.
7 Evaluation

| 1. $\forall A \in [\text{reset, Start}], \forall B \in [0], \forall C \in [\text{reset}], \forall D \in [0], \text{equal3}(\text{healthQopen}.A+B, \text{canProcess}.C+D,0)$ |
| 2. $\forall A \in [0, 1, 2, 2+?] \text{ lessThan}(\text{time.normal}+A,10)$ |
| 3. $\forall A \in [\text{reset}], \forall B \in [0] \text{ equal}(\text{time}.A+B,10)$ |
| 4. $\forall A \in [0, 1, 2, 2+?] \text{ equal}(\text{canProcess.normal}+A,1)$ |

Figure 7.8: Generated first order logic formulae for the queue timeout example

The third FOL formula states that the value of \textit{time} will be equal to 10 for the \textit{reset} marking. This states the requirement that the system must be in the reset state when no message is received for 10 time units.

The fourth FOL formula states that the value of \textit{canProcess} should be equal to 1 for all step values of the \textit{normal} marking. This states the requirement that the system can process the data when it is in the normal state.

In summary, the FOL formulae generated produce valuable and insightful properties. These can be used to describe properties of the requirements and can therefore be used for formal verification efforts as was discussed in section 2.6.3.

\textbf{Kripke Structure}

The Kripke structure produced by the system states the evolution of the system and what all properties are ensured by each state. Figure 7.9 shows the Kripke structure generated for this example.

It can be observed that every state has a certain set of propositional logic formulae that are true in that state. It is interesting to see that all \textit{normal} marked states share the same propositional logic formulae. These states contain the formulae that show that the time value should be less than 10 in these states. A formula also shows that the system can process the received data in these states.

The \textit{reset} state contains the propositional logic formulae that describe that the health queue should not be open and the system can not process the data received. It also states a very important property that the time value is equal to 10 in this state.

In summary, the Kripke structures generated by the system contain valuable properties that can be used to generate a Büchi automaton that describes the desired properties of
7 Evaluation

Figure 7.9: Generated Kripke structure illustrated with properties
the system. These can then be used with a model checker as was discussed in the section 2.6.2.
8 Conclusion

Formal verification methods require the creation of correct formal specifications. If the formal specifications are created by a human by reading and interpreting the natural language requirements then this process is similar to development of software which also involves interpreting requirements and writing software code. This means that correctness of specifications will have to be verified by ways in which correctness of software code is verified, namely, testing and reviews. This means that using formal methods like this would not solve the problem of asserting correctness of code if the formal specifications themselves must be tested or reviewed which is a time consuming and labor intensive task. Formal verification methods are expected to be significantly more automated.

An intuitive way to solve this problem would be to have a ground truth\(^1\) about the system and use it to generate formal specifications. The advantage here is that the ground truth is created only once and if this ground truth is agreed upon then the formal specifications generated from it must represent the ground truth. This means that a ground truth is used to create another ground truth. Since the source is verified to be correct and is trusted, therefore the created formal specifications automatically carry the same trust.

It can be observed that the creation of the ground truth and agreeing upon its correctness is now the primary challenge. At the same time, this created ground truth must also be in a form with which formal specifications can be generated.

In this work, this ground truth was assumed to be the example data that must accompany the natural language requirements. This is already a common feature of some software development methodologies like BDD (ref section 3.3.2). Since the example data is agreed upon at the same time as the requirements are agreed upon, therefore the example data carries the ground truth. This work proposes ways to express this example data and any accompanying information such that generating formal specifications from them is possible. This work also implements algorithms to generate the formal specifications from these inputs. The evaluation of the effectiveness of the proposed ways to express example data is also presented.

This work proposes the use of a marking column within the example data and specifying relationship models along with the example data. With the proposed syntax and semantics of the marking column, vital information about the example data is captured (ref section 5.2.2). It was observed that the marking column describes the organization of example data.

---

\(^1\) A fundamental truth based on observations. Like laws of physics for mechanical systems, requirements for software can be the ground truth for software systems
data. It is needed to produce formal specification in First Order Logic (FOL) and Kripke structures. The proposed marking syntax and semantics capture even more information than is strictly needed to create Kripke structures. This was observed as a limitation of the formalism of Kripke structures.

In addition, this work proposes specification of a relationship model along with the example data. It was shown that the relationship model significantly reduces the computational load of generating formal specifications. At the same time, the relationship model captures the interest of the engineer and only produces formal specifications that are interesting and relevant for the engineer. This makes the formal specification task much more shorter and relevant for the engineer. These contributions lay the foundation of Automated Property Formation in [87].

An example set of requirements from the VerITAS project were taken. These requirements were illustrated with example data and the proposed approach was applied on this example. The results showed that relevant formal specifications for this example were generated. These specifications formalized key aspects of the requirements.

The implementation of the algorithms was done in the Java programming language to ensure future re-use as an Eclipse™ plugin. The implementation resulted in 51 files structured in 10 packages. The project contains 3101 executable lines of code which excludes comments and blank lines. This excludes the grammar description for the marking syntax and other files needed for parsing. A performance evaluation of the proposed generation algorithms concluded that the relationship model allows the generation of formal specifications to occur within realistic processing times. It also highlights key points of improvement in the algorithm that could become a bottleneck for bigger specification problems.

During the course of this thesis, it was understood that formal verification methods solve a big problem of ensuring consistency and correctness of requirements and source code. However, this does not mean formal verification methods are supposed to solve all the problems related with creating safe software. Methods like testing, simulations and reviews are still needed to reduce costs and ensure safety. This work strengthens the connection between formal methods and testing by using example data to generate formal specifications which can also be used to generate test cases for testing. This work simply aims to facilitate the use of formal verification methods in the industry by providing a way to generate formal specifications instead of having to write them. Formal methods must co-exist with other forms of verification and validation methods.

This work opens up significant possibilities of improvement and research areas. These are discussed in the next chapter.

\footnote{1www.eclipse.org}
9 Future Work

This work accomplishes to generate formal specifications as FOL formulae and Kripke structures. This effort can be built upon further. During this work, possible next steps and various extensions were identified. These will be discussed in this chapter.

9.1 Next Steps

The next steps that can be taken to utilize the outputs created by the proposed system will be discussed in this section.

9.1.1 First Order Logic formulae

FOL formulae indicate the relationships between different arguments of a predicate. Various ways in which the generated FOL formulae can be used are:

- Multiple FOL formulae can be combined based on common sets for which the different formulae are true. This can provide a concise understanding to the engineer of what properties does the system obey at critical points of the system.

- FOL formulae can also provide mathematical relationships between data headings that the engineer did not expect. The relationship model guides the property formation process. The engineer can also create a relationship model that essentially says that the variables in the relationship model must not be related to each other with the predicates of the relationship model. The proposed system can then be used to find properties that exist in the data but they should not (according to the engineer). This can be used to find problems in the example data.

- FOL formulae are easy to translate to other languages since FOL is a formal language. They can be translated to SPARK contracts that can be used with the actual software. The only challenge in integrating with SPARK would be to understand what elements of the FOL formula refers to the actual software. This might be solvable using an interface document that the complete system obeys for data exchange.

- The generated FOL formulae can be used as a set of axioms about the system. These axioms can be provided to a theorem proving engine that can prove other properties about the system and check for inconsistencies.
9.1.2 Kripke Structure

Kripke structures as opposed to FOL formulae aid in understanding state-wise development of the specified system. It allows the engineer to visualize how the true properties of the system change in every step of the system.

Some key ways of using the generated Kripke structures are:

- A Kripke structure can be used to check LTL properties about the system. This means properties like deadlocks and liveness can be checked by the Kripke structure specification.

- Kripke structures can be converted to Büchi automata. These automata are used by tools like SPIN to verify LTL properties. Also two or more Büchi automata can be used to find contradictions between each scenario. This can be helpful if some processes are running simultaneously and it must be verified that they do not interfere with each other or lead to inconsistencies between critical parts of the system.

9.2 Extensions

9.2.1 FOL Formula Generalization

This work accomplished 3 types of marking generalizations which were discussed in section 5.5.1. At that point the objective for generalization was to construct FOL formulae. In this section, the objectives for further generalization can be changed to the following:

- Create lesser amount of unique FOL formulae. This is needed so as to provide concise information to the engineer. Providing thousands of correct formula add little value because this cannot scale for industrial size systems.

- Create shorter formulae. A shorter formula is easier to read. If a shorter formula expresses the same concept as a longer formula then, the shorter formula is preferred.

These objectives can guide the future generalization process. A few ideas to accomplish the above objectives are presented as follows:

- Currently, in the marking generalization, sets are created according to the generalized marking (ref section 5.5.1). Since each generalized marking belongs to a different variable, the simplest idea could be to combine the sets of different generalized markings in one formula, to one set and quantify over that set. This creates a shorter FOL formula.

- If there are multiple FOL formulae which have the same variable, same predicate and same argument signature but different sets of markings then, these can be generalized to create a formula that combines these markings into one bigger set. This shall reduce the number of total formulae.
Another way to reduce the number of formulae is to reduce the logically same formulae to a single formula. This is discussed in the next section.

### 9.2.2 Logical Reduction

With the formulae generated as an output of the proposed system, it was observed that there are many formulae that convey the same logical meaning but are syntactically different. This difference can be used to reduce the number of formulae further. The possible approaches to logically reduce the number of formulae are mentioned below:

- **Predicate definitions** can be used to understand what another FOL formula logically means. This understanding from multiple FOL formulae could be used to create a FOL formula that generalizes the meaning of all these multiple FOL formulae. The use of automated theorem provers could be useful, but at this point the exact methodology is not explored.

- If a formula is generated that quantifies over all possible markings of a variable then the markings can simply be dropped and the quantification can be made directly over the variable. This creates a stronger logical formula.

### 9.2.3 Improved Inputs

The inputs of Marked scenario data and Background knowledge can be further improved. The marked scenario data currently does not allow expressing types of data headings. It essentially means that currently all data headings in the marked scenario data are of the same type. This can be changed in the future to allow different types. Attributing types to the data headings will also mean that the predicates must also accept typed variables. This would mean that changes to the background knowledge would also be required. The background knowledge must also allow defining typed arguments.

Creating types for data headings and predicate arguments would essentially reduce the number of possible combinations that need to be checked for each relationship model. This would mean the exploration can be concluded faster because type compatibility will already reduce the number of possible combinations.

Another advantage of adding types could be that complex functions could be described by a type system and then these types could then be used in the data headings. This would definitely increase the complexity of the analysis but it shall, in theory, add much more flexibility in terms of the data that can be entered in the data headings.

Also adding types such as lists or arrays for the data headings would also allow a much easier representation of information in the scenario data.

Another addition that could improve the inputs would be to allow predicate definitions to use pre-defined predicates. This would allow much richer and complex predicate...
definitions to be used for the creation of formal specifications. This will result in producing concise formulae that convey a very specific meaning to the engineer.

9.2.4 Kripke Structures

The creation of Kripke structures can be improved in various ways. This is discussed here.

Multiple Initial States

Currently each data sequence has only one first state. Therefore the Kripke structure generated from this sequence will only have one initial state. This can be improved by combining multiple Kripke structures that are generated from the same scenario data file. This is possible because each scenario data file can contain multiple data sequences.

The Kripke structure from the same scenario data file represent the same set of requirements and therefore it is logical to combine them to form one Kripke Structure. When multiple Kripke structures are combined, the resultant Kripke structure would have multiple initial states.

Undefined Marking

Undefined markings represent an interrupt in a system. These are represented by an undefined marking for a data row. Each undefined marking can only occur once in a data sequence. This means that the interrupt is only described by one state. The marking syntax can be extended to allow representing interrupts that are more than one data step long and therefore are represented by more than one state.

This is a more realistic representation of interrupts in a system. This will increase the complexity of analysis but at the same time it will make the generated Kripke structure represent the underlying system more closely. This trade-off decision can be left up to the requirements engineer but it should be supported by the marking syntax and semantics.
Bibliography


BIBLIOGRAPHY


[34] Dick, J., Hull, E., Jackson, K.: Introduction, pp. 1–32. Springer International Publishing, Cham (2017), https://doi.org/10.1007/978-3-319-61073-3_1


BIBLIOGRAPHY

verification for reliable software. Workshop on AUTOMATED VERIFICATION OF CRITICAL SYSTEMS (2018)


[66] Ponsard, C., Dieul, E.: From Requirements Models to Formal Specifications in B

[67] RTCA Inc. and EUROCAE: Formal Methods Supplement to DO-178C and DO-278A (Dec 2011)

[68] RTCA Inc. and EUROCAE: Software Considerations in Airborne System and Equipment Certification (Dec 2011)


BIBLIOGRAPHY


Glossary

atomic proposition  An atomic proposition is a propositional logic statement that does not contain any logical connectives i.e. it cannot be broken down into further propositions.

avionic  aviation + electronic: electronic systems used in the aviation industry.

Backus-Naur Form  It is a notation technique for specifying grammars. These are often used to describe the syntax of formal languages.

Catastrophic  Failure Conditions, which would result in multiple fatalities, usually with the loss of the airplane[68].

concurrent system  “Concurrent systems are systems comprising a collection of independent components which may perform operations concurrently — that is, at the same instant of time” [74].

exploratory testing  “It is an approach to test software without pre-designed test cases. It consists of simultaneous learning, test design and test execution ”[4].

Hazardous  Failure Conditions, which would reduce the capability of the airplane or the ability of the flight crew to cope with adverse operating conditions to the extent that there would be: (1) A large reduction in safety margins or functional capabilities (2) Physical distress or excessive workload such that the flight crew cannot be relied upon to perform their tasks accurately or completely, or (3) Serious Of fatal injury to a relatively small number of the occupants other than the flight crew. [68].

Kripke structure  Is a model of a system where the possible states of the system are represented as nodes and transitions between the states as directed arrows between the nodes. Each node also contains the properties that are true in that state. This is discussed in detail in section 2.4.

Major  Failure Conditions, which would reduce the capability of the airplane or the ability of the crew to cope with adverse operating conditions, to the extent that there would be, for example, a significant reduction in safety margins or functional capabilities, a significant increase in crew workload or in conditions impairing crew
efficiency, or discomfort to the flight crew, or physical distress to passengers or cabin crew, possibly including injuries.[68].

**Minor** Failure Conditions which would not significantly reduce airplane safety, and which involve crew actions that are well within their capabilities. Minor Failure conditions may include, for example, a slight reduction in safety margins or functional capabilities, a slight increase in crew workload, such as routine flight plan changes, or some physical discomfort to passengers or cabin crew.[68].

**No Safety Effect** Failure Conditions that would have no effect on safety; for example, Failure Conditions that would not affect the operational capability of the airplane or increase crew workload.[68].

**recall** Consider a set of relevant items and irrelevant items. Recall is the ratio of correctly identified relevant items from a set of items to the total amount of relevant items in that set. It is typically used in context of machine learning techniques but it is a fairly general idea.

**SysML** “Systems Modelling Language is a general-purpose architecture modeling language for Systems Engineering applications” refer www.sysml.org.

**transition system** It is any system that is made up of states and transitions between those states. It can be labelled or unlabelled. A labelled transition system also has a set of label for each state. The label can represent any property about that state e.g input expected for any transition to occur, conditions that must occur for a transition to this state and anything else. Labelled transition systems were first called named transition systems[48].

**UML** Unified Modeling Language is a general purpose modeling languages used to model systems. Refer www.uml.org.

**unit test** “Unit testing is a technique applied by individual software developers to ensure that the smallest, self-contained pieces of code function as designed and provide the correct results.”[31].