Department of Computer Science

Submitted in partial fulfilment for the degree of MEng.

A Formal Requirements Engineering Toolkit

Oliver Dixon

25th April 2026

Supervisor: Simon Foster

To my fiancée, Maia,
with gratitude and love.

Acknowledgements

I would like to thank the following individuals for their assistance and support offered during the preparation of this work.

- Maia Kuszyk-Whittall, for her unconditional love and support during times of difficulty, involving C++ or otherwise;
- Simon Foster, for his unwavering support and expertise provided through academic supervision during the course of this project;
- Mark Dixon, for initiating my interest in Computer Science, Mathematics, and Software Engineering;
- Tim Gaskins, for providing a basis for my professional career in Software Engineering.
- Edward Poulter, for his continued conversations on formal verification in requirements engineering, and its potential applicability outside of software;
- John Murphy, for his amusing critiques of formal methods; and
- Desmond Morris, for his belief in, and enthusiasm for, early-career software engineers.

Of course, there are countless more people who have, knowingly or unknowingly, had an enormously positive impact on my personal, professional, and academic characteristics.

Table of Contents

Executive Summary ix

Statement of Ethics xi

1 Introduction and Literature Review 1
- 1.1 The Software Development Lifecycle . . . 1
- 1.2 Formality in Requirements Engineering . . . 2
- 1.3 Survey of Literature . . . 3
  - 1.3.1 Analysis and Critique of Existing Solutions . . . 4
  - 1.3.2 Academic State-of-the-Art . . . 5
- 1.4 Development of a New System . . . 9

2 Methodology and Planning 10
- 2.1 Introduction . . . 10
- 2.2 Overview of Project Methodology . . . 10
- 2.3 Discussion of Process Models . . . 11
- 2.4 Discussion of Development Methods . . . 12
- 2.5 Project Requirements . . . 13
- 2.6 Timeline Planning . . . 13

3 Design and Implementation 16
- 3.1 Selection of Technology Stack . . . 16
- 3.2 Front-End Graphical User Interface . . . 16
- 3.3 Storage and Manipulation of Expressions . . . 16
  - 3.3.1 FOL AST Design and Implementation . . . 17
  - 3.3.2 Lexing, Parsing, and Constructing FOL ASTs . . . 18
  - 3.3.3 FOL AST Visitors . . . 18
  - 3.3.4 Query-Driven Inference . . . 21
- 3.4 Test Integration and Automatic Reporting . . . 22

4 Testing and Evaluation 23
- 4.1 Software Validation . . . 23
  - 4.1.1 Performance and Correctness . . . 23
  - 4.1.2 Satisfaction against Requirements Criteria . . . 23
- 4.2 Application to Case Studies . . . 25
  - 4.2.1 Autonomous Chemical Detector . . . 25
  - 4.2.2 Submarine Sonar Controller . . . 27
  - 4.2.3 High-Voltage Controller . . . 28
- 4.3 Overall ATP Performance . . . 29
- 4.4 Conclusions and Future Work . . . 30

A ATP Implementation Details **31**
A.1 Further Details on CNF 31
A.2 Clause Storage and Resolution 31
A.3 Logging Samples 32
A.4 Unit Tests 36
A.5 FOL Expression Grammar 36

B Sample Report **40**

C Notes on Planning and Testing **41**
C.1 Latter Planning Stages and the Spiral SDLC 41
C.2 Case Study FOL Interpretations 41

Bibliography **46**

List of Figures

1.1 Typical representations of the Waterfall and Agile SDLCs. . 2
a The incremental stages of a classical Waterfall model. 2
b The cyclical stages of a classical high-level Agile model. 2

2.1 The project plan covering the initial research phases and construction of the first prototype. 15
a The initial research period and beginning of the ATP subsystem prototype. 15
b The second phase of the ATP subsystem prototype. 15
c Research and development plan for unit-testing integration. 15

3.1 An abridged mutable sentences and terms AST class model. 19
3.2 An abridged immutable sentences and terms AST class model. 20
3.3 Generated MSRG of an MP argument. 22

4.1 Enterprise functionality including reporting and testing. . . . 26
a Defining a requirement with tests. 26
b Generating a LaTeX report. 26
4.2 Entry of a MP basis into the software. 26
4.3 Reviewing failed unit tests in the *Testing and Compliance* view. 26
4.4 Automated deduction on CKC formulae. 27
4.5 Entry of the HVC case study into the software. 28

A.1 Sample prefix tree layout for a four-feature FVI clause store containing clauses C_1, C_2, C_3, C_4. 36

B.1 A sample Requirements Index from a CKC subsystem KB. . 40

C.1 Gantt charts covering report-generation and final testing. . . 41
a Development plan for report-generation. 41
b The latter project stages, reserved for testing and hardening. 41
C.2 The Spiral software development method. 42

List of Tables

2.1 Project Requirements Index 14

4.1 ATP performance metrics on known KBs. 29

A.1 The seven-stage CNF normalisation pipeline. 32
A.2 An abridged index of selected unit tests. 38

C.1 FOL interpretation of the ACD requirements. 43
C.2 FOL interpretation of the synthesised SSC case study requirements. 44
C.3 FOL interpretation of the HVC requirements. 45

List of Abbreviations

ACD Autonomous Chemical Detector (Case Study)

API Application Programming Interface

ASCII American Standard Code for Information Interchange

AST Abstract Syntax Tree

ATP Automated Theorem-Proving (or, Automated Theorem-Prover)

BNF Backus-Naur Form

CASE Computer-Aided Software Engineering

CKC *Curiosity Killed the Cat* (Example axiomatic basis)

CNF Conjunctive Normal Form

CSP Constraint Satisfaction Problem

DSR Design Science Research

DVI Device Independent Format

FM Formal Methods

FOL First-Order Logic

FRET Formal Requirements Elicitation Tool

FVI Feature Vector Indexing

GTK GNOME Toolkit

GUI Graphical User Interface

HTML Hypertext Markup Language

HVC High-Voltage Controller (Case Study)

IR Intermediate Representation

KB Knowledge Base

LALR(1) Single-token lookahead, left-to-right (Parser)

LHS Left-Hand Side

LL(*) Unbounded-token lookahead, left-to-right, with leftmost derivation (Grammar, or a parser for such a grammar)

LOC Lines-of-Code

LSEPI Legal, Social, Ethical, and Professional Issues

MDE Model-Driven Engineering

MoSCoW Must-Have; Should-Have; Could-Have; Won't Have (Prioritisation technique)

MP *Modus Ponens* (Example axiomatic basis)

MSRG Minimally Spanning Refutation Graph

OSI Open Source Initiative

PDF Portable Document Format

PI Programme Increment

PLM Product Lifecycle Management (equivalently, Project Lifecycle Management)

RE Requirements Engineering

RHS Right-Hand Side

RPR Resolution Potential Rating

SDLC Software Development Lifecycle

SOLID Single-Responsibility; Open-Closed; Liskov Substitution; Interface Segregation; Dependency Inversion (Software development principles)

SSADM Structured Systems Analysis and Design Method

SSC Submarine Sonar Controller (Case Study)

TCP Transport Control Protocol

UML Unified Modelling Language

VLE Virtual Learning Environment (https://vle.york.ac.uk)

Executive Summary

This project investigates, constructs, and evaluates a Requirements Engineering (RE) software solution complete with formal verification and reasoning capabilities through the use of a purpose-built First-Order Logic (FOL) inference engine. Coupled with the utilities of formal verification, the product aims to address problems identified with the existing academic state-of-the-art software, such as a lack of enterprise-grade features expected in industrial settings, with a principal focus on interoperability with standard tooling and processes.

Industry-standard RE workflows are unscientific in the majority of cases, which has, since at least 1975 been identified as a major (often primary) source of failures and delays in software engineering processes. In unscientific workflows, requirements at the systems and software levels are specified in an unstructured natural language and seldom subject to any sort of verification beyond peer-review.

Despite the general absence of Formal Methods (FM) practice in the wider software domain, certain industries and products require a markedly increased level of assurance. These nominally relate to contracts in defence, aerospace, transportation, energy, and any instances of critical national infrastructure. Where FM are used in early stages of development workflows, engineering teams have reported frustrations with substandard tooling lacking features standard in other engineering toolkits, such as integrated development environments and records-management systems; these shortcomings have been consistently implicated in relatively high initial costs. From these frustrations, a Research Question was posed:

Research Question. Can a FOL inference engine utilising Automated Theorem-Proving (ATP) be integrated with enterprise-grade business capabilities to provide an effective platform for modelling and managing software requirements?

To empirically answer the Research Question, *Optifol* was developed over a period of several months using a spiral software development lifecycle method consisting of repeated risk assessment and evolutionary restructuring of project priorities and goals. The overarching goal was the development of a solution that is suitable for use by software engineering teams to make accessible the benefits of mathematical rigour in RE, which has been the subject of substantial investigation by the academic community

over the past decades, unified with enterprise-grade functionality to reduce the labour requirements and initial cost of work.

A wide array of software engineering techniques were applied during development iterations including model-driven engineering, unit testing, fuzzing, static and dynamic analysis, and construction of extensive documentation.

Fundamental literature in ATP, particularly with regard to FOL, was investigated to guide the design and construction of an expression parser and pre-processor. From those foundations, an optimised ATP was designed and implemented in the C++ programming language. Relevant ATP literature was reviewed to adapt the prover to the RE domain, eventually incorporating aspects from current industry-standard provers.

In addition to the prover core, a cross-platform graphical front-end was developed to support the indexing and management of projects with formally verifiable requirements. Complementing the verification capabilities, the front-end assists engineering users throughout the entire development lifecycle from requirement specification to unit-testing and report-generation. The final product was moderately sized, measuring between 30,000 and 35,000 Lines-of-Code (LOC).

Three case studies: two adopted from existing literature, and one synthesised for the purposes of evaluation, were examined using the software; in all cases, the software was capable of managing the requirements and decidedly proving consistency on the corresponding Knowledge Bases (KBs). The ATP performance was also quantitatively assessed to demonstrate the advantages of various optimisations and heuristics utilised by the implementation, and dynamic analysis was undertaken to identify performance-critical portions of the algorithms which could be targeted for optimisation.

The project was largely successful. All major requirements specified at the outset were met. Development of the software provided valuable insight into the posed research question. The software served to demonstrate the manners by which ATP can be pragmatically unified with industrial integrations and existing workflows to capture project requirements in a clearly presentable and formally verifiable manner.

Optifol_

Inspiring Confidence in Engineering

The *Optifol* source code archive as of 25th April 2026 is packaged with the submission bundle uploaded to the VLE. A mirrored repository will be available online at https://github.com/oliverdixon/OptiFOL-Software ([1]) following completion of the assessment process.

Statement of Ethics

This project has followed all correct ethical procedures. No non-trivial risks were identified, and ethical approval was not sought. Legal, Social, Ethical, and Professional Issues (LSEPI) are discussed in Section 2.2.

1 Introduction and Literature Review

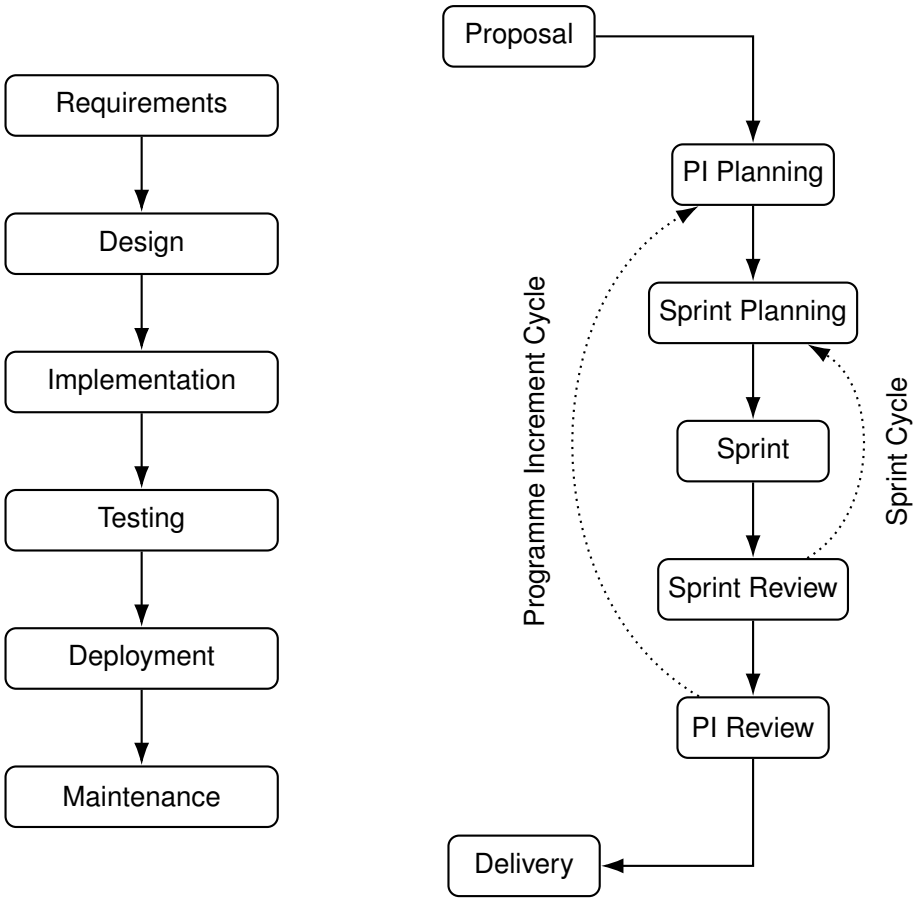
1.1 The Software Development Lifecycle

In 1956, H. D. Benington of the Lincoln Laboratory at the Massachusetts Institute of Technology proposed techniques for the systematic development of *Large Computer Programs for Control and Processing* [2]. Later to become known as the Waterfall Model (pictured in Figure 1.1a), Benington's proposal formed the basis of the Software Development Lifecycle (SDLC): a template for systematic processes by which complex software should be specified, designed, implemented, and tested. Since the 1956 symposium, tens of new SDLCs, frequently centred around the Agile system of SDLCs[1], have been described by academic and industrial leaders to document the engineering processes of increasingly elaborate software [3].

RE is notably present in all SDLCs, and often appears as the initial stage following Business Modelling. RE involves the procurement of system requirements, segregation into functional and non-functional categories, and translation into concrete specifications. Consequently, the results of RE form a crucial component of the technical contract between customer and developer, defining the precise nature of the solution to be developed and delivered [5].

Despite progress in other stages the SDLCs, RE has historically been problematic due to "lack of standard methods, lack of suitable formalisms, and lack of quality tooling" [6]. The enormous cost of poorly executed RE has, since at least 1975, been consistently documented throughout the preceding decades of software engineering literature, and is commonly implicated as "the part of the [software engineering] process which leads to more failures than any other" [7]. Contemporary literature indicates that inadequacies in the early stages of SDLC continue to cause costly failures, despite over four decades of innovation [8].

[1] Agile-based SDLCs are often implemented by a two-stage iterative process, commonly characterised by Programme Increment (PI) within which short Sprint sessions are performed and discussed at daily problem-solving meetings between engineers. Most tasks (RE, design, implementation, and testing) are performed during the Sprint. A high-level view of this process is shown in Figure 1.1b. The definition of the Agile process is constantly changing and emphasises adaptability over adherence to procedure [4].



(a) The incremental stages of a classical Waterfall model.

(b) The cyclical stages of a classical high-level Agile model.

Figure 1.1: Typical representations of the Waterfall and Agile SDLCs.

1.2 Formality in Requirements Engineering

Attempts to resolve problems associated with RE through the use of formal methods was recorded as early as the 1980s [9], with the wider use of mathematical rigour in the SDLC twenty years' prior in the 1960s [10].

Non-formal techniques can certainly be applied to RE processes to attain characteristics of good practice, including enhanced multi-round peer-review procedures, external moderation, and improved standards of communication between contract parties. Notwithstanding those alternatives, this report hypothesises that effective application of FM will provide the most consistent basis for desirable engineering outcomes[2].

[2]Relevant literature reports that empirical evidence indicates use of FM in any stage of the chosen SDLC more favourable outcomes, compared to non-falsifiable techniques due to formal verification, enhanced traceability, and improved change management throughout the engineering lifecycle [11].

This report further hypothesises that the use of FM within the *requirements capture and analysis* phase of RE will enable the construction of formal models, unambiguously modelling user requirements, and consequently allow deductive reasoning and inference within the constraints of the models. Under a sufficiently expressive logic, automatic verification of desirable properties is possible.

These hypotheses form the base of the *Research Question*, posed in the Executive Summary and restated below. The remaining sections of this report will investigate and discuss this topic in detail.

> **Research Question.** Can a FOL inference engine utilising ATP be integrated with enterprise-grade business capabilities to provide an effective platform for modelling and managing software requirements?

For compatibility and mutual understanding with the existing literature, Definition 1.1 defines terms used frequently henceforth.

Definition 1.1 (Standard terms)**.** *A* requirement *is an unambiguous statement, expressed in a chosen logical model, representing a desired behaviour of a system. Requirements may be specified formally as* sentences *in FOL, and multiple sentences may form a KB.*

A KB is consistent *if there is a model, or assignment of variables, such that all sentences in the KB are true; such individual requirements are thus said to be* satisfiable*.*

1.3 Survey of Literature

A 2012 meta-analysis and survey of RE identified that uncertainty and potential complexity of system behaviour, communication between customers, users, and technical experts, varying levels of details in requirement specifications, and verification of the deliverable system are repeatedly responsible for poor engineering outcomes [6]. The analysis lists, from relevant literature in software economics, five desirable characteristics of RE practice to mitigate the commonly identified shortfalls:

- “Abstraction and formality”, by providing representations that suppress irrelevant implementation detail while still admitting mathematically precise interpretation and analysis;
- “conceptual integrity”, by encouraging a single, coherent model of

system intent in which requirements are expressed consistently and contradictions can be exposed early;

- "readability and understandability", by presenting requirements in a notation and structure that support clear communication between stakeholders, even where full formalisation is reserved for critical properties;

- "range of applicability", by remaining usable across different stages of development and across systems with differing operational domains, criticalities, and levels of complexity; and

- "support by appropriate tools", by enabling automation for activities such as consistency checking, traceability, validation, and verification, thereby making rigorous RE practice practical at scale.

These five aspects were used to inform the justification, design, and development of a new system.

1.3.1 Analysis and Critique of Existing Solutions

To date, a number of options for RE have been made commercially available, commonly offered by corporations marketing the tooling as a component of an existing suite of engineering tools. Each commercially available solution lacks in at least of the five desirable attributes of an RE toolkit.

- *MathWorks Requirements Toolbox*: a requirements authoring tool enforcing tight integration with *MathWorks MATLAB* and *MathWorks Simulink* models, providing a wide range of capabilities including verification, modelling, and traceability; used in non-software engineering environments where physical phenomena is modelled [12].

- *IBM Rational DOORS*: the Digital Object-Oriented Requirements System from IBM, allowing a high degree of complex requirement organisation, distributed over substantial corporate network environments, and integration with other engineering tools, such as *IBM Rational Rhapsody*, but lacking any native modelling or automatic verification capabilities [13].

- *D-Risq Kapture*: a modularised component of the *D-Risq Modelworks* engineering suite of tools, capable of indexing and verifying formally specified requirements while allowing for tracing of software-level requirements to system-level requirements. *Kapture* offers integration with *MathWorks* and *IBM* solutions and has a limited export capability for report-generation. Inference is performed within a typed temporal-

style logic [14].

- *dSPACE SYNECT*: domain-specific data-management and testing tool for vehicle Electronic Control Units, providing capabilities to automatically convert formally specified requirements to executable software unit tests. Inference is provided using a typed first-order logic with a data dictionary and pre-defined templates [15], [16].

Within the landscape of commercially available RE tooling, solutions are often prohibitively integrated within existing suites of applications, offer limited inferential capabilities in contexts where FM may be advantageous for regulatory approval, or become highly domain-specific for non-trivial models. Production of accurate, representative models may also prove to offer diminishing returns due to the difficulty of specification for inexperienced engineers, as even tools marketed as accessible scale poorly on larger requirement bases and require bespoke consultancy from experts.

An undated case study published by D-Risq reported a three-week effort by a single engineer specifying 23 system requirements for a DO-178C Level A-certified flight control system required input from an external Kapture expert user and in-house Kapture developer, and resulted in over 700 symbols and 400 functions being defined for the model [17].

1.3.2 Academic State-of-the-Art

Capabilities of Contemporary Academic Tooling Extensive work has been undertaken in academia to construct suitably expressive logical models for the application of FM in RE. Often, though not always, RE-centric models, associated inference engines, and engineering front-end applications are built and marketed to industries concerned with the specification and development of software likely to benefit from the application of FM in stages of the SDLC other than technical implementation (where the job of formal verification is done transparently by compilers and Computer-Aided Software Engineering (CASE) tools). Typical industrial customers include large corporations developing software in safety-critical, performance-critical, or strictly regulated industries[3].

In contrast to the commercial domain, academic investigations into applications of FM in RE have notably advanced within recent years. In addition to development of expressive models, substantial investigations have been done to enhance accessibility and theoretical usefulness of FM. There are numerous active areas under investigation in this respect:

[3] See, for example, [18] and [19] for details of extensive regulatory requirements of FM in the aerospace software sector.

1. Transformation of formal requirements models into a system or software design model (or prototype thereof). These transformations can accelerate the SDLC following the RE construction process, although challenges have been identified in proving equivalence of the produced models. *RM2PT* is a reference implementation of a standardised transformer [20], and some commercial toolkits, notably *D-Risq Kapture*, have the ability to integrate with third-party tooling to generate production code skeletons and executable test harnesses.

2. Transformation of structured natural language into a formal or semi-formal requirements model. These transformations accelerate the beginning of the chosen SDLC, and reduce the demands on the expertise of systems engineering personnel. Work has also been done to further transform the natural-language requirements descriptions into executable test cases specifying the contracts inferred by the requirements. *NAT2TEST* is a prototype of a system supporting specification of requirements through structured language [21], [22].

3. Domain-specific application of FM in RE, such as the *ST-Tool* for formal verification in cryptographic engineering of protocols consisting of formalised requirement models.

4. Visualisation and graphical depictions of formal models and inference procedures, allowing animation and debugging of model characteristics. The Formal Requirements Elicitation Tool (FRET) from NASA provides a modern, well-known implementation of an ATP integrated with a system to graphically explain its deductive process; this has immediate uses in teaching [23].

5. Optimisation (through identification and removal of redundant formulae) of formally specified requirement models. *ReqV* is capable of optimisation, and provides an assurance of minimality in addition to correctness and consistency [24].

Reliable model transformations address some shortcomings identified by current offerings in the commercial sector, but remain largely inaccessible to industrial customers due to limited, or non-existent, implementation of the discussed approaches. When software implementations are realised, they often act as prototype reference implementations, or proofs of concept, and are not architected with the intention of large-scale use in an industrial environment, and frequently lack enterprise-level capabilities such as report-generation and support for distributed working. A comprehensive survey of all known software-driven FM/RE toolkits is available in [25].

Advances in Automated Theorem-Proving Further to the development of academic applications which support FM/RE, advances in saturation-

based ATP have made practical the application of ATP on very large KBs of formally specified system requirements. Due to these advances, requirement specifications forming KBs may be subject to iterative application of a pair of complementary reduction rules known as *unification* (Definition 1.2) and *resolution* (Definition 1.3) [26]. Together, these rules are known to form a refutation-complete inference procedure[4] that is sufficiently flexible to provide a computational basis for determining satisfiability over any predicate KB [27].

Previous runtime constraints, such as the exponential runtime time complexity of the algorithm demanded by Definition 1.2, have recently been discovered to be runnable in near-linear time [28]. Several innovations have also been made in the field of formula normalisation, or more precisely, the algorithms and data structures required to transform arbitrary FOL expressions into a normal form, such as the Conjunctive Normal Form (CNF) given by Definition 1.4, to minimise and precompute the bound for the maximum number of clauses in the CNF representation [29], [30].

Definition 1.2 (Unifier; unification; most general unifier)**.** *A* unifier σ *for FOL clauses* L_1 *and* L_2 *is a set of variable-to-term substitutions*

$$\sigma := \{[v_1 \mapsto t_1], \ldots, [v_n \mapsto t_n]\} \tag{1.1}$$

such that $\sigma(L_1) \equiv \sigma(L_2)$*, where* $\sigma(\cdot)$ *denotes application of all substitutions described by* σ *to its argument. The number of individual substitution map entries is denoted by* $|\sigma| := n$*.*

Unification *is the procedure by which unifiers are located. Where multiple unifiers* $\sigma_1, \ldots, \sigma_m$ *satisfy the required conditions,* $\sigma' := \arg\min_\sigma |\sigma|$ *is termed the* most general unifier*.*

Definition 1.3 (Binary resolution)**.** *Consider clauses* $L_1 = \{l_1, \ldots, l_n\}$ *and* $L_2 = \{m_1, \ldots, m_k\}$ *which do not depend on any common variables. If there exists a (most general) unifier* σ *such that* $\sigma(l_i) = \sigma(\neg m_j)$ *for some* $l_i \in L_1$ *and* $m_j \in L_2$*, then* binary resolution *dictates that the unified-complementary literals* l_i *and* m_j *can be eliminated under* σ*:*

$$\frac{l_1 \vee \ldots \vee l_n; \quad m_1 \vee \ldots \vee m_k}{\sigma\left(l_1 \vee \ldots \vee l_{i-1} \vee l_{i+1} \vee \ldots \vee l_n \vee m_1 \vee \ldots \vee m_{j-1} \vee m_{j+1} \vee \ldots \vee m_k\right).}$$

[4] The iterative application resolution is a *refutation-complete* formal system $\mathcal{S}$ due to its ability to derive a contradiction from any unsatisfiable set of axioms Γ_0. That is, $\Gamma_0 \models_{\mathcal{S}} \bot \implies \Gamma_0 \vdash_{\mathcal{S}} \bot$. Resolution does not necessarily terminate over a consistent knowledge base, and for this reason is typically bounded by a maximum execution time.

Definition 1.4 (Conjunctive Normal Form; clauses; literals)**.** *A sentence is in CNF if and only if it is written as a set of* clauses *under conjunction, each of which is expressed exactly as disjunctions of* literals*. That is, some CNF sentence S takes the form*

$$S = C_1 \wedge \ldots \wedge C_n \tag{1.2}$$

$$= \left(L_{1,1} \vee \ldots \vee L_{i(1),1}\right) \wedge \ldots \wedge \left(L_{1,n} \vee \ldots \vee L_{i(n),n}\right) \tag{1.3}$$

where $L_{t,s}$ is the t-th literal in the s-th clause and $i(j)$ denotes the number of literals in the j-th clause. Alternatively, S may be written as a nested unordered set, with the outermost dimension collecting clauses under conjunction, and the innermost dimension collecting literals under disjunction:

$$S = \{C_1, \ldots, C_n\} \tag{1.4}$$

$$= \left\{\left\{L_{1,1}, \ldots, L_{i(1),1}\right\}, \ldots, \left\{L_{1,n}, \ldots, L_{i(n),n}\right\}\right\}. \tag{1.5}$$

In addition to algorithmic improvements, recently discovered innovations in Feature Vector Indexing (FVI) (Definition 1.6) and *clause subsumption* (Definition 1.5) can drastically simplify large KBs by identifying and eliminating implications between sentences. In particular, [31] describes a novel self-simplifying data structure based on prefix trees and the use of clause feature functions to heuristically organise KBs according to resolution potential and "semantic similarity", which recent reviews of implementations such as the well-known VAMPIRE ATP project have demonstrated substantially improved runtime performance [32], [33].

Definition 1.5 (Clause θ-subsumption)**.** *A clause L_1* θ-subsumes *a clause L_2 if there exists a variable-to-term substitution map θ such that $\theta(L_1) \subseteq L_2$, where $\theta(\cdot)$ is an applicator as described by Definition 1.2. In practice, subsumption is equivalent to the deletion rule*

$$\frac{\theta(L_1) \vee \theta(L_2)\,; \quad L_1}{L_1.} \tag{1.6}$$

Definition 1.6 (Clause feature function; Feature Vector Indexing)**.** *A* clause feature function*, or simply* feature*, is a function mapping that produces a numerical characteristic of a clause. Features useful in determining resolution potential are f: Clauses $\to \mathbb{N}$ such that (C_1 subsumes C_2) $\implies f(C_1) \leq f(C_2)$; such features are termed* subsumption-compatible*.*

FVI *is the practice of using numerical features to order clauses within a data structure.*

1.4 Development of a New System

Although both commercial and academic communities have produced a broad range of RE tools, the literature reveals discrepancies between the academic state-of-the-art and the current availability of tooling to engineering teams. Commercial tools such as *IBM DOORS* or *dSPACE SYNECT* provide mature requirements management workflows but offer little or no capability for formal reasoning, relying instead on informal review processes. Conversely, academic prototypes frequently demonstrate strong theoretical capabilities, such as typed logic inference, natural-language transformation, or model optimisation, but are rarely engineered for scalability, maintainability, or integration into industrial toolchains.

This fragmentation results in a persistent gap: no existing solution delivers an expressive, domain-independent formal modelling environment that also satisfies the usability, interoperability, and organisational requirements of modern engineering practice. It is apparent from [6], [25] that existing tools typically lack one or more of the following features:

- A FOL engine with deductive inference capable of automatically identifying logical inconsistencies;
- Descriptive reasoning outputs suitable for debugging, certification, and audits;
- Enterprise-grade features such as robust reporting, integration with existing engineering software infrastructure, and cross-platform graphical interfaces; and
- Scalable tooling that avoids the heavy modelling overheads reported in industrial case studies.

Consequently, there is a need for a new standalone system that bridges the divide between academic innovation and industrial applicability by combining a rigorous logical foundation with the practical features required in real engineering workflows. Addressing this gap enables requirements engineers to benefit from mathematical rigour without sacrificing usability, interoperability, or productivity, and provides an immediate mapping from the toolkit capabilities to desirable RE practices defined by Partsch [6].

The remainder of this report discusses the planning, design, implementation, testing, and evaluation of the proposed system.

2 Methodology and Planning

2.1 Introduction

This chapter describes the processes used for the specification, design, implementation, and evaluation of the software. The theoretical models are discussed in three parts:

1. Project methods: the high-level methods used to orchestrate the execution of the overall project, including background research, product planning and implementation, and evaluation.

2. Process methods: the software development framework managing the lifecycle of product development through risk analysis, planning, design, implementation, and system and integration testing.

3. Development methods: the low-level techniques applied during the design, implementation, and testing phases of the process lifecycle.

The survey of literature was used to define the objectives of the system, each of which formed a distinct *software component* providing a set of capabilities for the product. The theoretical model was applied to the aggregated objectives to form a set of requirements, which, according to an ordering provided by risk analysis, informed the plan of the project SDLC.

2.2 Overview of Project Methodology

The project is experimental in nature. As a field-specific scientific investigation producing a practical artefact, the project was conducted under the Design Science Research (DSR) problem-solving paradigm, commonly identified to be suitable for management of experimental and high-risk projects [34]. Observations and informal case studies of existing RE solutions were examined prior to the execution of the process methods, however qualitative methods are generally not applicable to experimental software development due to a lack of prior art directly related to the topic under investigation. Research in existing literature was undertaken to identify capabilities not provided by existing software, or identify points where capabilities were perceived to be lacking by enterprise users.

From the outset, LSEPI were given significant consideration. Whilst research did not involve any surveys or human participation, and hence did

not require formal ethical approval, the potential legal and ethical issues of general engineering applications were noted, especially where these are implied to provide assurance relating to properties of safety-critical systems. An SDLC was chosen that emphasised risk mitigation and software was developed with the SOLID approach to permit future changes arising due to legal or professional issues to be implemented with minimal engineering effort [35]. It was also deemed crucial to depend only on third-party libraries distributed under an Open Source Initiative (OSI)-compliant licence[1] to avoid legal issues.

Although direct focus groups or surveys were not conducted, the author has years' of experience in observing the use of RE and FM tooling in industrial environments, often for large commercial systems involving thousands of engineers, which allowed for a foundational understanding of user perceptions relating to software-based RE.

Following the development of the product and construction of suitable software-level test harnesses, the success of the overall project was evaluated. High-level capabilities of the system were compared against originally stated objectives, in accordance with the MoSCoW staging approach [36], to quantitatively determine satisfaction against the starting criteria. To determine functional correctness, three case studies were considered: an Autonomous Chemical Detector (ACD) from the RoboStar research group; a High-Voltage Controller (HVC); and a synthesised case study which formalised safety and performance requirements for a Submarine Sonar Controller (SSC). Finally, the stated Research Question could be answered.

2.3 Discussion of Process Models

Given the experimental nature of the project, whereby industrial capabilities are coupled with techniques and algorithms from academia, it was deemed imperative to select a model that supports a *de-risked approach* software development. De-risking is the practice of decomposing a larger task into parts and consequently prioritising components that are more likely to fail or take excessive amounts of time beyond initial estimates.

A traditionally rigid method such as Waterfall was considered inappropriate due to the strict staging preventing effective prioritisation of the deliverable components. Other Waterfall-style process models considered include the V-model and Structured Systems Analysis and Design Method (SSADM), but were similarly not selected due to the imposition of overly strict require-

[1] OSI-compatible licences have been legally reviewed and certified to comply with the Open Source Initiative definition of Open Source Software. Compatibility indicates that the associated software may be freely used, modified, and shared.

ments and separation between stages of the development lifecycle [37]. Agile-aligned methods were also considered, including rapid prototyping and the Rational Unified Process, but were eliminated due to their unsuitability for high-risk research-driven projects where formal documentation is the single-source-of-truth, or limited applicability for experimental use-cases.

Likewise, team-oriented workflows such as SCRUM were discounted as unsuitable for a solo developer. Considering these constraints, the *spiral* process model was selected as the theoretical basis for the work due to its inherent support for risk-based prioritisation of deliverable components. Figure C.2 shows a typical visual representation of the spiral model, noting that evaluation of risk occurs during all prototype iterations, each of which carries an increased cumulative cost to resolve unanticipated issues, until the final operational prototype is achieved [38].

There were numerous disadvantages to the spiral model which were identified as a hindrance for a solo developer, including the likelihood of increased management overhead, the near-universal dependence on a realistic risk-analysis framework, and the potential for repeated (potentially infinite) iteration through pre-operational prototypes. The impact of these limitations was mitigated by careful structuring of work according to time constraints (see Section 2.6) and careful synthesis of different risk-analysis strategies.

Risk analysis naturally complements the spiral model, as it is an activity that must be undertaken repeatedly as a project progresses to accurate capture up-to-date information. During each phase of the spiral, risk analysis was composed of risk identification, risk scoring (according to a qualitative analysis computed in terms of the impact and probability), and composition into a risk register. The risk registers also included mitigations according to the impact, ranging from dropping entire subsystems, such as in the event of a feasibility test failing, to reducing the extent of a trivial integration.

Moreover, repeated risk analysis was considered necessary despite the small scale of the project due to the dependencies between subsystems: the risk profile of a deliverable component may change according to progression of prior iterations of the spiral process.

2.4 Discussion of Development Methods

In addition to the overall process model, individual strategies of Model-Driven Engineering (MDE) and unit-testing were used extensively for the implementation of the prototyped components. MDE was selected as a prototyping model due to its inherent capability to minimise coupling and allow effective reasoning over larger object models. Usage of a widely

understood development method also allowed traceability throughout the design-implement-test cycle. The Unified Modelling Language (UML) was used to implement MDE for elaboration of class models.

Continuous validation of the software permitted informed risk analysis for upcoming development cycles. For similarly sized software projects, unit tests have been noted to be especially useful when statement and branch coverage exceed 80% to 90% [39]. *Fuzzing*, the practice of injecting controlled random data as inputs, was also used given the well-researched benefits of verifying correct behaviour on invalid inputs and providing hardening opportunities for enforcing pre- and post-condition function contracts [40].

2.5 Project Requirements

Table 2.1 enumerates the project requirements categorised into subsystems which form the incremental prototypes referenced by the spiral model shown in Figure C.2. The requirements are further segregated into one of three risk profiles: low, moderate, and high, indicating the tentative probability of the requirement failing to materialise in the product[2]. Similarly, each requirement is labelled according to the necessity of its inclusion in the final product: essential, desirable, and optional. According to this framework, the combination of necessity and risk profile allow an ordering to be imposed on the realisation of each requirement such that essential, high-risk components are developed prior to optional, low-risk components.

2.6 Timeline Planning

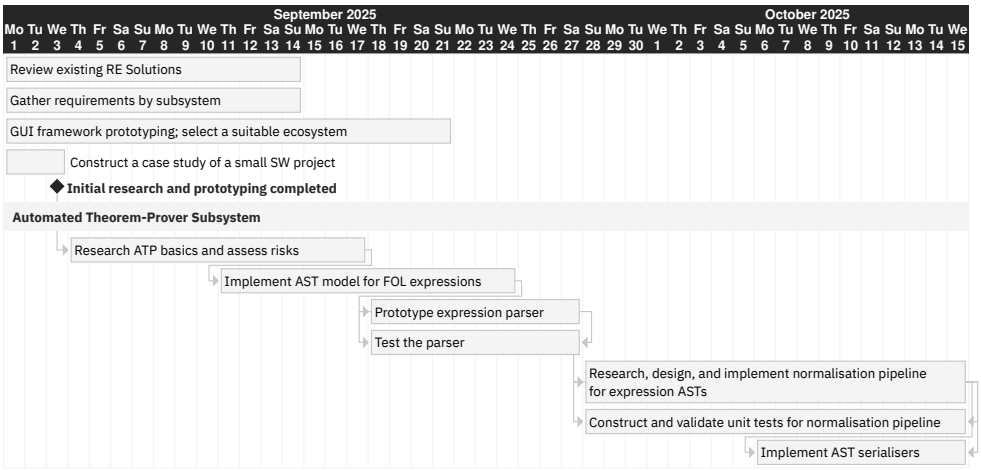
To complete the project within the allocated time of approximately one year on a part-time commitment, working time was partitioned and represented by a Gantt chart shown in Figure 2.1. Note that the chart was constructed following specification of the individual subsystems to ensure its usefulness. Timelines on the chart are relative, specified by the anticipated number of days required to complete the corresponding task.

In addition to the work depicted on Figure 2.1 and Figure C.1, prior investigation was undertaken to determine the feasibility of the project.

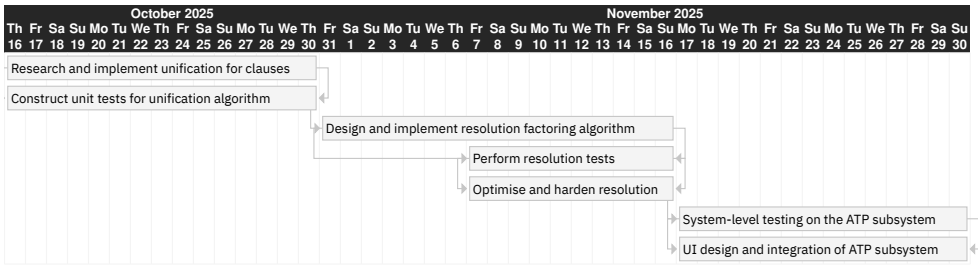
[2]The risk profile for each requirement was selected by the developer according to experience with developing similar systems. In particular, usage of Graphical User Interface (GUI) libraries and frameworks was deemed low-risk due to extensive prior knowledge of the topic, whereas implementation of an ATP engine to support automated inference was considered higher-risk given minimal existing exposure to the field.

Requirement	Necessity	Risk
GUI and management facilities		
Definition of projects and subsystems	Desirable	Low
Creation of requirements with free-text data	Essential	Low
Assignment of sentences to requirements	Essential	Low
Assignment of unit tests to requirements	Desirable	Low
Organisation of requirements by subsystem	Desirable	Low
GUI facility to manage FOL queries	Essential	Moderate
GUI facility to manage provided unit tests	Desirable	Low
GUI facility to manage generated reports	Desirable	Low
Manipulation of FOL expressions		
Preprocess and normalise FOL sentences	Essential	High
Compose KBs from numerous sentences	Essential	Moderate
Automated deductions on FOL KBs	Desirable	High
Construction of human-readable deductions	Optional	Moderate
Detection of tautologies in FOL KBs	Optional	High
Unit testing integration		
Query metadata from Google Test suites	Desirable	Moderate
Automatically execute Google Test suites	Desirable	Moderate
Query metadata from J-Unit test suites	Optional	Moderate
Automatically execute J-Unit test suites	Optional	Moderate
Report generation		
Serialisation of reports into LaTeX	Essential	Low
Serialisation of reports into HTML	Optional	Low
Serialisation of reports into plain text	Optional	Low
Inclusion of requirement metadata in reports	Essential	Moderate
Inclusion of deduction traces in reports	Optional	High
Inclusion of unit test results in reports	Desirable	Low
Accessibility and Engineering Practices		
Unit testing for high-risk requirements	Essential	Moderate
Fuzzing for high-risk requirements	Desirable	Moderate
Unit testing for moderate-risk requirements	Desirable	Low
Unit testing for low-risk requirements	Optional	Low
Native software support for GNOME systems	Essential	N/A
Support for non-GNOME UNIX-like systems	Desirable	Moderate
Support for Windows systems	Optional	Moderate
Compliance with an OSI-approved licence	Essential	N/A
Quantitative ATP performance analysis	Desirable	Moderate

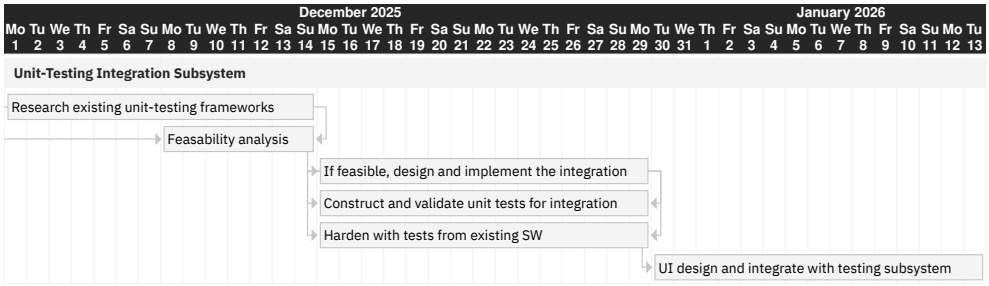
Table 2.1: Project Requirements Index



(a) The initial research period and beginning of the ATP subsystem prototype.



(b) The second phase of the ATP subsystem prototype.



(c) Research and development plan for unit-testing integration.

Figure 2.1: The project plan covering the initial research phases and construction of the first prototype.

3 Design and Implementation

3.1 Selection of Technology Stack

The application is required to support a breadth of functionality, from capabilities found in enterprise-level RE solutions such as GUI and integration with existing infrastructure, to performance-critical ATP algorithms used on the academic frontier. Due to these requirements, C++26 was selected as the principal high-level language of development[1]. Other high-level languages including Java, C#, Haskell, and OCaml were considered, but discounted due to inadequate support for one or more of the core requirements.

3.2 Front-End Graphical User Interface

The target users of the application are software, systems, networking, and hardware engineers working on medium-sized technical projects in an industrial environment where RE/FM is already an identified necessity. Therefore, a cross-platform desktop framework with native C++ integration was sought. Use of an OSI licence was also needed to prevent constraints on the linked application execution. Various well-supported candidates were considered including *Qt*, *wxWidgets*, *ImGui*, and *GTK+*. The latter was chosen due to its maturity, flexibility, existence of the well-documented *GTKmm* modern C++ bindings, lack of licensing restrictions, and support for vector graphics libraries such as *Cairo* and *Pango*.

GTKmm also provides an arbitrarily extensible object model, allowing programmers to easily define new types in the GNOME Toolkit (GTK) type system which supports garbage collection and runtime debugger capabilities. Custom polymorphic types were defined in the object hierarchy for requirements, reports, tests, and system analysis reports.

3.3 Storage and Manipulation of Expressions

User-defined requirements are composed of a unique identifier, a human-readable rich-text description, and additional metadata supported by tra-

[1] C++26 offers increased standard library support for lazily evaluated views and ranges for the implementation of binary resolution, the hive data structure for clause indexing, and saturation arithmetic for safer numerical computations [41].

ditional RE indexing and management systems. In addition to descriptive elements, requirements may be associated with a FOL sentence to support formal verification. Once a requirement is endowed with a FOL sentence, it may be associated with any number of *Analysis Groups* within the FOL KB of the project and subject to automated analysis.

3.3.1 FOL AST Design and Implementation

The infrastructure supporting expressions in FOL is the most substantial single component of the application. FOL expressions are represented as Abstract Syntax Trees (ASTs) whereby individual nodes are composed of data, and that of any children. Roughly, there are two categories of nodes in FOL expressions: *sentences* and *terms*. Sentences represent complete statements with a truth value and can be further segregated into predicates, connectives, and quantifiers. Terms are stateless objects such as variables, functions, and constants[2].

The software defines two classes of ASTs: mutable and immutable, designed to express different ownership semantics of the nodes detained by the AST. The mutable AST can be constructed incrementally and transformed arbitrarily; it also expresses a strong model of ownership, whereby AST nodes hold single responsibility over their child nodes, and any transfer in ownership must be explicit. The strong-owning mutable AST elaborated in UML by Figure 3.1, noting the expression of strong ownership with compositional aggregation on child relations.

The immutable AST model, elaborated by Figure 3.2, expresses a weak model of ownership, whereby AST nodes maintain observing references to their child nodes and do not hold any responsibility over object lifetimes. Sentences and terms are owned by a globally accessible *symbol repository*, which provides strong lifetime guarantees, uniqueness, transparent hashing, stable pointers, and stable iterators.

Crucially, formulas held in the immutable representation are expected to be expressed in CNF; this invariant allows the design of an API consistent with the standard mathematical reference, in particular the set notation given in Definition 1.4. In the UML class models, precise ownership semantics are indicated by the «Outbound», «Inbound», and «Observe» stereotypes.

The C++ instantiation of the modelled ASTs makes use of features from modern iterations of the language to semantically express ownership and verify constraints at compile-time. In particular, std::unique_ptr is used to model strong ownership, and const-qualified raw points are used to model

[2] Constants are expressed as zero-arity functions due to the identical semantics.

```
[[nodiscard]] const Node * Node::observe() const noexcept;
[[nodiscard]] std::unique_ptr<Node> Node::take() noexcept;
void Node::put(std::unique_ptr<Node> &&node) noexcept;
```

Listing 3.1: Three member functions to model different ownership scenarios.

observing references. Listing 3.1 provides a sample of member functions implementing the three ownership models, taking note of const-correctness and exception guarantees.

3.3.2 Lexing, Parsing, and Constructing FOL ASTs

A lexer and parser were used to translate structured human-readable text expressions into mutable ASTs. In particular, the Flex lexer was used to tokenise expressions based on a syntactic specification, and the Bison parser was used to parse the tokenised streams and incrementally construct the polymorphic mutable trees.

Both Flex and Bison are *code-generating parsers*: given a C/C++ template file, and the relevant token or grammar specification file, a preprocessor generates C/C++ translation units usable by the client application. As templates may contain arbitrary code that is injected into the generated translation units, the use of Flex and Bison allowed direct construction of the AST using the types defined by Figure 3.1. Listing A.3 shows a Bison grammar segment defining the construction of a mutable universally quantified or predicate sentence.

Other C++-compatible lexer and parser generators were considered, especially ANTLR and BtYacc, but Flex and Bison were preferred as they minimised additional complexity in the build process and supported the LALR(1) grammar required for FOL expressions[3].

3.3.3 FOL AST Visitors

To efficiently traverse arbitrarily deep ASTs, the visitor pattern was used extensively due to the recursive and polymorphic nature of the node structure. Use of the visitor pattern in any double-despatch language has further documented benefits, such as intuitive delineation between the paradigms of object definition (declarative) and object manipulation (imperative or functional) [43]. In the implementation, two major categories of visitors were used: *mutating visitors* and *observing visitors*.

[3] ANTLR uses recursive descent to parse LL(*) languages [42], but this was excessive for simple FOL expressions. The LALR(1) grammars parsed by Bison were ideal.

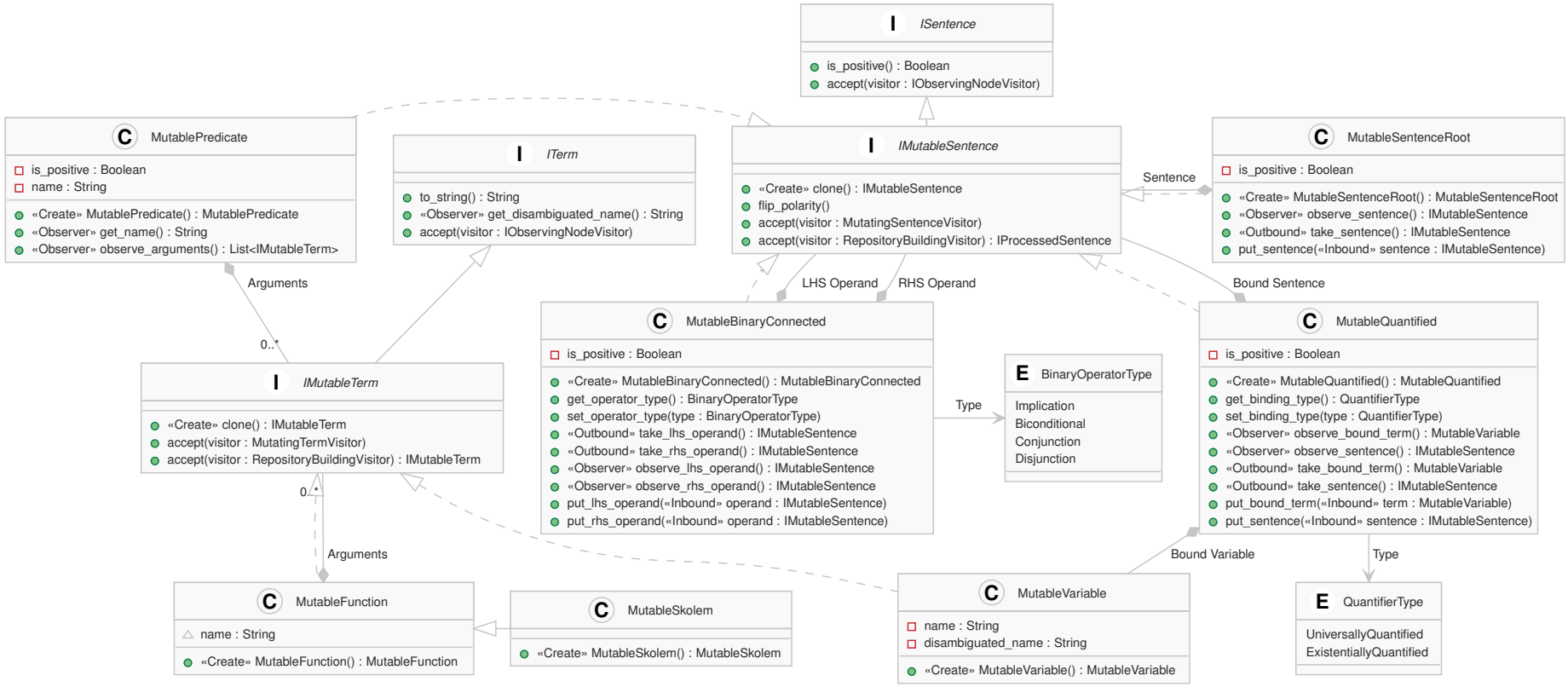


Figure 3.1: An abridged mutable sentences and terms AST class model.

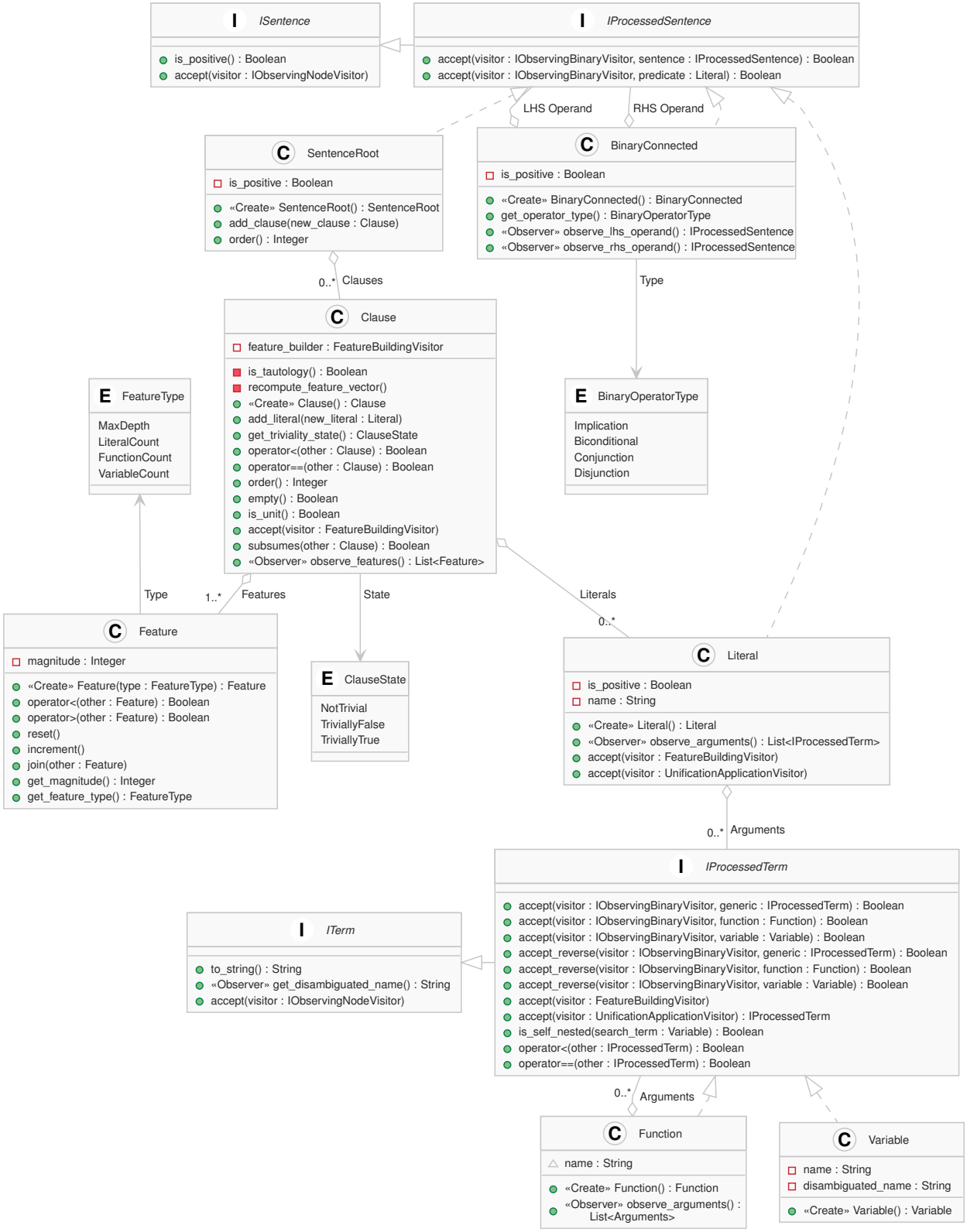


Figure 3.2: An abridged immutable sentences and terms AST class model.

Mutating Visitors The most interesting set of transformations is the normalisation process to convert an arbitrary FOL formula to an equivalently satisfiable sentence expressable in CNF, as defined by Definition 1.4. The exact transformations are enumerated in Table A.1. In the implementation, a mutating visitor was constructed for each stage to recurse down the entire tree, capture relevant node classes, and apply the suitable normalisations. The pipeline for specifying and registering a FOL expression is as follows:

1. Flex tokenises an input string from the operator and Bison executes a shift-reduce parsing algorithm on the tokenised stream and incrementally builds a mutable AST.

2. The mutable AST is transformed into CNF by executing the mutating visitors enumerated by Table A.1.

3. The ownership of symbols detained by the CNF-normalised mutable AST are transferred into a global symbol repository.

4. An immutable AST is constructed by producing duplicate nodes containing weak references to symbols held by the repository.

Observing Visitors In addition to mutating visitors, observing visitors were implemented. Observing visitors cannot modify the object or change ownership, and were used extensively for *serialisation* of ASTs (expression in a human-readable form) in HTML, LaTeX, and plain text formats. An observing visitor was used to implement subsumption checks, described by Definition 1.5, to recursively search for similarities across ASTs. Observing visitors were also be used to implement comparators on ASTs.

3.3.4 Query-Driven Inference

Since the inception of the project, a major objective has involved the capability to perform computational inference on user-specified requirements. The processes introduced by Definition 1.2 and Definition 1.3 were implemented according to Algorithm A.1.

Users may, having populated a FOL KB organised into Analysis Groups, query the consistency of arbitrary FOL sentences within the requirements forming the group. Where a query is consistent, the software dynamically generates a Minimally Spanning Refutation Graph (MSRG)[4], designed to appear similar to the graphs in [27], which graphically indicates the clauses used to resolve a contradiction for the negation of the query. Figure 3.3 provides an MSRG for a *Modus Ponens* (MP) KB.

[4] MSRGs are visual deductive traces from KB axioms to a contradiction and include only the resolvents which contributed to the derivation of the $\bot$ clause.

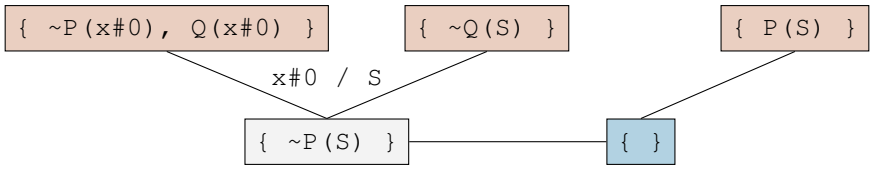


Figure 3.3: Generated MSRG of an MP argument.

Although the resolution is refutation-complete, its total runtime is dependent on the order in which clauses are supplied to the unifiers. Generated resolvents may also be repeated for equivalent pairs of clauses, which needlessly populate the working queue with redundant resolutions. This can be mitigated by ordering clauses according to *resolution potential* and simplifying the KB by eliminating clauses that are mutually θ-subsuming (Definition 1.5). Resolution potential is determined by a widely used combination of numerical features, restated by Definition 1.6, to order the clauses in a std::set encoded into a prefix tree as shown in Figure A.1. The standard algorithms to manipulate the FVI store were adapted from Schulz [31].

3.4 Test Integration and Automatic Reporting

To improve traceability throughout the SDLC, all requirements may be associated with an unlimited number of software unit tests specified by the operator. During analysis, tests associated with requirements are executed, and any failed tests are presented on the GUI.

A polymorphic object model was added to support discovery and execution of user-provided tests, in which a common interface is defined independent of any individual testing driver. When a test is queried or executed from within the application, a Transport Control Protocol (TCP) socket is opened to asynchronously receive results. In the operational prototype, a backend was defined for Google Test. As Google Test uses a custom protocol, another Flex/Bison interpreter was written for the streamed payload.

Furthermore, the capability to automatically generate human-readable reports for status-reporting and auditing compliance is commonplace in commercial engineering software. The infrastructure surrounding this functionality was designed with similar abstractions to the testing models: reports are composed through a format-independent interface to produce an Intermediate Representation (IR), and a serialisation driver converts the IR into a file. By the operational prototype, the product had stable serialisation drivers for LaTeX/PDF[5].

[5]The LaTeX is serialised into temporary text file, and the local *latexmk* [44] instance is invoked through an asynchronous subprocess interface to produce a PDF.

4 Testing and Evaluation

4.1 Software Validation

4.1.1 Performance and Correctness

To verify the correctness and performance of the implementation, a suite of unit tests were written in the Google Test unit-testing framework to target each component of the application subsystems. Tests were organised into a number of harnesses, arranged according to the feature-under-test.

Table A.2 describes a representative subset of the executed tests; in total, hundreds of timed unit tests were constructed to test all non-trivial aspects of the application, and code coverage of non-GUI routines was within the desired bounds of 80% to 90%. Fuzzing was used to randomise and stress selected test cases; introducing pseudo-randomness into the test harnesses rapidly accelerated the discovery of erroneous application behaviour. All tests passed at the point of submission. Due to time and practicality constraints, no automated testing was performed on the GUI aspects of the application, which was verified during spiral SDLC iterations with manual use.

In addition to unit and integration testing, extensive dynamic analysis was performed on the binaries under Linux hosts. The *Valgrind Memcheck* [45] and Google *Address Sanitizers* [46] were used to verify correctness of memory transactions and heap-allocation strategies (some of which were designed and implemented with C++17 polymorphic allocators to support pool-allocation of large node clusters). The *Valgrind Callgrind* [47] and *Cachegrind* [48] tools were used to optimise performance-critical sections.

The *log4cxx* C++ logging framework was used to provide highly configurable logging facilities [49]. Granular logs aided in debugging algorithms and complex interactions; Listing A.1 and Listing A.2 provide samples of logs generated from routine procedures.

4.1.2 Satisfaction against Requirements Criteria

Table 2.1 outlined numerous project requirements over five categories, each labelled with a rating of necessity (optional; desirable; essential) and anticipated risk profile (low; moderate; high). All desirable and essential

requirements were successfully met, and of the optional requirements, four were removed from the project scope:

- *Querying and executing J-Unit test suites (optional; moderate).* Integration with J-Unit testing was not achieved due to major differences with the protocol support added for Google Test. Although the unit-testing adapter was sufficiently flexible, the J-Unit infrastructure would have required a larger time commitment than was available given the remaining budget. For these reasons, its inclusion was deliberately removed from the project scope.

- *Inclusion of deduction traces in reports (optional; high).* Rendering full deduction traces in a variety of backend formats was an unattainable goal for inference procedures over any moderately sized KB. In addition to typesetting constraints, the internal drawing format used by the drawing toolkit did not cleanly map to a portable intermediate representation. For this reason, it was de-scoped.

- *Serialisation of reports into HTML (optional; low).* The existing reporting formats of LaTeX and plain text were deemed sufficient for most use cases given the universality of TeX driver formats PDF and DVI. Despite the low risk profile of HTML reporting, the time budget was allocated to hardening more essential components.

- *Unit testing for low-risk requirements (optional; low)*. Automated unit testing was not attempted or conducted for low-risk requirements. Time budgets were allocated to allow focus on rigorous testing of moderate- and high-risk requirements. The majority of low-risk requirements also related to GUI functionality which was not deemed suitable for validation through automated testing.

Centrally important GUI functions are demonstrated in Figures 4.1 to 4.4. In particular, Figure 4.1a shows the GUI elements for defining a requirement with associated metadata, FOL formulae, and an executable unit test. Figure 4.1b displays the integrated report generation capability in which a PDF file is produced from auto-generated LaTeX code.

Figure 4.2 includes the *Project Explorer* pane on the left-hand side and the *Requirements Index* view on the right-hand side. The former allows organisation of requirements into projects and nested subsystems, while the latter displays a tabulated view of specified requirements for the MP KB. Figure 4.3 displays the results of executed software tests for requirements aggregated in a named *Test Group*; failed tests are described with a tabulated view in the lower pane.

Figure 4.4 shows the *Analysis and Optimisation* view for a *Curiosity Killed*

the Cat (CKC) KB[1] organised into a single named Analysis Group. An MSRG is also dynamically generated and rendered to graphically demonstrate the validity of a user-specified query on the KB. For debugging and education purposes, the tabulated view also displays CNF sentences for each requirement.

4.2 Application to Case Studies

For a complete evaluation, the product was applied to three non-trivial case studies: a formally verifiable *Autonomous Chemical Detector* from the RoboStar research group [50]; an expository, synthesised study of a submarine sonar controller, and a *High-Voltage Controller for an Industrial Robotic System* [51]. System requirements from these studies were extracted and translated to the plain FOL domains supported by the product, enumerated by Tables C.1 to C.3 respectively. The requirements were then entered, with suitable metadata, into a subsystem KB within the product for analysis and reporting.

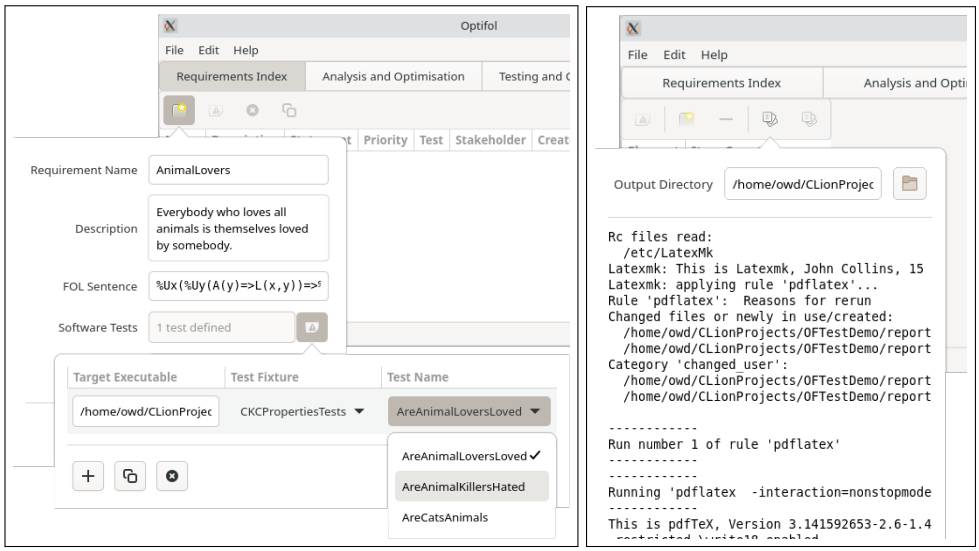
The software successfully verified the mutual consistency of the populated FOL KBs from each study, and aided synthesis and debugging of the sonar controller.

4.2.1 Autonomous Chemical Detector

The ACD describes an autonomous robotic chemical detector with behaviour organised around gas analysis, movement control, and overall coordination of the detector as a reactive system. Its requirements are expressed in terms of determinism, freedom from deadlocks, freedom from divergence, and required responses to environmental inputs such as gas readings. The requirements encapsulated by the ACD extend beyond simple static invariants by introducing behavioural requirements relating commands and reactions, making it a useful demonstration of abstracting event-driven system properties in untyped FOL.

The ACD also exposes numerous flaws of using an untyped FOL to express reactive systems involving timing. In the original literature, the ACD was largely described as a set of Constraint Satisfaction Problems (CSPs) to be checked dynamically with the *RoboTool* formal verification platform. The translation from CSP into plain FOL is a deliberate abstraction into functional requirements rather than an attempt to fully preserve semantics[2].

[1] CKC is a commonly used toy-example to demonstrate FOL inference procedures, such as by [27].



(a) Defining a requirement with tests. (b) Generating a LaTeX report.

Figure 4.1: Enterprise functionality including reporting and testing.

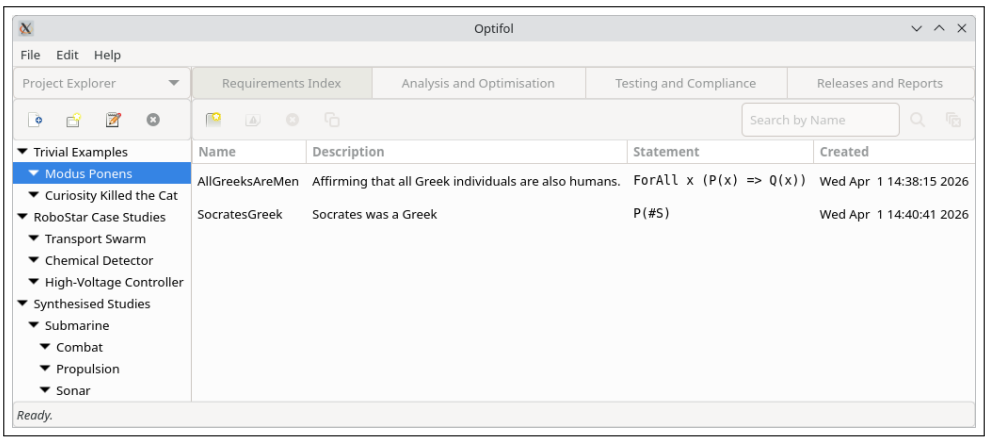


Figure 4.2: Entry of a MP basis into the software.

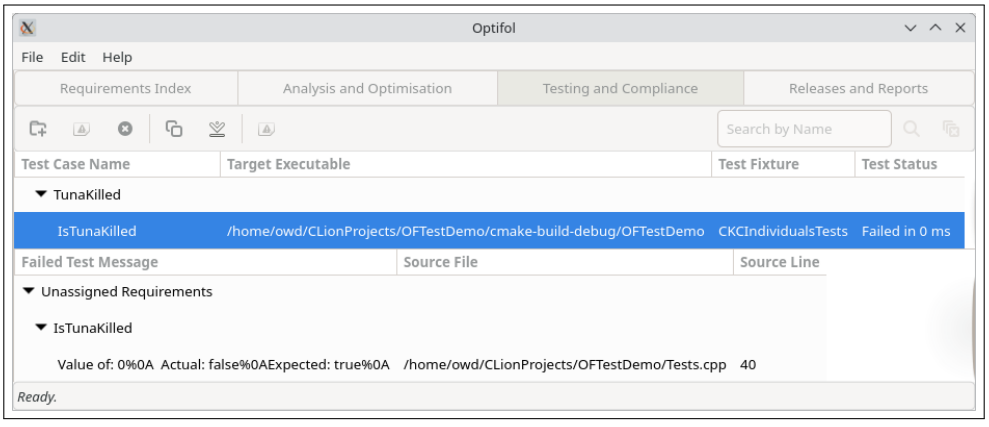


Figure 4.3: Reviewing failed unit tests in the *Testing and Compliance* view.

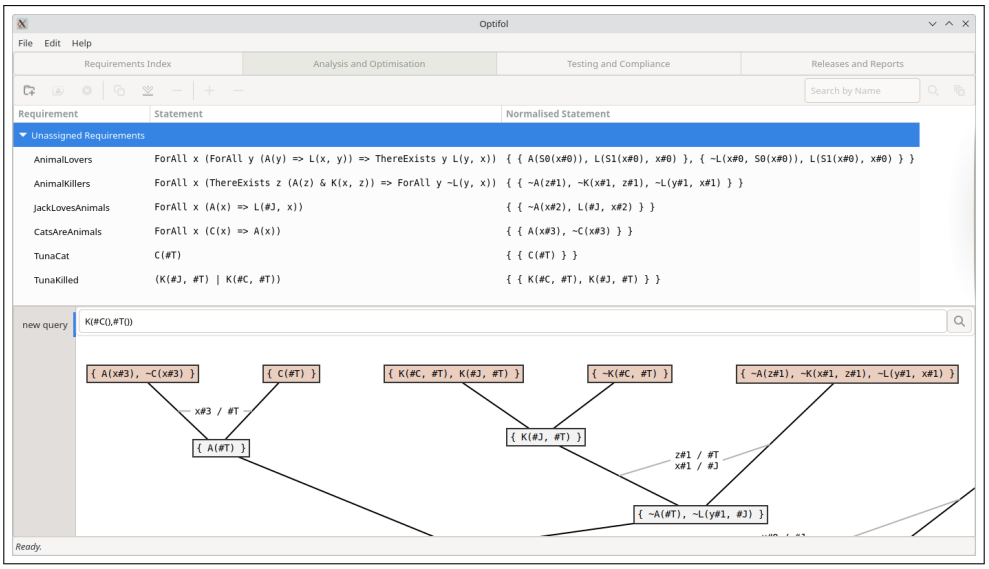


Figure 4.4: Automated deduction on CKC formulae.

The translation of the ACD requirements, encoded using introduced predicates GasAnalysis, Movement, and GasReading, were inserted into the system. The KB was found to be decidedly consistent, and a PDF report was generated to display the FOL constructions of requirements.

4.2.2 Submarine Sonar Controller

This synthetic case study models a high-level SSC subsystem. The system reasons about detected contacts, classifications such as hostile, friendly, or unknown, alert generation, investigation recommendations, and strict constraints on engagement permission. Its relevance lies in providing a realistic safety-critical domain in which the requirements are deliberately structured to suit plain FOL: the study combines consistency constraints, implication-based operational rules, and safety guards, allowing the toolkit to demonstrate contradiction detection and redundancy detection.

The system was used for real-time verification during the development of the study. Whilst the final set of requirements are decidedly consistent, the

[2] As with other cross-formalism encodings, a direct translation can be logically correct yet still lose important structure from the source language. In particular, CSP notions such as trace behaviour, synchronisation, deadlock semantics, timing, and typed events are not native to the chosen FOL structure, so they must be represented indirectly through auxiliary predicates; this makes the encoding less faithful and may weaken the correspondence between reasoning results in the target logic and the original process model. Further discussion of encoding quantified CSPs in alternative paradigms can be found in [52].

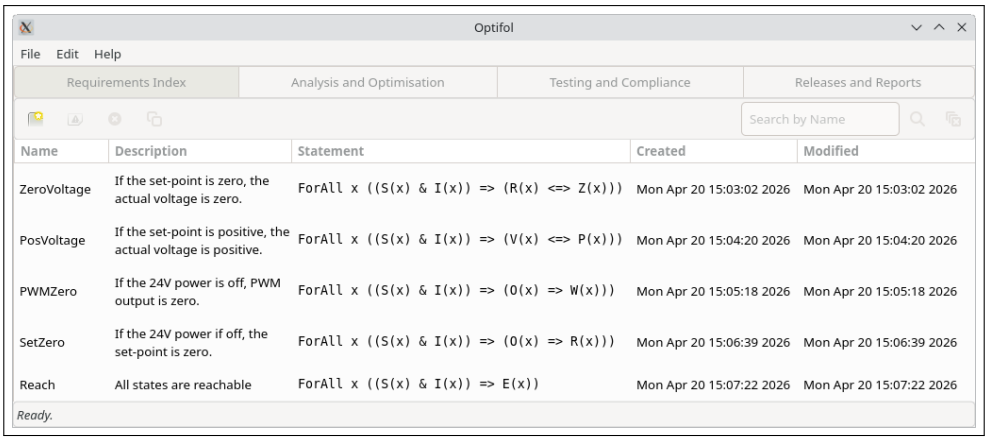


Figure 4.5: Entry of the HVC case study into the software.

system correctly identified and eliminated a number of subtly problematic axioms from preliminary versions of the study. For example, it was considered that every sonar contact under investigation should be permitted for engagement, canonicalised as

$$\forall x(\,\mathrm{Contact}(x) \implies (\mathrm{Investigate}(x) \implies \mathrm{EngagementPermitted}(x))). \tag{4.1}$$

The system correctly identified that introduction of Equation (4.1) would create an inconsistency[3]with Requirements B.4 and C.5 from Table C.2.

4.2.3 High-Voltage Controller

This case study is drawn from a HVC used in an industrial paint robot, in which the controller regulates electrostatic painting voltage. Strict safety requirements are set as incorrect behaviour can create dangerous discharge and ignition risks. The verified properties include ensuring that the actual high voltage follows the set-point, that PWM output is disabled when the input source is lost, that the set-point is reset when power is off, and that the controller state machine is deadlock free with all states reachable. It is especially relevant because it represents an industrial safety-critical application whose requirements are naturally close to logical invariants, making it an excellent baseline example for expressing and reasoning about requirements in FOL.

[3]The resolution-refutation algorithm (Algorithm A.1) noted that existing Requirement B.4 enforces $\mathrm{Investigate}$ on any contact x for which $\mathrm{Unknown}(x) \wedge \mathrm{EZone}(x)$, and that Requirement C.5 restricts any unknown-classified contact y such that $\mathrm{Unknown}(y) \implies \neg\,\mathrm{EngagementPermitted}(y)$. Hence, introduction of Equation (4.1) would apply both $\mathrm{EngagementPermitted}(z)$ and $\neg\,\mathrm{EngagementPermitted}(z)$ on any contact z which is both unknown-classified and within the submarine exclusion zone. An MSRG, similar in style to Figure 3.3, was drawn to isolate the involved axioms.

KB	Clauses	Resolv.	Dupl.	Pruned	Steps	Refuted?
MP	3	4	1	0	4	Yes
CKC	10	22	7	3	18	Yes
ACD	28	64	21	8	39	No
SSC†	48	131	54	17	63	Yes
SSC	47	119	46	19	88	No
HVC	21	51	17	8	34	No

Table 4.1: ATP performance metrics on known KBs.

The software did not identify any inconsistencies in the FOL KB given by Table C.3, which further demonstrates the wide-ranging applicability of the resolution-refutation algorithm. Figure 4.5 shows the entry of the HVC FOL KB into the software according to the expressions enumerated in Table C.3.

4.3 Overall ATP Performance

Quantitative performance metrics were measured from each stage of the ATP consistency-checking process when applied to the MP and CKC examples, and the three ACD, SSC, and HVC case studies.

Table 4.1 gives the results. Included for each KB are the number of initial CNF-normalised clauses, the initial number of resolvents generated, the number of duplicate clauses pruned, the number of clauses pruned due to being identified as subsumed by other clauses in the KB, the total number of iterations prior to termination, and the overall consistency outcome. The SSC† entry represents the SSC study with Equation (4.1) inserted.

The *Pruned* metric measures the total number of clauses removed from the KB due to subsumption, as described in Definition 1.5. These results show that prover effort increases broadly with the complexity of the KBs: the smallest terminate after only 4 and 18 steps, whereas the larger studies require between 34 and 88 steps. The subsumption metrics also indicate that a substantial amount of redundant work is removed prior to propagation through the resolution queue, particularly on the larger workloads. This is most visible in the SSC† case, refuted after 63 steps, while the final consistent model requires 88 steps before queue exhaustion, suggesting that contradictions may often be exposed earlier than full consistency can be established. Taken together, these results support the claim that the prototype is capable of analysing non-trivial requirement KBs, and that the FVI-backed pruning strategy makes a measurable contribution to tractability. They should, however, be interpreted as evidence of practical feasibility on representative case studies rather than proof of industrial-scale scalability.

4.4 Conclusions and Future Work

The final software artefact was moderately sized, measuring between 30,000 and 35,000 LOC, excluding generated Flex and Bison translation units. It served to provide insight into the Research Question which queried the usefulness and applicability of integrating existing technologies with academic ATPs. Although testing was only performed on modest case studies, the software has demonstrated that business-standard capabilities can be effectively combined with a platform to provide FM assurance in RE and trace the verification through the latter stages of SDLCs without sacrificing usability.

The broader software engineering industry, particularly parties involved in the development, integration, and test of products likely to benefit from FM application in the early SDLC stages, can examine the work described herein to consider the use of tighter integration of ATP toolkits within their existing development workflows. In particular, the development of *Optifol* has demonstrated that application of FM in SDLCs needn't be an artefact sitting adjacent to existing practices, but form a core, traceable, and reproducible stage in software engineering.

Research Question Answer. ATP techniques can be adapted, with relative ease, into featureful front-end applications combined with integrations into existing software infrastructure. With careful software architecture design, standard capabilities such as test-association and report-generation, may be configured to integrate with a wide variety of backend platforms already established in company development processes.

ATP is a rapidly progressing field with few defined borders. In the future, the prover may be improved to capture more detailed semantic properties in CSPs or higher-order logics to improve the expressiveness of representations of reactive systems in the industrial SDLC workflow. Various improvements to the logical model could be made, including further research on use of heuristics in ATP optimisation, addition of useful built-in predicates such as arithmetic operators, and stress-testing on larger KBs.

The *Optifol* application could be enhanced to include de-serialisation (saving and loading) of KBs and support a wider range of backend test and reporting frameworks. Native integration with a Product Lifecycle Management (PLM) system would also be advantageous for industrial application, as this would increase traceability, while managing coupling complexity, between RE and other phases of the SDLC.

A ATP Implementation Details

A.1 Further Details on CNF

The translation of an arbitrary FOL expression to an equivalently satisfiable CNF representation is a seven-stage process, within which each transformation is executed recursively over the tree. Table A.1 enumerates the stages, the match patterns, and the rewrite rule for captured nodes.

Trivial transformations, such as $\neg\neg P \mapsto P$, are handled during the initial parse. Introduction of Skolem functions $\mathcal{S}_0, \mathcal{S}_1, \ldots$ preserves satisfiability, but not equivalence, of the original sentence [27]. Universal elimination relies on implicit quantification of non-free variables.

It was remarked in the implementation notes that on extremely deep ASTs, the recursive implementation strategy used by the visitor pattern may cause a machine stack overflow; to mitigate this risk, the recursive algorithms were designed to use tail-recursion where practicable. This allows the compiler to perform *tail-call optimisation* and collapse the call stack before the innermost function fully returns.

A.2 Clause Storage and Resolution

The algorithms and data structures used to store, retrieve, and manage CNF clauses are performance critical. As the clause structures are used extensively by the queue-based resolution ATP algorithm, described by Algorithm A.1, the storage strategies were designed to minimise the average runtime of typical deductions. Clauses are assigned a Resolution Potential Rating (RPR) according to four heuristically derived features:

1. Maximum nesting depth;
2. Number of literals under disjunction;
3. Number of unique functions; and
4. Number of unique variables.

Maximising the RPR over the selected features is experimentally observed to minimise the average number of steps required during resolution[1]; this is an equivalent notion to preventing generation of new, unseen resolvents

Stage	Target	Rewrite Rule
Remove implications	$P \implies Q$	$\neg P \vee Q$
	$P \iff Q$	$(P \vee \neg Q) \wedge (\neg P \vee Q)$
Apply DML	$\neg(P \vee Q)$	$\neg P \wedge \neg Q$
	$\neg(P \wedge Q)$	$\neg P \vee \neg Q$
	$\neg \forall x P(x)$	$\exists x \neg P(x)$
	$\neg \exists x P(x)$	$\forall x \neg P(x)$
Standardise variables	$(\ldots x \ldots) \cdot (\ldots x \ldots)$	$(\ldots x_0 \ldots) \cdot (\ldots x_1 \ldots)$
Extract quantifiers	$P \wedge \forall x Q(x)$	$\forall x \, (P \wedge Q(x))$
	$P \vee \forall x Q(x)$	$\forall x \, (P \vee Q(x))$
	$P \wedge \exists x Q(x)$	$\exists x \, (P \wedge Q(x))$
	$P \vee \exists x Q(x)$	$\exists x \, (P \vee Q(x))$
Introduce Skolems	$\forall x_1 \ldots \forall x_n \exists y P(y)$	$\forall x_1 \ldots \forall x_n$ $P\left(\mathcal{S}\left(x_1, \ldots, x_n\right)\right)$
Eliminate universals	$\forall x P(x)$	$P(x)$
Distribute disjuncts	$P \vee (Q \wedge R)$	$(P \vee Q) \wedge (P \vee Q)$

Table A.1: The seven-stage CNF normalisation pipeline.

which prevent the queue from collapsing. The features are used to organise clauses in a prefix tree as shown by Figure A.1.

θ-subsumption checks are also performed prior to the clause store accepting new entries, such that the structure is self-deduplicating for logically equivalent formulae as well as superficial copies.

A.3 Logging Samples

Listing A.1 and Listing A.2 demonstrate the logging capability of the application for the specification, normalisation, and deduction of FOL expressions, and KBs thereof. C++20 std::format was also used with the AST serialisers to provide descriptions of the system state throughout during operations.

In Listing A.1, the axiomatic basis for a MP argument is being entered into the system as a couple of requirements. Transparently to the user, the normalisation pipeline transforms both expressions into CNF-compliant clause sets and inserts them into the FVI data structure used to maintain the active Analysis Group for the Subsystem. In Listing A.2, the complete KB is queried on a simple property.

[1] [53] provides a C#/.NET implementation and relevant benchmarking of some commonly used features in FOL deduction routines.

Input : Clause set KB
Input : Query sentence Q
Input : Maximum number of resolution steps N
Output : ENTAILED, NOTENTAILED, or TIMEDOUT

Function ResolutionRefutation(KB, Q, N)
    $Derived \leftarrow \text{CNF}(\neg Q)$;
    $Base \leftarrow KB \cup Derived$;
    $Queue \leftarrow \emptyset$;
    $Seen \leftarrow \emptyset$;
    $i \leftarrow 0$;
    foreach $C_1 \in Base$ **do**
        foreach $C_2 \in Base$ **do**
            foreach $R \in \text{Resolve}(C_1, C_2)$ **do**
                $Queue \leftarrow Queue \cup \{R\}$;
            end
        end
    end
    while $Queue \neq \emptyset$ **do**
        $R \leftarrow \text{Pop}(Queue)$;
        $i \leftarrow i + 1$;
        if $R \in Seen$ **then**
            continue;
        end
        $Seen \leftarrow Seen \cup \{R\}$;
        if $R = \bot$ **then**
            return ENTAILED;
        end
        if $R \notin Derived$ **then**
            $Derived \leftarrow Derived \cup \{R\}$;
            foreach $C \in (KB \cup Derived)$ **do**
                foreach $R' \in \text{Resolve}(R, C)$ **do**
                    if $R' \notin Seen$ **then**
                        $Queue \leftarrow Queue \cup \{R'\}$;
                    end
                end
            end
        end
        if $i \geq N$ **then**
            return TIMEDOUT;
        end
    end
    return NOTENTAILED;

Algorithm A.1: Core binary resolution refutation procedure.

```
[INFO] [LogicServices.CNFNormalisation] Beginning CNF pipeline transformation.
[INFO] [LogicServices.CNFNormalisation] Initial sentence: ForAll x (P(x) => Q(x))
[DEBUG] [LogicServices.CNFNormalisation.ImplicationElimination] ForAll x (~P(x) | Q(x))
[DEBUG] [LogicServices.CNFNormalisation.DeMorgan] ForAll x (~P(x) | Q(x))
[DEBUG] [LogicServices.CNFNormalisation.SymbolStandardiser] ForAll x (~P(x) | Q(x))
[DEBUG] [LogicServices.CNFNormalisation.QuantifierExtraction] ForAll x (~P(x) | Q(x))
[DEBUG] [LogicServices.CNFNormalisation.SkolemIntroduction] ForAll x (~P(x) | Q(x))
[DEBUG] [LogicServices.CNFNormalisation.UniversalElimination] (~P(x) | Q(x))
[DEBUG] [LogicServices.CNFNormalisation.DisjunctionDistribution] (~P(x) | Q(x))
[INFO] [LogicServices.CNFNormalisation] Completed CNF transformation.
[INFO] [LogicServices.CNFNormalisation] Normalised sentence: (~P(x) | Q(x))
[DEBUG] [GUI.StorageControl.Requirement] Updated FOL statement for requirement "AllGreeksAreMen".
[DEBUG] [GUI.StorageControl.Subsystem] Handling requirements change: 1 additions and 0 deletions.
[DEBUG] [GUI.StorageControl.Subsystem] Propagating addition of "AllGreeksAreMen" to default Analysis Group.
[INFO] [LogicServices.FVIKnowledgeBase] Accepting clause { ~P(x#0), Q(x#0) } into the KB.
[DEBUG] [LogicServices.FVIKnowledgeBase] The KB now contains 1 clauses.
[INFO] [GUI.RequirementsIndex.NewRequirement] Created new Subsystem Requirement with name "AllGreeksAreMen".
```

Listing A.1: Log capturing insertion and normalisation of an MP basis requirement.

```
[INFO] [LogicServices.CNFNormalisation] Normalised sentence: ~Q(#S)
[INFO] [LogicServices.Prover] Querying the KB of 2 clauses for the negation of { { ~Q(#S) } }.
[TRACE] [LogicServices.Prover] Dumping initial knowledge base...
[TRACE] [LogicServices.Prover] KB Clause 1/2: { ~P(x#0), Q(x#0) }
[TRACE] [LogicServices.Prover] KB Clause 2/2: { P(#S) }
[TRACE] [LogicServices.Prover] KB Clause (negated goal): { ~Q(#S) }
[INFO] [LogicServices.FVIKnowledgeBase] Accepting clause { ~Q(#S) } into the KB.
[DEBUG] [LogicServices.Prover.Resolution] Constructed a unified clause of 1 literals.
[DEBUG] [LogicServices.Prover.Resolution] Resolved { ~P(x#0), Q(x#0) } and { P(#S) } to { Q(#S) }.
[DEBUG] [LogicServices.Prover.Resolution] Constructed a unified clause of 1 literals.
[DEBUG] [LogicServices.Prover.Resolution] Resolved { ~P(x#0), Q(x#0) } and { ~Q(#S) } to { ~P(#S) }.
[DEBUG] [LogicServices.Prover.Resolution] Constructed a unified clause of 1 literals.
[DEBUG] [LogicServices.Prover.Resolution] Resolved { P(#S) } and { ~P(x#0), Q(x#0) } to { Q(#S) }.
[DEBUG] [LogicServices.Prover.Resolution] Constructed a unified clause of 1 literals.
[DEBUG] [LogicServices.Prover.Resolution] Resolved { ~Q(#S) } and { ~P(x#0), Q(x#0) } to { ~P(#S) }.
[DEBUG] [LogicServices.Prover.Resolution] Resolution initial search gathered 4 resolvents.
[INFO] [LogicServices.Prover.Resolution] Step 1 is using resolution { Q(#S) }.
[INFO] [LogicServices.FVIKnowledgeBase] Accepting clause { Q(#S) } into the KB.
[DEBUG] [LogicServices.Prover.Resolution] Resolution skipping step 2 due to repeated resolvent.
[INFO] [LogicServices.Prover.Resolution] Step 3 is using resolution { ~P(#S) }.
[INFO] [LogicServices.FVIKnowledgeBase] Accepting clause { ~P(#S) } into the KB.
[DEBUG] [LogicServices.Prover.Resolution] Constructed a unified clause of 0 literals.
[DEBUG] [LogicServices.Prover.Resolution] Resolved { ~P(#S) } and { P(#S) } to {  }.
[DEBUG] [LogicServices.Prover.Resolution] Resolution skipping step 4 due to repeated resolvent.
[INFO] [LogicServices.Prover.Resolution] Step 5 is using resolution {  }.
[INFO] [LogicServices.FVIKnowledgeBase] Accepting clause {  } into the KB.
[INFO] [LogicServices.Prover] A contradiction was derived in 5 steps; the queried sentence is consistent.
```

Listing A.2: Log capturing a refutation-resolution procedure on the MP KB.

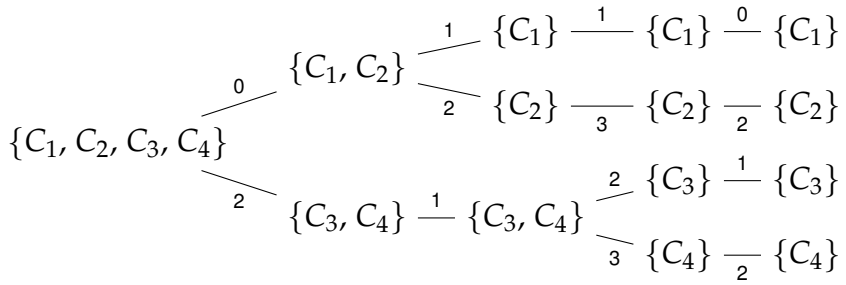


Figure A.1: Sample prefix tree layout for a four-feature FVI clause store containing clauses C_1, C_2, C_3, C_4.

A.4 Unit Tests

Table A.2 enumerates a representative subset of the unit tests defined and executed for the ATP and ATP-adjacent components of the application. For each test, organised by harness, an *Action* and *Expected Result* is specified according to the constructed state and expected resultant state following the operation-under-test.

The symbols detained by FOL expressions are typed implicitly according to the FOL grammar: see Grammar A.1 for a full description.

A.5 FOL Expression Grammar

Grammar A.1 describes a simplified Backus-Naur Form (BNF) representation of the FOL syntax and grammar implemented by the Flex lexer and Bison parsers, respectively. Non-terminal symbols are represented as regular expressions, and the empty symbol is notated by λ. All binary connectives are left-associative in the canonical precedence order, and the unary negation operator is right-associative.

In the final implementation, mathematical symbols are typed as representative ASCII character sequences, such as "=>" in-place of " $\Longrightarrow$ ". Additional implementation details include error-handling on erroneous sequences and compatibility with cross-platform line endings.

Listing A.3 displays a Bison/C++ fragment to generate a mutable AST node, according to the class model pictured in Figure 3.1, from a universally quantified variable bound over a sentence. Referenced symbols, such as `Universal`, `Variable`, and `Predicate`, are provided as a tokenised symbol stream from the Flex lexer runtime.

Test Harness / Test Name	**Action**	**Expected Result**
Parsing and AST Construction	*Input String*	*AST Construction*
`Quantifier_Universal`	`%Ux(P(x))`	$\forall x P(x)$
`Quantifier_Existential`	`%Ey(~Q(y))`	$\exists y \neg Q(y)$
`Quantifier_Nested`	`%Ux(%Ey(P(x) & Q(y)))`	$\forall x \exists y \left(P(x) \wedge Q(y)\right)$
`TermBuilder_NoArguments`	`%Ux(P())`	$\forall x P$
CNF Normalisation	*Input Expression*	*Output Expression*
`ImpElim_Basic`	$P \implies Q$	$\neg P \vee Q$
`ImpElim_Biconditional`	$P \iff Q$	$(P \vee \neg Q) \wedge (\neg P \vee Q)$
`ImpElim_UnaryNesting`	$(P \implies Q) \iff R$	$(\neg (\neg P \vee Q) \vee R) \wedge ((\neg P \vee Q) \vee \neg R)$
`ImpElim_BinaryNesting`	$(P \implies Q) \iff (R \implies S)$	$(\neg (\neg P \vee Q) \vee (\neg R \vee S)) \wedge ((\neg P \vee Q) \vee \neg (\neg R \vee S))$
`DisjDist_BasicUnary`	$P \vee (Q \wedge R)$	$(P \vee Q) \wedge (P \vee R)$
`DisjDist_BasicBinary`	$(P \vee (Q \wedge R)) \wedge ((A \wedge B) \vee C)$	$A \wedge (P \vee (Q \wedge R))$
`DisjDist_NestedNoOp`	$A \wedge (P \vee (Q \wedge R))$	$A \wedge ((P \vee Q) \wedge (P \vee R))$
Bidirectional Unification	*Operands* (LHS; RHS)	*Substitution Set*
`Positive_Single_FunVar`	$P(C, x); P(C, D)$	$\{x \mapsto D\}$
`Positive_Multiple_FunVar`	$P(C, x); P(y, D)$	$\{x \mapsto D, y \mapsto C\}$
`Positive_Multiple_VarVar`	$P(a, b); P(c, d)$	$\{c \mapsto a, d \mapsto b\}$
`Negative_Multiple_VarVar`	$P(C, x); P(x, D)$	$\emptyset$
Unidirectional Unification	*Operands* (General; Instance)	*Substitution Set*
`Positive_Single_FunVar`	$P(C, x); P(C, D)$	$\{x \mapsto D\}$

Continued on the next page.

`Negative_Single_VarFun`	$P(C, x); P(C, D)$	$\emptyset$
Knowledge Base Construction	*Input Expression*	*Knowledge Base*
`Clause_Predicate`	$P(x, y, z)$	$\{\{P(x_0, y_0, z_0)\}\}$
`Clause_Predicates`	$P \vee Q$	$\{\{P, Q\}\}$
`Clauses_Predicate`	$P \wedge Q$	$\{\{P\}, \{Q\}\}$
`Clauses_Predicates`	$(P \vee Q) \wedge (R \vee S)$	$\{\{P, Q\}, \{R, S\}\}$
Resolution Correctness	*Knowledge Base*	*Intended Deduction (Consistent?)*
`ModusPonens`	$\{\forall x (P(x) \implies Q(x)), P(C)\}$	$Q(C)$ (consistent)
`Reject_Trivial`	$\{P(C), Q\}$	$\neg R(C)$ (inconsistent)
`CuriosityKilledTheCat`	Omitted for brevity	Kills (Curiosity, TheCat) (consistent)
Feature Vector Index Construction	*Inserted Clauses*	*Accepted Clauses*
`Positive_Trivial`	$\{P, Q, R\}$	$\{P, Q, R\}$
`Ignore_Trivial`	$\{P, Q, P\}$	$\{P, Q\}$
Clause Construction	*Inserted Literals*	*Constructed Clause Metadata*
`TriviallyTrue_Simple`	$\{P, \neg P\}$	Trivially true; zero literals inserted
`TriviallyFalse_Simple`	$\emptyset$	Trivially false; zero literals inserted
`NonTrivial_Simple`	$\{P, Q\}$	Non-trivial; $\{P, Q\}$ inserted

Table A.2: An abridged index of selected unit tests.

$$
\begin{aligned}
\langle \text{sentence} \rangle &\models \langle \text{quantifier} \rangle \langle \text{variable} \rangle \langle \text{sentence} \rangle \\
&\mid \langle \text{predicate} \rangle \, (\langle \text{term\_vector} \rangle) \\
&\mid \neg \langle \text{sentence} \rangle \\
&\mid \langle \text{sentence} \rangle \langle \text{connective} \rangle \langle \text{sentence} \rangle \\
&\mid \, (\langle \text{sentence} \rangle) \\
\\
\langle \text{quantifier} \rangle &\models \forall \mid \exists \\
\\
\langle \text{variable} \rangle &\models \texttt{[a-z]+} \\
\\
\langle \text{predicate} \rangle &\models \texttt{[A-Z]+} \\
\\
\langle \text{term\_vector} \rangle &\models \langle \text{term} \rangle \\
&\mid \langle \text{term\_vector} \rangle, \langle \text{term} \rangle \\
&\mid \lambda \\
\\
\langle \text{connective} \rangle &\models \wedge \mid \vee \mid \implies \mid \impliedby \mid \iff \\
\\
\langle \text{term} \rangle &\models \langle \text{function} \rangle \\
&\mid \langle \text{function} \rangle \, (\langle \text{term\_vector} \rangle) \\
&\mid \langle \text{variable} \rangle \\
\\
\langle \text{function} \rangle &\models \texttt{\#[a-zA-]+}
\end{aligned}
$$

Grammar A.1: FOL Grammar Specification

```
sentence :
    Universal Variable LeftParen sentence RightParen
    {
      $$ = new MutableQuantified(
        QuantifierTypes::Universal,
        std::unique_ptr<MutableVariable>($2),
        std::unique_ptr<IMutableSentence>($4)
      );
    }
    |
    Predicate LeftParen term_vector RightParen
    {
      $$ = new MutablePredicate($1, std::move($3));
    }
    ;
```

Listing A.3: Bison clause instruction building universally quantified sentences and predicates into the AST.

B Sample Report

Figure B.1 presents the Requirements Index from a generated report built on a CKC subsystem KB. The PDF report was produced from automatically serialised LaTeX, which supports correctly typeset tables and mathematical notation. Margin sizes have been altered for typographical reasons.

Although HTML and plain text backends were de-scoped from the project due to time constraints, as justified in Section 4.1.2, similar documents could be theoretically produced in a variety of formats.

In addition to the Requirements Index, full *Optifol* reports detain results of any queries executed on subsystem Analysis Groups and details of associated software tests in Test Groups. Latter sections have been omitted.

Project Subsystem Requirements Report

Optifol User

Generated on 2026-04-25 15:01:17+01:00

1 Index of Requirements

Name	Description	Statement
AnimalLovers	Anybody who loves all animals is themselves loved by somebody.	$\forall x \left(\forall y \left(A \left(y\right) \implies L \left(x, y\right)\right) \implies \exists y \left(L \left(y, x\right)\right)\right)$
AnimalKillers	Anybody who kills an animal is loved by nobody.	$\forall x \left(\exists z \left(A \left(z\right) \implies K \left(x, z\right)\right) \implies \forall y \left(L \left(y, x\right)\right)\right)$
JackLovesAnimals	Jack loves all animals	$\forall x \left(A \left(x\right) \implies L \left(\#J \left(\right), x\right)\right)$
CatsAreAnimals	All cats are animals	$\forall x \left(C \left(x\right) \implies A \left(x\right)\right)$
TunaCat	Tuna is a cat	$C \left(\#T \left(\right)\right)$
TunaKilled	Tuna was killed by Jack or Curiosity	$K \left(\#J \left(\right), \#T \left(\right)\right) \vee K \left(\#C \left(\right), \#T \left(\right)\right)$

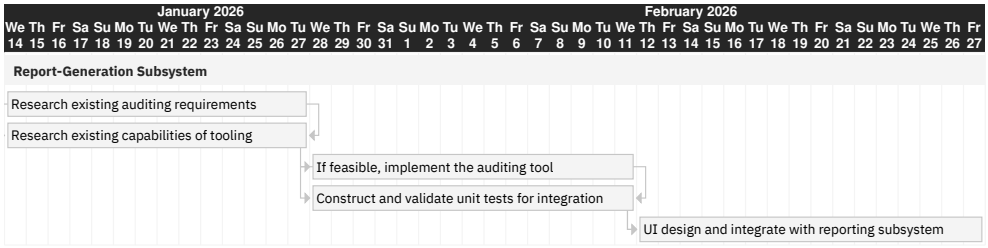
Page 1 Generated by Optifol User with Optifol In-Development

Figure B.1: A sample Requirements Index from a CKC subsystem KB.

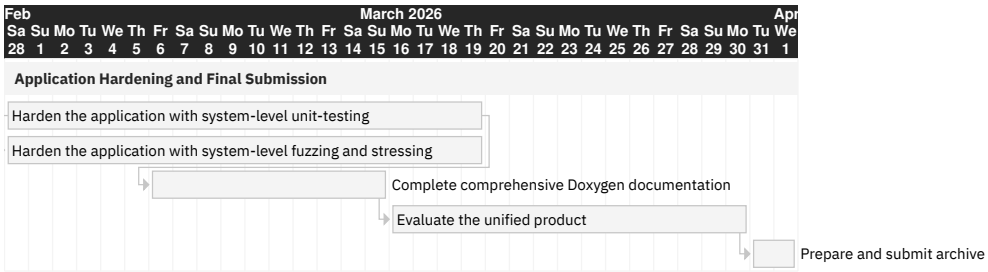
C Notes on Planning and Testing

C.1 Latter Planning Stages and the Spiral SDLC

Figure C.1 displays the Gantt chart for the development of the report-generation and final integration testing. Figure C.2 displays a typical single-iteration spiral model taken from [38] and included here for completeness.



(a) Development plan for report-generation.



(b) The latter project stages, reserved for testing and hardening.

Figure C.1: Gantt charts covering report-generation and final testing.

C.2 Case Study FOL Interpretations

Tables C.1 to C.3 enumerate the translated FOL requirements for the ACD, SSC, and HVC respectively. Requirement numbers (marked with **#**) in Table C.1 and Table C.3 reflect indices provided in the original literature, and Table C.2 requirements are delimited by subsystem type.

In all cases, predicates and functions are introduced implicitly to model the CSP-style constraints from the respective scenarios.

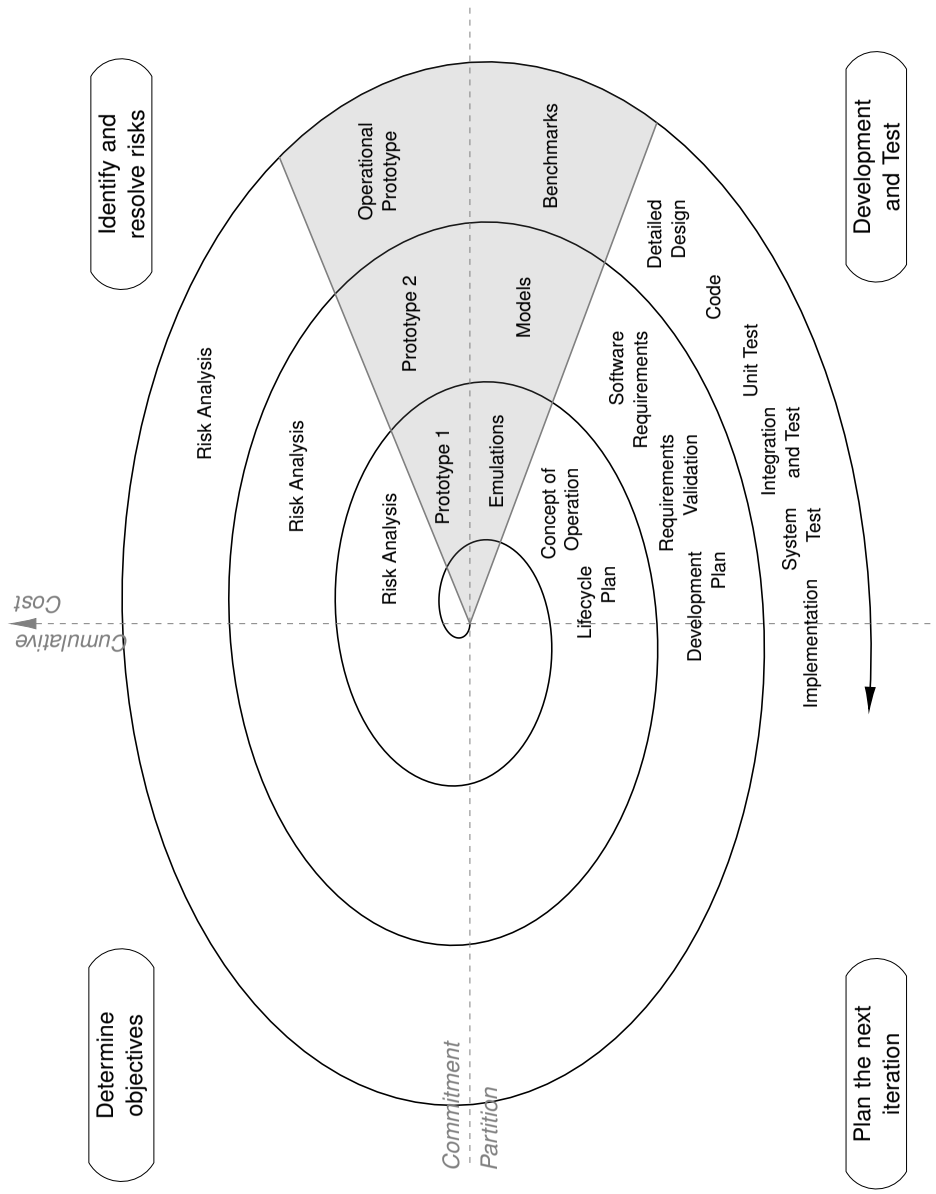


Figure C.2: The Spiral software development method.

#	Requirement	FOL Expression
A.1	GasAnalysis is divergence-free	$\forall x \exists t\, \text{HaltsAfter}(\text{GasAnalysis}(x), t)$
A.2	GasAnalysis deadlocks only after a `stop`	$\forall x (\text{Deadlock}(\text{GasAnalysis}(x)) \implies \text{Stop}(x))$
A.3	Movement is divergence-free	$\forall x \exists t\, \text{HaltsAfter}(\text{Movement}(x), t)$
A.4	Movement deadlocks only after a `flag`	$\forall x (\text{Deadlock}(\text{Movement}(x)) \implies \text{Flag}(x))$
A.5	ChemicalDetector deadlocks only after termination	$\forall x (\text{Deadlock}(\text{ChemDetector}(x)) \implies \text{Terminate}(x))$
B.1	Every gas reading leads to a command	$\forall x (\text{GasReading}(x) \implies \exists y (\text{Resume}(y) \lor \text{Stop}(y) \lor \text{Turn}(y)))$
B.2	Every command causes a reaction	$\forall x (\text{MoveCommand}(x) \implies \exists y (\text{Reaction}(y) \land \text{BeforeNext}(y)))$
B.3	No gas, no termination	$\forall x (\neg\, \text{GasPresent}(x) \implies \neg\, \text{Terminate}(x))$

Table C.1: FOL interpretation of the ACD requirements.

#	Requirement	FOL Expression
A.1	Every hostile contact is classified	$\forall x (\text{Contact}(x) \implies (\text{Hostile}(x) \implies \text{Classified}(x)))$
A.2	Every friendly contact is classified	$\forall x (\text{Contact}(x) \implies (\text{Friendly}(x) \implies \text{Classified}(x)))$
A.3	Every unknown contact is classified	$\forall x (\text{Contact}(x) \implies (\text{Unknown}(x) \implies \text{Classified}(x)))$
A.4	No contact is both hostile and friendly	$\forall x (\text{Contact}(x) \implies \neg(\text{Hostile}(x) \land \text{Friendly}(x)))$
A.5	No contact is both hostile and unknown	$\forall x (\text{Contact}(x) \implies \neg(\text{Hostile}(x) \land \text{Unknown}(x)))$
A.6	No contact is both friendly and unknown	$\forall x (\text{Contact}(x) \implies \neg(\text{Friendly}(x) \land \text{Unknown}(x)))$
A.7	No contact is both high-confidence and low-confidence	$\forall x (\text{Contact}(x) \implies \neg(\text{HighConfidence}(x) \land \text{LowConfidence}(x)))$

Continued on the next page.

B.1	Every detected contact is tracked or classified	$\forall x(\text{Contact}(x) \implies (\text{Detected}(x) \implies \text{Tracked}(x)))$
B.2	Every hostile contact raises an alert	$\forall x(\text{Contact}(x) \implies \implies (\text{Hostile}(x) \implies \text{Alerted}(x)))$
B.3	Every non-friendly contact in the exclusion zone raises an alert	$\forall x(\text{Contact}(x) \implies ((\text{EZone}(x) \wedge \neg\,\text{Friendly}(x)) \implies \text{Alerted}(x)))$
B.4	Every unknown contact in the exclusion zone is recommended for investigation	$\forall x(\text{Contact}(x) \implies ((\text{Unknown}(x) \wedge \text{EZone}(x)) \implies \text{Investigate}(x)))$
B.5	Every biologic contact is non-hostile	$\forall x(\text{Contact}(x) \implies (\text{Biologic}(x) \implies \neg\,\text{Hostile}(x)))$
B.6	Every biologic contact is not engagement-permitted	$\forall x(\text{Contact}(x) \implies (\text{Biologic}(x) \implies \neg\,\text{EngagementPermitted}(x)))$
C.1	Engagement permission requires hostile classification	$\forall x(\text{Contact}(x) \implies (\text{EngagementPermitted}(x) \implies \text{Hostile}(x)))$
C.2	Engagement permission requires operator confirmation	$\forall x(\text{Contact}(x) \implies (\text{EngagementPermitted}(x) \implies \text{OperatorConfirmed}(x)))$
C.3	Engagement permission requires high confidence	$\forall x(\text{Contact}(x) \implies (\text{EngagementPermitted}(x) \implies \text{HighConfidence}(x)))$
C.4	Friendly contacts are never engagement-permitted	$\forall x(\text{Contact}(x) \implies (\text{Friendly}(x) \implies \neg\,\text{EngagementPermitted}(x)))$
C.5	Unknown contacts are never engagement-permitted	$\forall x(\text{Contact}(x) \implies (\text{Unknown}(x) \implies \neg\,\text{EngagementPermitted}(x)))$

Table C.2: FOL interpretation of the synthesised SSC case study requirements.

#	Requirement	FOL Expression
P.1a	If the set-point is zero, the actual voltage is zero.	$\forall x((\text{State}(x) \wedge \text{In}(x, \text{Machine})) \implies (\text{SetZero}(x) \iff \text{ActualZero}(x)))$
P.1b	If the set-point is positive, the actual voltage is positive.	$\forall x((\text{State}(x) \wedge \text{In}(x, \text{Machine})) \implies (\text{SetPositive}(x) \iff \text{ActualPositive}(x)))$
P.2	If the 24V power is off, PWM output is zero	$\forall x((\text{State}(x) \wedge \text{In}(x, \text{Machine})) \implies (\text{PowerOff}(x) \implies \text{PWMOff}(x)))$
P.3	If the 24V power is off, the set-point is zero	$\forall x((\text{State}(x) \wedge \text{In}(x, \text{Machine})) \implies (\text{PowerOff}(x) \implies \text{SetZero}(x)))$
P.4	The machine is deadlock-free	$\text{DeadlockFree}(\text{Machine})$
P.5	All states are reachable	$\forall x((\text{State}(x) \wedge \text{In}(x, \text{Machine})) \implies \text{Reachable}(x))$

Table C.3: FOL interpretation of the HVC requirements.

Bibliography

[1] O. Dixon. '*Optifol* software source repository,' Accessed: 9th Apr. 2026. [Online]. Available: https://github.com/oliverdixon/OptiFOL-Software.

[2] H. D. Benington, 'Production of large computer programs,' in *Symposium on Advanced Programming Methods for Digital Computers*, (13th–14th Jun. 1956), M. E. Rose, Ed., Washington D.C., USA: Office of Naval Research, U.S. Department of the Navy, 1956, pp. 15–27.

[3] N. B. Ruparelia, 'Software development lifecycle models,' *ACM SIGSOFT Software Engineering Notes*, vol. 35, no. 3, pp. 8–13, May 2010, ISSN: 0163-5948. DOI: 10.1145/1764810.1764814.

[4] D. Cohen, M. Lindvall and P. Costa, 'An introduction to agile methods,' *Advances in computers*, vol. 62, no. 3, 2004, ISSN: 1945-3116.

[5] P. A. Laplante and M. Kassab, *Requirements Engineering for Software and Systems*, 4th ed. New York, NY, USA: Auerbach Publications, 2022, ISBN: 9781003129509.

[6] H. A. Partsch, *Specification and Transformation of Programs: A Formal Approach to Software Development*, 4th ed. Springer Science & Business Media, 2012, pp. 19–20.

[7] J. Schwartz, 'Construction of software: Problems and practicalities,' in *Practical Strategies for Developing Large Software Systems*, E. Horowitz, Ed. Reading, MA, USA: Addison-Wesley Publishing Company, Inc., 1975, pp. 57–71.

[8] J. Iqbal et al., 'Requirements engineering issues causing software development outsourcing failure,' *PloS One*, vol. 15, no. 4, 2020. DOI: 10.1371/journal.pone.0229785.

[9] E. Astesiano, G. Reggio and M. Cerioli, 'From formal techniques to well-founded software development methods,' in *Formal Methods at the Crossroads. From Panacea to Foundational Support: 10th Anniversary Colloquium of UNU/IIST, the International Institute for Software Technology of The United Nations University*, B. K. Aichernig and T. Maibaum, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 132–150, ISBN: 978-3-540-40007-3. DOI: 10.1007/978-3-540-40007-3_9.

[10] J. A. Van der Poll, 'Formal methods in software development: A road less travelled,' *South African Computer Journal*, vol. 2010, no. 45, pp. 40–52, 2010. DOI: 10.10520/EJC28102.

[11] P. Zave and M. Jackson, 'Four dark corners of requirements engineering,' *ACM Trans. Softw. Eng. Methodol.*, vol. 6, no. 1, pp. 1–30, Jan. 1997, ISSN: 1049-331X. DOI: 10.1145/237432.237434.

[12] MathWorks Ltd. 'Requirements Toolbox, Author, link, and validate requirements for designs and tests,' Accessed: 1st Dec. 2025. [Online]. Available: https://uk.mathworks.com/products/requirements-toolbox.html.

[13] IBM Inc. 'IBM engineering requirements management,' Accessed: 1st Dec. 2025. [Online]. Available: https://www.ibm.com/products/requirements-management.

[14] Drisq Ltd. 'Kapture®,' Accessed: 1st Dec. 2025. [Online]. Available: https://www.drisq.com/kapture-more-about-information.

[15] dSPACE GmbH. 'Synect, Data management and collaboration software for automated ecu testing,' Accessed: 1st Dec. 2025. [Online]. Available: https://www.dspace.com/en/ltd/home/products/sw/datenmanagement/synect.cfm.

[16] S. Burmester, K. Lamberg, C. Wewetzer and C. Thiessen, 'Method of creating a requirement description for testing an embedded system,' U.S. Patent 7970601B2, 28th Jun. 2011. [Online]. Available: https://patents.google.com/patent/US7970601B2.

[17] Drisq Ltd. 'Flight control system case study,' Accessed: 1st Dec. 2025. [Online]. Available: https://www.drisq.com/flight-control-system-case-study.

[18] 'Software considerations in airborne systems and equipment certification,' RTCA, Inc., Washington, D.C., RTCA Standard DO-178C, 2011.

[19] 'DO-333: Formal methods supplement to DO-178C and DO-278A,' RTCA, Inc., Washington, D.C., RTCA Supplement DO-333, 2011.

[20] M. G. Hinchey, J. Rash and R. C. A., 'Requirements to design to code: Towards a fully formal approach to automatic code generation,' Goddard Space Flight Center, National Aeronautics and Space Administration, Greenbelt, MD, USA, Technical Report 212774, 2005.

[21] G. Carvalho, A. Cavalcanti and A. Sampaio, 'Modelling timed reactive systems from natural-language requirements,' *Form. Asp. Comput.*, vol. 28, no. 5, pp. 725–765, Sep. 2016, ISSN: 0934-5043. DOI: 10.1007/s00165-016-0387-x.

[22] G. Carvalho, F. Barros, A. Carvalho, A. Cavalcanti, A. Mota and A. Sampaio, 'Nat2test tool: From natural language requirements to test cases based on csp,' in *Software Engineering and Formal Methods*, R. Calinescu and B. Rumpe, Eds., Cham, Switzerland: Springer International Publishing, 2015, pp. 283–290, ISBN: 9783319229690. DOI: 10.1007/978-3-319-22969-0_20.

[23] D. Giannakopoulou, A. Mavridou and J. Rhein, 'Formal requirements elicitation with FRET,' in *International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ-2020)*, Pisa, Italy: National Research Council (CNR), 2020.

[24] S. Vuotto, M. Narizzano, L. Pulina and A. Tacchella, 'Automatic consistency checking of requirements with ReqV,' in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 363–366. DOI: 10.1109/ICST.2019.00043.

[25] R. Lorch et al., 'Formal methods in requirements engineering: Survey and future directions,' in *Proceedings of the 2024 IEEE/ACM 12th International Conference on Formal Methods in Software Engineering (FormaliSE)*, ser. FormaliSE '24, Lisbon, Portugal: Association for Computing Machinery, 2024, pp. 88–99, ISBN: 9798400705892. DOI: 10.1145/3644033.3644373.

[26] U. Waldmann, S. Tourret, S. Robillard and J. Blanchette, 'A comprehensive framework for saturation theorem proving,' *Journal of Automated Reasoning*, vol. 66, no. 4, pp. 499–539, Nov. 2022, ISSN: 1573-0670. DOI: 10.1007/s10817-022-09621-7.

[27] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd. USA: Prentice Hall Press, 2009, ISBN: 0136042597.

[28] D. de Champeaux, 'Faster linear unification algorithm,' *Journal of Automated Reasoning*, vol. 66, no. 4, pp. 845–860, Nov. 2022, ISSN: 1573-0670. DOI: 10.1007/s10817-022-09635-1.

[29] F. M. García-Olmedo, J. García-Miranda and P. González-Rodelas, 'Mathematical foundation of a functional implementation of the cnf algorithm,' *Algorithms*, vol. 16, no. 10, 2023, ISSN: 1999-4893. DOI: 10.3390/a16100459.

[30] G. Masina, G. Spallitta and R. Sebastiani, 'On CNF Conversion for Disjoint SAT Enumeration,' in *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)*, M. Mahajan and F. Slivovsky, Eds., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 271, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 15:1–15:16, ISBN: 978-3-95977-286-0. DOI: 10.4230/LIPIcs.SAT.2023.15.

[31] S. Schulz, 'Simple and efficient clause subsumption with feature vector indexing,' in *Automated Reasoning and Mathematics*. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 45–67, ISBN: 9783642366741.

[32] M. Hajdu, L. Kovács, M. Rawson and A. Voronkov, 'The VAMPIRE approach to induction,' in *Proceedings of the 8th Workshop on Practical Aspects of Automated Reasoning*, B. Konev, C. Schon and A. Steen, Eds., ser. PAAR '22, Haifa, Israel, Aug. 2022. [Online]. Available: https://ceur-ws.org/Vol-3201.

[33] A. Riazanov and A. Voronkov, 'The design and implementation of vampire,' *AI communications*, vol. 15, no. 2, pp. 91–110, 2002.

[34] S. T. March and V. C. Storey, 'Design science in the information systems discipline: An introduction to the special issue on design science research,' *Management Information Systems Quarterly*, vol. 32, no. 4, pp. 725–730, Dec. 2008, ISSN: 0276-7783. DOI: 10.2307/25148869.

[35] R. C. Martin, 'Design principles and design patterns,' *Object Mentor*, 2000. [Online]. Available: https://objectmentor.com/resources/articles/Principles_and_Patterns.pdf.

[36] A. Cline, *Agile development in the real world*. Springer, 2015.

[37] R. Rock-Evans, *Data Modelling and Process Modelling using the most popular Methods: Covering SSADM, Yourdon, Inforem, Bachman, Information Engineering and'Activity/Object'Diagramming Techniques*. Butterworth-Heinemann, 2014.

[38] B. W. Boehm, 'A spiral model of software development and enhancement,' *Computer*, vol. 21, no. 5, pp. 61–72, 1988. DOI: 10.1109/2.59.

[39] S. Rho, P. Martens, S. Shin, Y. Kim, H. Heo and S. Oh, 'Coyote C++: An industrial-strength fully automated unit testing tool,' 2023. arXiv: 2310.14500 [cs.PL]. [Online]. Available: https://arxiv.org/abs/2310.14500.

[40] A. Takanen, J. D. Demott and C. Miller, *Fuzzing for Software Security Testing and Quality Assurance*, 2nd ed. USA: Artech House, Inc., 2018, ISBN: 1608078507.

[41] Working Group 21, 'Programming languages — C++,' International Organization for Standardization, Standard Working Draft ISO/IEC 14882:2026, 2025. [Online]. Available: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2025/n5032.pdf.

[42] T. J. Parr and R. W. Quong, 'Antlr: A predicated-LL(k) parser generator,' *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995. DOI: 10.1002/spe.4380250705.

[43] B. C. Oliveira, M. Wang and J. Gibbons, 'The visitor pattern as a reusable, generic, type-safe component,' *SIGPLAN Not.*, vol. 43, no. 10, pp. 439–456, Oct. 2008, ISSN: 0362-1340. DOI: 10.1145/1449955.1449799.

[44] J. Collins, E. McLean and D. J. Musliner. 'latexmk – Fully automated LaTeX document generation,' Accessed: 22nd Feb. 2026. [Online]. Available: https://ctan.org/pkg/latexmk.

[45] J. Seward, N. Nethercote et al., 'Using Valgrind to detect undefined value errors with bit-precision,' in *USENIX ATC, General Track*, USA: USENIX Association, 2005, pp. 17–30.

[46] K. Serebryany, D. Bruening, A. Potapenko and D. Vyukov, 'Address-sanitizer: A fast address sanity checker,' in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12, Boston, MA: USENIX Association, 2012, p. 28.

[47] J. Weidendorfer, M. Kowarschik and C. Trinitis, 'A tool suite for simulation based analysis of memory access behavior,' in *International Conference on Computational Science*, Springer, 2004, pp. 440–447.

[48] N. Nethercote, 'Dynamic binary analysis and instrumentation,' University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-606, Nov. 2004. DOI: 10.48456/tr-606. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-606.pdf.

[49] Apache Software Foundation. 'Apache Log4cxx: Introduction,' Accessed: 9th Apr. 2026. [Online]. Available: https://logging.apache.org/log4cxx/1.7.0/.

[50] A. Miyazawa and A. L. C. Cavalcanti. 'Autonomous chemical detector,' Accessed: 9th Apr. 2026. [Online]. Available: https://robostar.cs.york.ac.uk/case_studies/autonomous-chemical-detector/autonomous-chemical-detector.html.

[51] Y. Murray, D. A. Anisi, M. Sirevåg, P. Ribeiro and R. S. Hagag, 'Safety assurance of a high voltage controller for an industrial robotic system,' in *Formal Methods: Foundations and Applications*, G. Carvalho and V. Stolz, Eds., Springer International Publishing, 2020, pp. 45–63, ISBN: 978-3-030-63882-5.

[52] I. P. Gent, P. Nightingale and A. Rowley, 'Encoding quantified CSPs as quantified boolean formulae,' in *Proceedings of the 16th European Conference on Artificial Intelligence*, ser. ECAI'04, Valencia, Spain: IOS Press, 2004, pp. 176–180, ISBN: 9781586034528.

[53] S. Condon. '`SCFirstOrderLogic.Inference.Basic` source repository,' Accessed: 9th Apr. 2026. [Online]. Available: https://github.com/sdcondon/SCFirstOrderLogic.Inference.Basic.