Formal Requirements Engineering Toolkit: PBRX Project Proposal

Oliver Dixon
20th October 2025

Contents

1 Overview of Proposal

1.1 Premise

This document outlines a potential undergraduate Computer Science independent project to be completed for the 40-credit PBRX module between S1/5 and S2/10 during the 2025/26 academic year. The proposal concerns development of a graphical desktop-based software application ("the Tool") to support corporate Requirements Engineering (RE), whereby requirements may be associated with sentences in First-Order Logic (FOL) and subject to automated reasoning.

In addition to machine-assisted reasoning, the Tool should include capabilities for organisation of requirements in complex projects consisting of multiple subsystems, assignment of requirements to stakeholders, generation of requirement reports, and integration with software unit-testing frameworks.

1.2 Rationale

Within the Software Development Lifecycle (SDLC), gathering and organisation of customer requirements forms a major component of the first stage: Planning. Despite significant advancements in System Design, Implementation, Testing, and Deployment, RE is often an unscientific process which, when done incorrectly, may cause serious and irreparable project-level issues for engineering teams.

Advancements in RE are often domain-specific and tightly integrated within larger SDLC-management tools. Examples can be seen in the following software suites:

- *D-RisQ Kapture* — a modularised component of the *D-RisQ Modelworks* software-engineering suite; capable of indexing and verifying formally specified requirements, while also allowing for tracing of software-level requirements to system-level requirements.
- *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.
- *IBM Rational DOORS* — the Digital Object-Oriented Requirements System from IBM, allowing a high degree of complex requirement organisation and integration with other engineering tools, such as *IBM Rational Rhapsody*, but lacking any native consistency-checking or requirement verification.
- *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; largely based on the namesake patent US7970601B2.

The Tool should unify the standalone functionality of the existing solutions whilst avoiding domain-specific restrictions. Four fundamental components of the Tool have been identified through reviews of existing solutions, and are elaborated upon in Section 2.

2 Core Functionality of the Tool

2.1 Specification and Organisation of Requirements

The atomic unit within the Tool is the *Requirement*. A Requirement may consist of arbitrary metadata, a human-readable description, and principally, a FOL sentence to formally express the semantics of the Requirement. As all non-trivial engineering projects are likely to consist of thousands of Requirement-like objects, collated into mutually independent *Subsystems*, the Tool must provide a visual and intuitive feature-set to support the definition and organisation of Requirements within a *Project*.

Within a Subsystem, Requirements should be further separable into *Analysis Groups*, *Test Groups*, and *Reporting Groups* for the purpose of formal reasoning, automated software-level test associations, and report-generation, respectively.

This will require the specification and implementation of an internal storage object model within the software.

2.2 Automated Reasoning of FOL Knowledge Bases

Aggregation of Requirements under Analysis Groups forms a FOL *Knowledge Base* (KB) with the sentences associated with the included Requirements. More precisely, the KB is formed by the sets of literals produced when the sentences are normalised into Conjunctive Normal Form (CNF). Equation (1) and Equation (2) demonstrate CNF for a statement composed of M disjunctive groups under conjunction, each consisting of iN_i literals L.

$$
\begin{aligned}
&(L_{11} \vee \ldots \vee L_{1N_1}) \wedge \ldots \wedge (L_{M1} \vee \ldots L_{MN_M}) && (1)\\
=&\{\{L_{11}, \ldots, L_{1N_1}\}, \ldots, \{L_{M1}, \ldots, L_{MN_M}\}\} && (2)
\end{aligned}
$$

Any FOL sentence can be transformed into an equivalently satisfiable CNF formula through a pipeline of morphisms:

1. Substitution of implications and biconditionals with equivalent binary operators;
2. Application of de Morgan's Laws such that only predicates may be negated;
3. Introduction of Skolem functions to eliminate existential quantifiers, such that $\forall x_1 \ldots \forall x_N \exists y P(y) \mapsto \forall x_1 \ldots \forall x_N P(S(x_1, \ldots, x_N))$, where S is the synthesised Skolem function;
4. Elimination of universal quantifiers such that $\forall x P(x) \mapsto P(X)$, where the quantified variable x is transformed to a global constant X; and
5. Distributing disjunctive clauses over conjuncts to transform the expression into the form described by Equation (1).

Given a FOL KB of CNF clauses, the Tool should be capable of accepting arbitrary queries from the user, such as predicate assertions, and providing a traced deduction through the KB. Algorithmic deduction can be (trivially) implemented with a *binary unification* algorithm (Equation (3)) to determine substitution maps under which similar clauses achieve equality[1], coupled with a *binary resolution* (Equation (4)) algorithm to eliminate redundant clauses and derive contradictions in the KB.

$$\text{Unify}(p, q) := \theta \text{ such that } \text{Sub}(\theta, p) = \text{Sub}(\theta, q) \tag{3}$$

$$\frac{\Gamma_1 \cup \{L_1\} \quad \Gamma_2 \cup \{L_2\}}{(\Gamma_1 \cup \Gamma_2)\,\phi}\ \phi, \text{ where } \phi \text{ is the MGU of } L_1 \text{ and } \overline{L_2}. \tag{4}$$

This approach allows the Tool to deduce on arbitrary, operator-entered queries in addition to model consistency verification. The Tool should also be capable of producing a graphical representation of the deduction, allowing the operator to inspect the derived MGUs and resolution steps from within the GUI.

2.3 Integration with Unit Testing

In addition to FOL sentences, Requirement objects should support optional associations with one or many software unit tests. In particular, the Requirement may be "tested" by the satisfaction of pre-defined unit tests. To support this capability, the Tool must provide controls for specifying an arbitrary number of unit tests for each Requirement, grouping test-endowed Requirements into Test Groups, transparently executing the tests from within the Tool, and interpreting

[1]Most unification algorithms attempt to locate the *most general unifier* (MGU). The MGU is the substitution map under which the clauses contain the minimum number of free variables.

the results. A test driver backend should be provided for Google Test, and be sufficiently extensible to support drivers for additional unit-testing frameworks in the future.

The Tool should also display the results of any failed tests in a dedicated section, combined with any additional metadata provided by the framework. Crucially, the choice of unit-testing framework should be abstracted from the user, and the Tool should support the use of multiple frameworks within any single Requirement.

2.4 Report Generation

Formatted human-readable reports should be produced from within the Tool by the operator. Reports should contain the index of Requirements, including all metadata and the corresponding LaTeX-typeset FOL statement, the results of deductions performed by the operator on Analysis Groups, and any applicable results of software unit-testing on Test Groups.

Exported reports should be available in a variety of formats; for the initial iteration of the Tool, LaTeX and HTML should be available for generation. In the LaTeX case, the Tool should produce a PDF by generating the code and invoking a suitable compiler on the host system.

3 Additional Functionality

Additional capability, such as serialisation and recovery of the data to complex formats (e.g. relational databases) should not be scoped into the initial project specification due to time constraints. Future work could involve the construction of a client-server model to enhance collaboration features across a set of enterprise users, or optimisation of the deduction algorithms to make use of linear-bounded unification or many-operand resolution. Many of the existing features could be trivially extended for future work, such as integration with more unit-testing frameworks, or additional export formats for report-generation.