Automating Theory Repair in First Order Logic

Thomas Wong

Master of Science
Cyber Security, Privacy and Trust
School of Informatics
University of Edinburgh
2023

Abstract

Automatic theory repair systems help identify and repair faults in a knowledge base, which has useful applications in artificial intelligence such as decision systems in autonomous vehicles. The ABC system is a state-of-the-art of such systems which combines three existing techniques: abduction, belief revision and conceptual change, but with a limitation that it only accepts Datalog logic. To enhance its expressive power, this study extends the ABC system to first-order logic (ABC_FOL), by augmenting the fault detection module and adding new repair plans and heuristics to the system. The resultant extended system is able to correctly identify faults and generate sensible repairs across a diverse set of first-order logic examples that cannot be expressed in Datalog logic.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Thomas Wong*)

Acknowledgements

I am profoundly grateful to my supervisors, Dr. Xue Li and Prof. Alan Bundy, for their unwavering support throughout the duration of this project. Their prompt responsiveness to my inquiries and their dedication to organizing biweekly meetings to monitor my progress have been instrumental. The achievement of the project's objectives would not have been as attainable without their invaluable guidance and assistance.

Furthermore, I extend my heartfelt appreciation to my parents, whose unyielding support and belief in me have been a constant source of strength. Their encouraging words have carried great significance and have been pivotal in sustaining my momentum throughout the project's execution.

Lastly, I extend my gratitude to all my colleagues and peers who engaged in discussions with me and shared their valuable insights for this project. I would like to extend a special thank you to Pak Yin Chan, a fellow student in the ABC lab. Our collaborative efforts have resulted in numerous insightful discussions and a fruitful exchange of ideas, greatly contributing to the progress of both of our projects. These interactions have fostered a sense of mutual support and reassurance, reminding me that I am not navigating this project alone.

Table of Contents

Chapter 1

Introduction

1.1 Motivation and Hypothesis

Automated theory repair is a subfield of artificial intelligence (AI) that concentrates on creating algorithms and techniques for automatically correcting discrepancies or errors present in logical theories or knowledge bases. Logical theories play a pivotal role in various AI tasks, encompassing question answering, planning, learning, and reasoning [1]. Particularly, the task of automated reasoning with logical theories has been gaining increasing attention in the AI community, notably as applications like autonomous vehicles gain popularity. Given the paramount significance of ensuring the safety and correctness of decisions made by autonomous vehicles, there's a pronounced preference for advanced automated reasoning systems over the deployment of machine learning (ML) models [18]. This is due to the inherent imperfection of ML models, which often lack complete accuracy, and the opacity of their outputs, rendering it challenging to explain their reasoning—a crucial aspect in scenarios such as accident investigations.

Logical theory sets frequently serve as representations of an agent's environment. For instance, this could entail a collection of traffic rules that an autonomous vehicle must adhere to. This representation is far from static and is susceptible to faults, particularly when the environment undergoes changes or the agent confronts new kinds of objectives [8]. An instance of a flawed theory is illustrated in Example 1.

Example 1: Faulty Definition of Naturals

$$
\begin{aligned}
&\implies naturals(2).\\
naturals(X) &\implies naturals(suc(X)).
\end{aligned}
$$

In this theory, it is posited that if X represents a natural number, then the successor of X (defined as $suc(X) := X + 1$) is also a natural number. However, an erroneous base case is defined as $naturals(2)$, leading to the failure of proving the proposition $naturals(1)$ within this theory. This serves as an example of insufficiency — a true conjecture that cannot be proven by the theory. Such errors often stem from human mistakes, a common programming pitfall caused by overlooking edge cases.

In light of these challenges, Li and Bundy proposed the ABC system (ABC) [8, 23], a domain-independent program scripted in Prolog aimed at generating repairs for faulty logical Datalog theories. ABC harnesses three repair techniques — abduction, belief revision, and conceptual change — to rectify both incompatibility (when a false conjecture is proven) and insufficiency (when a true conjecture cannot be proved) present within logical theories. ABC's prowess lies in its capability to process any Datalog logical theory and yield comprehensive repairs.

Though ABC is undeniably a groundbreaking domain-independent theory repair system, its applicability is restricted by the expressive confines of Datalog theory. Datalog fails to capture negations, functions, and existential quantifiers, thus falling short in representing the diverse real-world scenarios such as legal statutes, intricate mathematical equations, and complex interrelationships within a social network.

This project's core challenge revolves around the expansion of ABC to provide support for first-order logic (FOL). The extended system should be able to accommodate a logical theory expressed in first-order Logic, detect any underlying faults (incompatibility or insufficiency), devise repairs for these issues, and generate a fault-free set of logical theories. Crucially, this extended system must preserve its intrinsic domain-independence and retain backward compatibility — capable of generating all feasible repairs for a Datalog theory, as needed. The research hypothesis is formulated as follows:

Hypothesis: The techniques within ABC — abduction, belief revision, and conceptual change — can be adapted for repairing general domain-independent theories expressed in first-order logic, yielding a subset of potential repairs, although not necessarily encompassing all feasible solutions.

It is a well-established fact that first-order logic, in its general form, is undecidable — a conclusion derived through a reduction to Turing's proof of the Entscheidungsproblem [33]. As a consequence, we refrain from asserting that ABC possesses the capability to discover all conceivable solutions for a flawed theory, given that the defect might elude initial detection. Nevertheless, our aspirations are centered on the anticipation that the extended ABC system will exhibit the capacity to identify and present a diverse array of repairs for the majority of flawed theories, particularly those comprising of simpler theorems. The effectiveness of the extended ABC system is shown in subsequent sections.

1.2 Contributions

The contributions of this project are summarized as follows:

- Implementation of an automatic theorem prover for first-order logic. Distinguishing itself from existing FOL theorem provers, the ABC theorem prover generates partial proofs (incomplete proofs that do not establish a required conjecture). This unique feature aids in the generation of repairs.
- Extension of ABC's existing repair strategies to accommodate first-order logic.
- Design and implementation of novel repair strategies tailored specifically to FOL theories.
- Development of a set of flawed FOL examples, serving both for evaluating the ABC system and for future comparisons with similar systems.
- Evaluation and analysis of the extended system, which provides insights into its limitations and offers heuristics and directions for enhancing the ABC system in the future.

1.3 Structure of the Thesis

The remainder of the thesis is structured as follows: Chapter 2 provides readers with essential background knowledge and concepts, offering the foundational understanding necessary to comprehend the contributions introduced in the project. Chapter 3 explains the methodology adopted for the project. Detailed implementation specifics of the extended ABC system are provided, offering an in-depth understanding the modifications made to accommodate first-order logic. Chapter 4 delves into the evaluation process undertaken to assess the effectiveness and functionality of the extended ABC system. Detailed analysis of the results obtained is presented, shedding light on the performance of the system. Chapter 5 offers insights into the limitations inherent in the project and discusses potential avenues for future work and further enhancements to the extended ABC system.

Chapter 2

Background

2.1 Litertaure Review

Automated theorem proving in first-order logic is a well-investigated topic, with multiple techniques developed to prove or disprove theories from knowledge bases [28]. A common technique is the resolution method, initially proposed by Robinson [29]. Resolution proves a theorem by demonstrating the unsatisfiability of its negation: the objective is negated, then resolved through unification with a clause containing contradictory literals. A solution is achieved upon deriving the empty clause (or equivalently, false). This method is widely used in automated theorem provers, including prominent ones like Prover9 [25].

Additional techniques have also been developed, such as the tableaux method [32]. It depicts theories as a tree-like structure and pursues proofs by opening and closing branches. A goal is considered unsatisfiable when all branches are closed. This method is utilized in various theorem provers, like Isabelle [26].

Furthermore, other proof techniques include Model Elimination [24], genetic algorithms [31], and machine learning [5], though they are comparatively less widespread.

Conversely, automated theory repair is a field often overlooked. As far as our knowledge extends, ABC stands as the only theory repair system that (i) maintains domain independence, (ii) integrates diverse repair techniques, and (iii) exclusively relies on automated reasoning methods. Other literature often centers on fault detection in knowledge databases or employs machine learning for learning and generating repairs.

In the field of knowledge engineering, theory repair within knowledge graphs (KG) gains the most attention, with applications spanning Semantic Web and Information Retrieval. This avenue of research concentrates on spotting and rectifying inconsistencies within large-scale knowledge graphs, often extracted from social networks or expansive databases. Numerous recent approaches target the embedding algorithm of knowledge bases. For instance, Wiharja et al. [34] introduced Schema Aware Triple Classification (SATC), facilitating the detection of inconsistencies amid entities during knowledge graph completion. Additionally, Du et al. [13] adapt embedding into an optimization problem that enforces logical consistency. Nevertheless, most prevailing research does not seamlessly combine automatic inconsistency detection and repair. The scale of large KGs often prevents comprehensive automatic inconsistency checks, while many repairs prove excessively generic, demanding substantial manual intervention. The efforts of ABC strive to address this issue within a narrower scope of theory sets, offering insights into repairing extensive knowledge graphs.

In the domain of machine learning techniques, it's also worth noting that Machine Learning (ML) is a prominent avenue of research in theory repair systems. For instance, ML is applied to learn and discover equations, especially in revising quantitative scientific models (such as [3, 30]), which can subsequently be employed to unveil and revise inconsistent equations. Nonetheless, ML primarily learns from training datasets and summarizes existing patterns, potentially lacking the ability to generalize across domain-independent theories if the data is not diverse and large-scale enough. Other challenges include the explainability of ML models and limited training data availability. In contrast, ABC relies on explicit repair algorithms, thereby circumventing the need for ML techniques.

2.2 First Order Logic

First Order Logic (FOL) is a formal logic system that extends propositional logic by introducing variables, predicates, and quantifiers. FOL boasts more powerful expressive abilities than propositional logic, allowing it to succinctly represent a wide array of natural language statements [32].

2.2.1 Definition

An inductive definition of FOL consists of the following elements:

- Constants: these are terms that represent individual entities. This includes numbers (e.g., 1, 2) as well as constant entities like ”william”, ”cat”, and ”edinburgh”.
- Variables: Terms such as X, Y, and Z.

- Functions: map terms to other terms. For instance, if $t_1, \ldots, t_n$ are terms and f is an n-ary function, then $f(t_1, \ldots, t_n)$ constitutes a term. Common examples include *sqrt* and *avg*.
- Predicates: These define relationships between terms. If $t_1, \ldots, t_n$ are terms and p is an n-ary predicate, then $p(t_1, \ldots, t_n)$ forms a formula.
- Connectives: $\vee$ (conjunction), $\wedge$ (disjunction), $\neg$ (negation), $\Rightarrow$ (implication), and $\Leftrightarrow$ (equivalence). Given two formulas F and G, valid formulas include $F \vee G$, $F \wedge G$, $\neg F$, $F \Rightarrow G$, and $F \Leftrightarrow G$.
- Quantifiers: $\exists$ (existential quantifier), $\forall$ (universal quantifier). If F is a formula and x is a variable, then $\exists x F$ and $\forall x F$ are valid formulas.
- Equality: $=$

2.2.2 Clausal Form

To facilitate automated reasoning, the FOL formula is converted into clausal form using equivalence rules and existential quantifier elimination via skolemization [6]. Consider the formula:

$$\forall x \exists y. Love(x,y) \implies Love(y,x)$$

The first step is removing implications:

$$\forall x \exists y. \neg Love(x,y) \vee Love(y,x)$$

Negations are then pushed down to atomic formulas (already done here):

$$\forall x \exists y. \neg Love(x,y) \vee Love(y,x)$$

Subsequently, existential quantifiers are eliminated through skolemization: variables bound by the existential quantifier are replaced with a function of the variables bound by universal quantifiers.

$$\forall x. \neg Love(x, f(x)) \vee Love(f(x), x)$$

Universal quantifiers are then dropped:

$$\neg Love(x, f(x)) \vee Love(f(x), x)$$

Finally, conjunctions are distributed over disjunctions, which makes no change in this example:

$$\neg Love(x, f(x)) \vee Love(f(x), x)$$

2.2.3 Terms and Naming Conventions

It is noteworthy that, within discussions of theory examples in this thesis, the following naming convention aligns with Prolog's conventions:

- Variables are in uppercase, while constants, predicates, and functions all begin with a lowercase letter.
- In clausal form, each negated predicate is a *negative literal*, whereas non-negated predicates are *positive literals*.
- Clauses with only positive literals are *assertions*, and those with only negative literals are *constraint axioms*. All other clauses are called *rules*.
- Clauses with at most one positive literal are *Horn clauses*, while those with more are *non-Horn clauses*.

2.2.4 Differences between First Order Logic and Datalog

Datalog, a declarative logic programming language, is designed to be a decidable subset of first-order logic and is popular in the database community [10]. The primary distinctions between FOL and Datalog theories are as follows:

- Datalog excludes negations, functions and existential quantification.
- In Datalog, all rules and axioms are expressed as Horn clauses, while FOL allows for non-Horn clauses.

The primary focus of this project is to address these differences when extending the ABC system to support first-order logic.

2.3 Unification

Unification is the process of determining if two terms can be made equivalent through a substitution known as the most general unifier (mgu). Formally, a unification problem is defined as a conjunction $e_1 \equiv e'_1 \wedge \ldots \wedge e_n \equiv e'_n$, where e_i and e'_i are terms or predicates. A unifier is the solution to a unification problem, represented as a substitution σ, ensuring that $e_i\sigma$ is identical to $e'_i\sigma$ for all $i \in [n]$. An mgu is a unifier σ such that for any other unifier σ', there exists a substitution σ'' such that $\sigma' = \sigma \oplus \sigma''$ [9].

The unification algorithm is outlined in Table 2.1. Given a unification problem that matches the **Before** column, the algorithm checks the **Condition**. If it evaluates to true, the algorithm transforms the problem into the pattern as described in the **After** column.

It then proceeds recursively until either termination or failure. The algorithm records the substitutions made in σ and returns it if the algorithm successfully terminates.

Case	**Before**	**Condition**	**After**
Base	$\top;\sigma$		Terminates
Trivial	$s \equiv s \wedge u;\sigma$		$u;\sigma$
Decomp	$F(\vec{s}^m) \equiv F(\vec{t}^n) \wedge u;\sigma$		$\bigwedge_{i=1}^n s_i \equiv t_i \wedge u;\sigma$
Clash	$F(\vec{s}^m) \equiv G(\vec{t}^n) \wedge u;\sigma$	$F \neq G \vee m \neq n$	fail
Orient	$t \equiv x \wedge u;\sigma$		$x \equiv t \wedge u;\sigma$
Occurs	$x \equiv s \wedge u;\sigma$	$x \in \mathcal{V}(s) \wedge x \neq s$	fail
VarElim	$x \equiv s \wedge u;\sigma$	$x \notin \mathcal{V}(s)$	$u\{x/s\};\sigma \oplus \{x/s\}$

Table 2.1: Standard Unsorted Unification Algorithm: *F and G are functors; t is a non-variable term and s is any term; s_i and t_j are terms; $m,n > 0$; x and y are distinct variables; and u is a unification problem; $s \equiv t$ is the problem of unifying s and t; and $\mathcal{V}(t)$ is a set of free variables of term t.[9].*

In ABC, the implemented unification algorithm does not contain the *Occurs* case as this was unnecessary in Datalog theories. In this project, the Occurs check, an algorithm that deals with the *Occurs* case, is added back to the unification function.

2.4 Linear Resolution with Selection Function

Linear Resolution with Selection Function (SL-resolution) is first proposed by Kowalski and Kuener [19] and serves as the cornerstone of ABC's automated theorem prover. Notably, SL-resolution is both sound and complete [15], although it is undecidable in the context of first-order logic. This method involves initially negating the objective and subsequently unifying it with a clause containing contradictory literals. Successful resolution terminates with deriving the empty clause (or equivalently, false).

The selection function pertains to the choice of the literal (referred to as sub-goals) within the objective to be resolved. ABC adheres to a left-most selection strategy, always opting for the left-most sub-goal initially. Illustrating SL-resolution within ABC is the subsequent theory example:

Example 2: Bird Theory

$$\begin{aligned} bird(X) &\implies fly(X) \\ penguin(Y) &\implies bird(Y) \\ &\implies penguin(lucy) \end{aligned}$$

Suppose the aim is to establish $fly(lucy)$. Commencing with the negation of the goal:

$$fly(lucy) \implies$$

At this stage, all literals assume the role of goals to be resolved throughout the resolution process. The following steps then recur until the empty clause is deduced (signifying no remaining sub-goals):

- One sub-goal is chosen for resolution — in this instance, $fly(lucy)$.
- An input literal from the theory is subsequently selected to undertake the resolution.
- The unification algorithm is run to see if the two literals can be made identical. If yes, the sub-goal is resolved and the remaining literals in the input clause is added to the goal, with the substitutions output from unification applied to the whole goal. Otherwise, choose another literal from the theory.

Each iteration of this cyclic procedure is called a Resolution Step (RS). The entire inference process unfolds as follows:

$$\dfrac{\dfrac{\dfrac{fly(lucy) \implies \quad bird(X) \implies fly(X)}{bird(lucy) \implies} \quad penguin(Y) \implies bird(Y)}{penguin(lucy) \implies} \quad \implies penguin(lucy)}{\implies}$$

Figure 2.1: The inference of $fly(lucy)$

In this project, SL-resolution is reused for fault detection in FOL, enriched with the integration of ancestor resolution, as further elaborated in section 3.2.

2.5 Abduction

Abduction is a technique employed to identify the most probable hypothesis that can explain a given set of observations. It is one of the primary repair techniques employed within ABC. Unlike deductive reasoning, which derives conclusions from existing theory, abduction focuses on introducing new hypotheses into the theory set to enable the deduction of the target observation [12]. Formally, abduction can be defined as follows: given a theory set P, a truth set T, a query Q, and a set A of ground abducible atoms, the goal of abduction for a query Q is to identify a subset $\Delta \subseteq A$ satisfying the following conditions: (1) $P \cup \Delta \models Q$; (2) $P \cup \Delta \models T$; and (3) $P \cup \Delta$ remains consistent.

In the context of ABC, the abduction technique is primarily employed for repairing insufficiencies. Cox and Pietrzykowski's algorithm [11] is utilized to add supplementary axioms or delete preconditions to establish properties in the truth set. According to Cox and Pietrzykowski, a cause c for an observation e (obtained through abduction) should possess the following properties (here, K denotes the knowledge base) [11]:

- *Minimal*: $\forall c'$, if $c \implies c'$ then $c \equiv c'$.
- *Consistent*: $c \cap K$ is satisfiable.
- *Nontrivial*: $e \implies c$ does not hold.
- *Basic*: all consistent cause of c is trivial.

A cause that satisfies all these properties is referred to as a *fundamental* cause. Readers can refer to [22] for the detailed process of identifying a fundamental clause in ABC.

In this project, the abduction technique remains largely unchanged. However, it's important to note that in the resolution step for finding fundamental clause, the search process might not terminate in an FOL theory. To address this, various heuristics are introduced, including loop detection and search depth limits, as described in section 3.4.

2.6 Belief Revision

Belief revision involves the task of modifying or relinquishing some of the initial input theories to accommodate a new belief, ensuring the coherence and consistency of the theory set. A significant challenge in belief revision arises when there are multiple possible old beliefs to be removed, and a choice must be made to ensure minimum modification.

Gärdenfors [16] formally outlined belief revision with six fundamental postulates, which serve as widely accepted constraints guiding the implementation of belief revision processes. Readers can refer to [16] for those postulates.

In the context of ABC, belief revision plays a pivotal role in incompatibility repairs, where axioms are either removed or unprovable conditions are introduced to reconcile inconsistencies. ABC employs the maxi-choice contraction model to determine and delete the minimum number of axioms necessary, as described in [22]. Maxi-choice contraction, introduced by Alchourron and Makinson [2], seeks to identify a maximal consistent subset. Formally, given belief ϕ to be contracted, K_1 is the maximal subset of belief set K that doesn't entail ϕ, if and only if:

1. $K_1 \subseteq K$,
2. $\phi \notin Cn(K_1)$,
3. for any K_2 such that $K_1 \subset K_2 \subseteq K$, $\psi \in Cn(K_2)$.

Here, $Cn(K)$ denotes the set of all logical consequences of K, and ψ represents a revised belief that replaces ϕ.

In this project, while the belief revision technique remains largely unchanged, a variant of belief revision is explored. This variant aims to introduce unresolved alternatives to mitigate incompatibilities, as discussed in Chapter 3.3.3.

2.7 Reformation

Conceptual change in the context of the ABC system is facilitated by the reformation algorithm, initially proposed by Bundy and Mitrovic [9]. This algorithm aims to repair FOL theories through a sequence of block and unblock operations that target a specific unification step. In reformation, the goal of a block operation is to obstruct an unwanted yet successful proof while the unblock operation enables a partial but desired proof. Unlike abduction and belief revision, reformation doesn't merely involve the addition or deletion of axioms; it proposes conceptual changes by means such as function and predicate splitting, merging, or altering the arity of functions. Remarkably, reformation boasts a self-inverse property, where a blocking operation can be undone by an unblocking operation, and vice versa. The reformation algorithm is depicted in Table 2.2.

Case	**Before**	**Condition**	**Block**	**Unblock**
Base	$\top$		Failure	Success
CC_s	$F(\vec{s}^m) \equiv G(\vec{t}^n)$ $\wedge u$	$F = G$ $\wedge\ m = n$	Make $F(\vec{s}^m) \neq F(\vec{t}^m)$ $\bigvee_{i=1}^n$ Block $s_i \equiv t_i$ $\vee$ Block u	$\bigwedge_{i=1}^n$ Unblock $s_i \equiv t_i$ $\wedge$ Unblock u
CC_f		$F \neq G$ $\vee\ m \neq n$	Success	Make $F(\vec{s}^m) = G(\vec{t}^n)$ $\bigwedge_{i=1}^n$ Unblock $\nu(s_i) \equiv \nu(t_i)$ $\wedge$ Unblock $\nu(u)$
VC_s	$x \equiv t \wedge u$ or $t \equiv x \wedge u$	$x \notin \mathcal{V}(t)$	Make $x \in \mathcal{V}(t)$ $\vee$ Block $u\{x/t\}$	Unblock $u\{x/t\}$
VC_f		$x \in \mathcal{V}(t)$	Success	Make $x \notin \mathcal{V}(t)$ $\wedge$ Unblock $\nu(u\{x/t\})$

Table 2.2: The Reformation Algorithm for First-Order Logic, where a block operation invalidates a successful unification, and an unblock operation achieves a previously failed unification: *F and G are functors; t is a non-variable term and s is any term; s_i and t_j are terms; $m, n > 0$; x and y are distinct variables; and u is a unification problem; $s \equiv t$ is the problem of unifying s and t; and $\mathcal{V}(t)$ is a set of free variables of term t. CC refers to a rule that is applicable when the inputs are both compound terms or constants, and VC when just one of the inputs is a variable. Their s and f subscripts indicate rules resulting in success and failure, respectively [9].*

In ABC, reformation is instrumental in both insufficiency and incompatibility repairs. It's important to note that the operations "Make $x \in \mathcal{V}(t)$" and "Make $x \notin \mathcal{V}(t)$" are not utilized in ABC_Datalog. This is due to their inapplicability to Datalog theories, which lack the presence of functions.

In this project, the reformation algorithm undergoes modifications, specifically by adding the previously overlooked operations "Make $x \in \mathcal{V}(t)$" and "Make $x \notin \mathcal{V}(t)$". Further enhancements are made to the operations "Make $F(\vec{s}^m) = G(\vec{t}^n)$" and "Make $F(\vec{s}^m) \neq G(\vec{t}^n)$" to incorporate nested support for functions. These adjustments are explained in greater detail in section 3.3.

2.8 The ABC System

The ABC repair system [23] integrates the capabilities of abduction, belief revision, and conceptual change to automatically rectify faulty theories using Prolog. In its prior version, the input is expected to be a Datalog-like theory. The pipeline of the ABC system is depicted in Figure 2.2.

ABC accepts the a *Theory* and a *Preferred Structure* as input. The *Theory* (denoted by $\mathbb{T}$) consists of logical formulas describing the facts and rules of a particular scenario, while the *Preferred Structure* (denoted by $\mathbb{PS}$) dictates the propositions that is desired to be provable or unprovable by $\mathbb{T}$. The $\mathbb{PS}$ is further split into two sets: the true Set $\mathcal{T}(\mathbb{PS})$ and the false set $\mathcal{F}(\mathbb{PS})$. True set is the set of ground propositions that should be provable by $\mathbb{T}$, while the false set is the set of ground propositions that should not be provable by $\mathbb{T}$. An example of valid input is shown in listing 2.1: lines 1-5 are the Theory, lines 8-9 are the Preferred Structure.

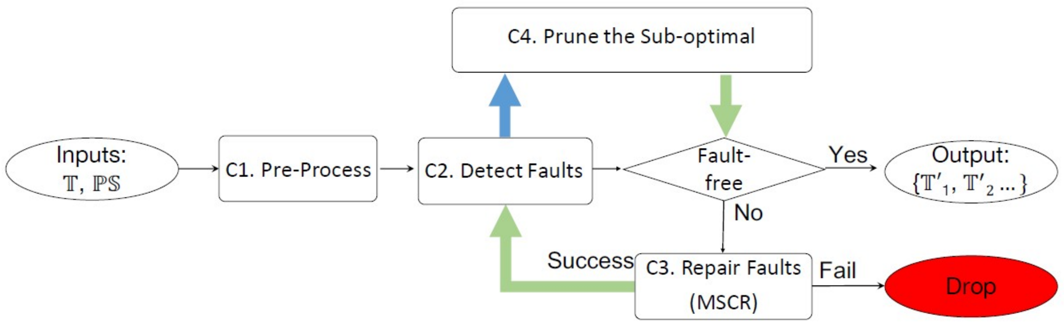
C4. Prune the Sub-optimal
Inputs: $\mathbb{T}$, $\mathbb{PS}$
C1. Pre-Process
C2. Detect Faults
Fault-free
Yes
Output: $\{\mathbb{T}'_1, \mathbb{T}'_2 \ldots\}$
No
Success
C3. Repair Faults (MSCR)
Fail
Drop

Figure 2.2: Pipeline of ABC, cited from [23].

```
1 axiom([+penguin(tweety)]).
2 axiom([+bird(polly)]).
3 axiom([-bird(\x), +fly(\x)]).
```

```
axiom([-penguin(\y), +bird(\y)]).
axiom([-bird(\y), +feather(\y)]).

%Preferred Structure
trueSet([feather(tweety),feather(polly), fly(polly)]).
falseSet([fly(tweety)]).
```

Listing 2.1: A minimal example of a valid ABC input

In ABC, the propositions in the true and false sets are restricted to be *ground assertions*, meaning that they should be a single positive literal with no variables. This restriction helps keep the preferred structure inline with the observations of real world and simplifies the inference process. In the extended ABC system, this restriction will be conserved.

Following the initial preprocessing of inputs (which involves checking for self-contradictions in $\mathbb{PS}$ and pruning redundant theories), the system enters the main loop where processes C2, C3, and C4 iteratively execute until the theory is completely free of faults or certain faults prove to be irreparable. Specifically:

- C2 employs SL resolution to detect faults and subsequently conveys proofs or failed proofs to C3.
- C3 computes maximal sets of commuting repairs (MSCR) and generates repairs using abduction, belief revision, and reformation for faults within different MSCRs in parallel.
- Lastly, C4 prunes suboptimal theories to streamline further processing.

The primary focus of this project is to introduce changes to C2 and C3 within the ABC system. For C2, the challenge lies in extending fault detection techniques to detect faults in first-order logic, which can be potentially undecidable. Meanwhile, C3 necessitates a thorough exploration of how abduction, belief revision, and conceptual change can be applied to the realm of FOL, encompassing functions, negations, and other intricate constructs. These modifications are elaborated upon in greater detail in the subsequent sections.

In the remaining sections, the old system is denoted as ABC_Datalog while the extended system is denoted as ABC_FOL.

Chapter 3

Implementation

3.1 Methodology

The approach to this project involves a structured series of steps, as outlined below:

1. **Reviewing literture**: A comprehensive review of the existing literature on ABC_Datalog [23, 8] is conducted to gain an in-depth understanding of its underlying principles and mechanics.
2. **Reviewing codebase**: Given the implementation-focused nature of the project, a thorough examination of ABC_Datalog codebase [21] is carried out. This review aims to identify areas for potential extensions and enhancements.
3. **Creating examples set**: A collection of well-designed knowledge theories is created. These theories are expressed in first-order logic and serve as illustrative examples to demonstrate the repair process and facilitate system evaluation.
4. **Creating implementation plan**: Drawing insights from the literature and in consultation with project supervisors, a detailed implementation plan is formulated. This plan outlines the specific functions that need to be developed and the algorithms to be employed for their realization.
5. **Implementing extensions**: The designated extensions are implemented and seamlessly integrated into ABC_Datalog to become ABC_FOL. The developed code is open-sourced and shared on GitHub [14], fostering collaboration and accessibility.
6. **Testing and evaluation**: Rigorous testing is conducted on ABC_FOL using a range of test theories. This process aims to identify and rectify any potential bugs or issues. Additionally, the extended system is evaluated using defined metrics, showcasing its robustness and providing empirical evidence to support the research hypothesis.

3.2 Fault Detection

3.2.1 Overview

The following flowchart in Figure 3.1 illustrates the sequence of steps within the fault detection module. An explanation of each component is presented below:

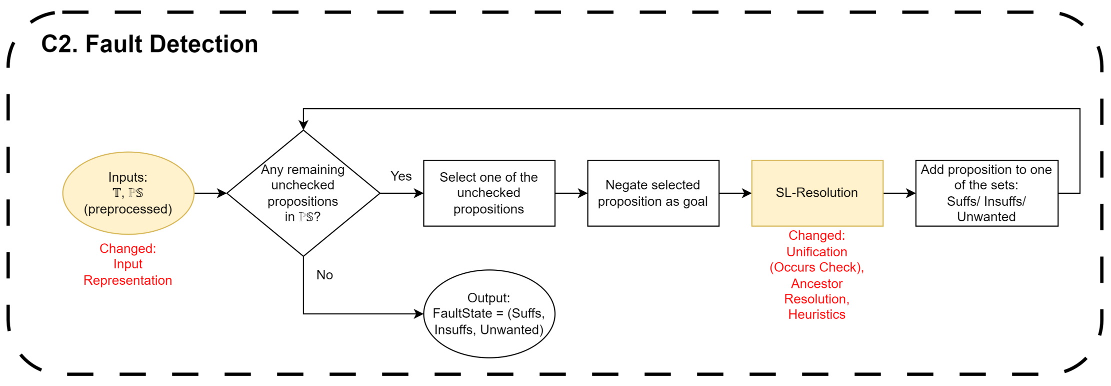
C2. Fault Detection
Inputs: 𝕋, ℙ𝕊 (preprocessed)
Changed: Input Representation
Any remaining unchecked propositions in ℙ𝕊?
Yes
No
Select one of the unchecked propositions
Negate selected proposition as goal
SL-Resolution
Changed: Unification (Occurs Check), Ancestor Resolution, Heuristics
Add proposition to one of the sets: Suffs/ Insuffs/ Unwanted
Output: FaultState = (Suffs, Insuffs, Unwanted)

Figure 3.1: Flowchart of Fault Detection Module (Changes implemented in this project are highlighted in red)

- Initially, the input theory and prederred structure are provided to the module. Notably, this project extends the input representation to accommodate functions and equality axioms, as elaborated in section 3.2.2.
- Then, each proposition in the preferred structure (True sets and False sets) are checked one by one. This process forms an iterative loop as depicted in the flowchart.
- In each iteration, the designated proposition is initially negated, establishing it as the goal. Subsequently, SL-resolution is executed in an attempt to ascertain a proof. The outcome of SL-resolution can either be a complete proof or a partial proof, contingent on the success of the resolution. This project introduces modifications to SL-resolution, including the integration of an occurs check, ancestor resolution, and new heuristics, as outlined in sections 3.2.3 to 3.2.5.
- Based on the outcome of the SL-resolution, the proposition, along with its corresponding proof or partial proof, is categorized into one of three groups: $Suffs$ (indicating propositions within the true set with successfully derived proofs), $Insuffs$ (representing propositions within the true set where proof derivation fails), and $Unwanted$ (covering propositions within the false sets and constraint axioms with successfully derived proofs).

The subsequent subsections provide an in-depth exploration of each of the implemented enhancements.

3.2.2 Theory Representation

Given that FOL includes functions, ABC_FOL needs to address how these functions are represented as inputs within the system and how they are evaluated during the resolution process.

To enhance user experience, functions are intuitively symbolized as $f(x_1,...,x_n)$, wherein f represents the function signature, and $x_1,...,x_n$ denote the function arguments. As a result, Example 1, which involves a faulty definition of naturals, is expressed within ABC_FOL as follows:

```
axiom([+naturals(2)]).
axiom([-naturals(\x), +naturals(suc(\x))]).

%Preferred Structure
trueSet([naturals(1)]).
falseSet([]).
```

Listing 3.1: Input theory file of Example 1

In this context, the function *suc* is formatted according to the above representation.

To ensure accurate proof generation, it is essential to address how functions are evaluated within this system. One approach is to solicit users to provide an algorithm, implemented as a separate Prolog function, for function evaluation. However, the drawback of employing user-provided algorithms for function evaluation lies in their unpredictability and potential inefficiency. As a solution, we suggest employing *equality axioms* for the symbolic evaluation of functions:

Definition 3.2.1 (Equality Axioms) *An equality axiom is a relation $x = y$ or $y = x$, where x, y are terms in FOL. x is a function term, consisting of one or more arguments, all of which does not contain variables. y is a constant term denoting the evaluation result of x.*

Equality axioms facilitate the symbolic evaluation of functions, streamlining the process by only requiring a symbolic match with one side of the equality axiom and substituting the term with the other side of the axiom. In practical terms, this is represented as a two-element list in Prolog, as illustrated in Listing 3.2.

Although functions in FOL are total, implying they must be validly defined for all inputs within the domain, the resolution process often requires definitions for only a select subset of inputs. By supplying equality axioms solely for a few relevant inputs, we could potentially expedite the resolution process. These limited inputs effectively serve as heuristics, preventing the program from traversing possibly infinite branches.

In the naturals example (example 1), the equality axioms could be defined as in listing 3.2, which allows the inference process of $naturals(3)$ as shown in figure 3.2.

```
eqAxiom([suc(1),2]).
eqAxiom([suc(2),3]).
```

Listing 3.2: Equality Axioms of Example 1

$$\begin{array}{ll}
\underline{naturals(3) \implies} & suc(2) = 3 \\
\underline{naturals(suc(2)) \implies} & naturals(X) \implies naturals(suc(X)) \\
\underline{naturals(2) \implies} & \implies naturals(2) \\
\implies &
\end{array}$$

Figure 3.2: The inference of $naturals(3)$

3.2.3 Ancestor Resolution

Ancestor resolution refers to the resolution of the current clause with one of its own *ancestors* [6]. In SL-resolution, *ancestors* refer to all the intermediate goal clauses derived through resolution steps (RS). The rationale behind incorporating ancestor resolution within FOL lies in its capacity to handle non-Horn clauses, which made derivation of new rules and new theorems possible during the resolution process. Such rules and theorems might be useful (and sometimes necessary) to correctly derive the desired goal.

Example 2 serves to illustrate a scenario wherein ancestor resolution becomes indispensable for establishing the desired goal. Figure 3.3 demonstrates the inference of $like(george, running)$:

Example 2: Sports Theory

$$\begin{aligned}
like(X, hiking) &\implies like(X, running). \\
like(X, running) &\implies like(X, hiking). \\
&\implies like(george, running) \lor like(george, running)
\end{aligned}$$

$$\begin{array}{ll}
\underline{like(george, running) \implies} & \implies like(george, running) \lor like(george, running) \\
\underline{\implies like(george, hiking)} & like(X, hiking) \implies like(X, running) \\
\underline{\implies like(george, running)} & like(george, running) \implies \\
\implies &
\end{array}$$

Figure 3.3: The inference of $like(gerorge, running)$

In the final RS depicted in Figure 3.3, the initial goal clause "$like(george, running) \implies$" is employed to resolve the ultimate subgoal. This exemplifies the integration of ancestor resolution within ABC_FOL.

The comprehensive algorithm for SL-resolution within ABC is documented in Xue's work [22], specifically in Table 7.1. By referring to Table 7.1 in [22], the only condition for a successful resolution of a goal without equality is as follows:

$$G_1 = -P(\vec{s}),$$
$$\exists(+P(\vec{t}) \vee -\vec{B}, S) \in \mathbb{T},$$
$$P(\vec{s}) \equiv P(\vec{t}) \wedge \mu,\ \mu = \{\vec{s}/\vec{t}\}$$

where G_1 denotes the selected literal to be resolved; P is a predicate; $\vec{s}$ and $\vec{t}$ are arguments, where each of s_i, t_i are terms; $-\vec{B}$ are a collection of negative literals; S is the source axiom (where $+P(\vec{t}) \vee -\vec{B}$ comes from); $P(\vec{s}) \equiv P(\vec{t})$ is the problem of unifying $P(\vec{s})$ and $P(\vec{t})$, μ is the substitution returned by the unification algorithm.

In this extension, for the incorporation of ancestor resolution, the conditions for successful resolution are divided into two cases:

Case 1:

$$G_1 = -P(\vec{s}),$$
$$\exists(+P(\vec{t}) \vee \vec{B}, S) \in \mathbb{T} \cup D_G,$$
$$P(\vec{s}) \equiv P(\vec{t}) \wedge \mu,\ \mu = \{\vec{s}/\vec{t}\}$$

Case 2:

$$G_1 = +P(\vec{s}),$$
$$\exists(-P(\vec{t}) \vee \vec{B}, S) \in \mathbb{T} \cup D_G,$$
$$P(\vec{s}) \equiv P(\vec{t}) \wedge \mu,\ \mu = \{\vec{s}/\vec{t}\}$$

where D_G denotes all the new goals in the previous derivation steps D.

The new condition exhibits two significant distinctions from the original version:

1. The resolution condition is partitioned into the disjunction of two potential scenarios. Case 1 addresses situations where the ongoing sub-goal, G_1, takes the form of a negative literal. Conversely, case 2 manages positive literals. This division is essential due to the presence of non-Horn clauses within FOL, which allows for the presence of positive literals as sub-goals.

2. The choice of the literal $\pm P(\vec{t})$ used for goal resolution, from the clause $(\pm P(\vec{t}) \vee \vec{B}, S)$, can now originate from the set of preceding derivations denoted as D_G. This collection, D_G, contains all the new goals derived from earlier resolution steps, except for those that have already been resolved away. For instance, in Figure 3.3, D_G comprises the clause $\{like(george, running) \implies\}$ after the initial resolution step, and two clauses after the subsequent step:
$\{(like(george, running) \implies), (\implies like(george, hiking))\}$.

3.2.4 Occurs Check

The occurs check is a safety condition to guarantee the soundness of inference. Its purpose is to prevent the creation of cyclic substitutions where a variable is bound to a term in which it itself appears, as exemplified by $f(Y)/Y$ [6].

In the previous ABC version, the occurs check was not implemented, as functions were absent. The issue of binding a variable to a term containing itself only arises when the term in question is a function, not a constant.

An illustrative example in which the occurs check would result in unification failure is presented below:

$$data(X,X) \equiv data(Y,model(Y))$$

In this scenario, one predicate asserts that the two arguments are identical, while the other asserts that the second argument is the model number of the first argument (hence a function of the first argument). During unification, the substitution Y/X is initially established when unifying the first argument. This leads to a situation where unifying the second argument becomes the challenge $Y \equiv model(Y)$, ultimately causing the occurs check to fail.

In this project, the occurs check is reintegrated into the unification algorithm. This corresponds to verifying the condition $x \in \mathcal{V}(s)$ within the *Occurs* case. The algorithm is outlined in Algorithm 1 in appendix A.

It's important to note that the occurs check's complexity can be substantial. Unifying with the occurs check entails a runtime bound of $O(size(t_1), size(t_2))$, where t_1 and t_2 denote the terms to be unified. In contrast, unification without the occurs check requires only $O(min(size(t_1), size(t_2)))$. Due to these complexities, the built-in unification algorithm of Prolog omits the occurs check [4]. Consequently, several heuristics have been devised to reduce the runtime, as described in the subsequent section.

3.2.5 Heuristics

Several heuristics have been developed to enhance the efficiency of the resolution process:

- **Loop Detection**: This heuristic already exists in the previous version of ABC. If all goals from a previous resolution step can be deduced using the most recent goal, the latest resolution step is pruned, as the goal has either become identical or more intricate. In ABC_FOL, this heuristic is retained to ensure that the resolution process avoids entering into circular loops.

- **Depth Limit**: The SL-resolution process in ABC employs a depth-first search strategy to find a valid proof. The depth limit determines the maximum depth of the search tree and halts the search branch if the current depth surpasses the set limit. This feature was present in ABC_Datalog and is maintained in ABC_FOL to guarantee the termination of the process.

- **Full Resolution**: Full resolution entails resolving more than one goal within the current goal clause [6]. By concurrently resolving multiple goals, the number of search branches is effectively reduced, increasing the likelihood of finding a solution within the depth limit. While absent in ABC_Datalog, this heuristic is introduced in ABC_FOL. Presently, additional sub-goals are only resolved if they are identical the selected sub-goal to prevent overlooking potential search branches. For instance, given the goal clause:

  $$-likes(X, orange), -likes(X, orange), -likes(X, apple)$$

  and an assertion $+like(toby, orange)$ used to resolve the first sub-goal, full resolution would also resolve the second identical sub-goal, resulting in $-likes(toby, apple)$ as the remaining goal. However, if the goal clause is:

  $$-likes(X, orange), -likes(Y, orange), -likes(X, apple)$$

  and the assertion $+like(toby, orange)$ is utilized to resolve the first sub-goal, the second sub-goal would not be resolved, as variable Y could instantiate to constants other than $toby$.

- **Tautology Detection**: Given that non-Horn clauses are permissible in FOL theories, it's possible to derive clauses containing complementary literals, which result in tautologies (clauses that invariably evaluate to true). Since deducing false from true is impossible, resolution steps resulting in tautologies are discarded. An example is illustrated in Figure 3.4.

$$\cfrac{\cfrac{p1(apple) \implies \qquad p2(Y) \wedge p3(X) \implies p3(Y) \vee p1(X)}{p2(Y) \wedge p3(apple) \implies p3(Y) \qquad \implies p2(apple)}}{p3(apple) \implies p3(apple)}$$

Figure 3.4: Example of discarded search branch due to tautology

3.3 Repair Generation

3.3.1 Overview

A flowchart illustrating the repair generation module's process is depicted in Figure 3.5. The following outlines each element:

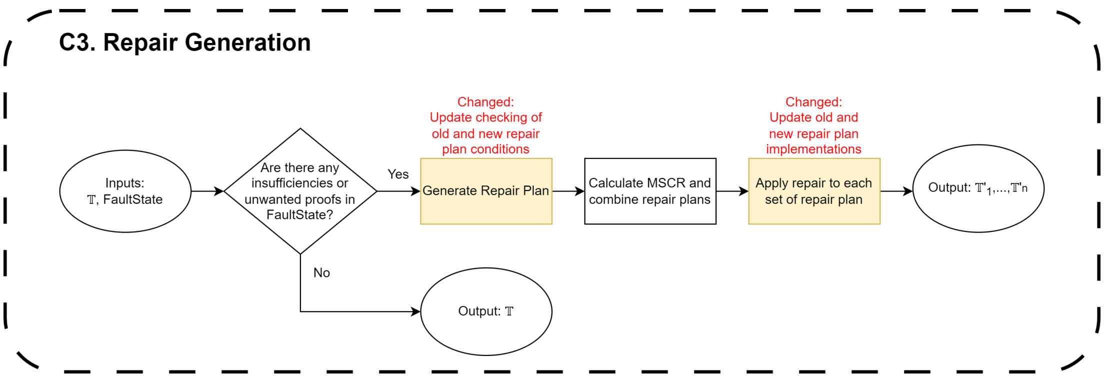
C3. Repair Generation
Inputs: 𝕋, FaultState
Are there any insufficiencies or unwanted proofs in FaultState?
Yes
No
Changed: Update checking of old and new repair plan conditions
Generate Repair Plan
Calculate MSCR and combine repair plans
Changed: Update old and new repair plan implementations
Apply repair to each set of repair plan
Output: 𝕋'1,...,𝕋'n
Output: 𝕋

Figure 3.5: Flowchart of Repair Generation Module (Changes implemented in this project are highlighted in red)

- The input consists of the Theory and the FaultState obtained from the fault detection module. Initially, the system checks for any insufficiencies or unwanted proofs. If the original theory is already fault-free, it is directly output.
- If faults necessitating repair are identified, the fault state is forwarded to the repair plan generation function. This function evaluates the conditions of each viable repair plan (as detailed later in this section) and assembles a list of potential repair plans that can be applied to the theory.
- Subsequently, the system computes the maximal set of commutative repair plans (MSCR) and clusters repair plans that can be concurrently executed (refer to Section 6.3 of [22]).
- For each group of repair plans, all repairs within the group are implemented on the theory $\mathbb{T}$, altering it to $\mathbb{T}'$. The updated theory is then verified prior to being output.

This project enhances the fault detection system by extending the existing repair plans to accommodate FOL and introducing novel repair plans tailored to FOL. A summarized overview of both legacy and new repair plans is provided in Table 3.1.

In this project, seven new repair plans, highlighted in red within Table 3.1, have been conceived and integrated into the ABC system. Among these, six are directed towards incompatibility repairs, while one addresses insufficiency. These novel repair plans are tailored to the distinct attributes of FOL not found in Datalog theories, including the incorporation of functions, the necessity of an occurs check, and the presence of non-Horn clauses.

These repair plans are strategically crafted to possess the self-inverse property. This property signifies that an incompatibility repair can be reversed through an insufficiency repair, and vice versa:

RS	Target match / mismatch		Repair Plan	Technique
$p_1(\overrightarrow{S}^n) \equiv p_2(\overrightarrow{T}^n)$ (Incompatability)	$p_1 = p_2$		CR1. Rename predicate on either side: $p_1(\overrightarrow{S}^n) \not\equiv p_2'(\overrightarrow{T}^n)$ or $p_1'(\overrightarrow{S}^n) \not\equiv p_2(\overrightarrow{T}^n)$.	Reformation
	$s_i \equiv t_i \equiv c$		CR2. Rename s_i or t_i to c'.	
	$(s_i = X \wedge t_i = c) \vee (s_i = c \wedge t_i = X)$		CR3. Weaken variable X to c'.	
			CR4. Add different constants: $p_1(\overrightarrow{S}^n, c_1) \not\equiv p_2(\overrightarrow{T}^n, c_2)$.	
			CR5. Delete the axiom A_u.	Belief Revision
			CR6. Add an unprovable precondition $q(\overrightarrow{Z}^w)$ to A_u.	Belief Revision (Variant)
	$s_i = f_1(\vec{x})$ $t_i = f_2(\vec{y})$ $s_i \equiv t_i$	$f_1 = f_2$	CR7. Rename predicate on either side: $f_1(\vec{x}) \not\equiv f_2'(\vec{y})$ or $f_1'(\vec{x}) \not\equiv f_2(\vec{y})$.	Reformation
		$x_i = y_i = c$	CR8. Rename x_i or y_i to c'.	
		$(x_i = X \wedge y_i = c) \vee$ $(x_i = c \wedge y_i = X)$	CR9. Weaken variable X to c'.	
			CR10. Add different constants: $f_1(\vec{x}, c_1) \not\equiv f_2(\vec{y}, c_2)$.	
	$(s_i = X \wedge t_i = k) \vee (s_i = k \wedge t_i = X)$		CR11. Add variable Y to k.	
			CR12. Add an unprovable alternative $q(\overrightarrow{Z}^w)$ to A_u.	Belief Revision (Variant)
$p_1(\overrightarrow{S}^n) \not\equiv p_2(\overrightarrow{T}^m)$ (Insufficiency)			SR1. Reform either $p_2(\overrightarrow{T}^m)$ or $p_1(\overrightarrow{S}^n)$ to the other.	Reformation
	$p_1 = p_2, s_i \neq t_i$		SR2. Extend t_i to variable Z.	
			SR3. Add the assertion $p_1(\overrightarrow{S}^n)$.	Abduction
			SR4. Add a rule which proves $p_1(\overrightarrow{S}^n)$.	
			SR5. Delete $p_1(\overrightarrow{S}^n)$ from original input axiom.	Abduction (Var.)
	$p_1 = p_2, (s_i = X \wedge t_i = k) \vee (s_i = k \wedge t_i = X)$		SR6. Remove all occurrences of variable Y in k.	Reformation

Table 3.1: A summary of repair plans: *Unification is denoted by $\equiv$, '=' denotes equality between terms or symbols, p_1 and p_2 are predicates, S and T are arguments of a predicate (constants, variables or functions), c is a constant, X is a variable, f_1 and f_2 are functions, x and y are arguments of a function (constants, variables or functions), k is either a constant or a function (but not variable). The left of the unification comes from the sub-goal, while the right comes from an input clause. Text highlighted in red denotes new repair plans, the background colour denotes sets of repair plans which are self-inverse.*

Repair Plans	Trace-back	Postive Sub-goals	Functions	Conditions or Heuristics
CR1	✓	✓		
CR2	✓	✓		
CR3	✓	✓		
CR4	✓	✓		
CR5	✓	✓		
CR6	✓	✓	✓	✓
SR1	✓	✓	✓	
SR2	✓	✓	✓	✓
SR3		✓	✓	✓
SR4		✓	✓	✓
SR5	✓	✓		✓

Table 3.2: A summary of extensions to old repair plans

- *(yellow)* CR1, CR2, CR4, CR7-CR10 encompass repairs that break unification by modifying one side of the unification to render them distinct. This can be undone by SR1, which reforms both sides of the unification to make them identical.
- *(green)* CR3 breaks unification by weakening a variable into a different constant. This can be reversed by SR2, which extends the constant back into a variable.
- *(blue)* CR5 deletes the axiom required to prove the incompatibility, which can be added back to the theory by SR3 and SR4.
- *(pink)* CR6 and CR12 introduce fresh literals (either positive or negative) to an axiom, which can be removed by SR5.
- *(orange)* CR11 breaks unification by introducing variables to induce occurs check failure. This can be undone by SR6 which resolves occurs check issues.

The self-inverse property illustrates that the application of repair plans retains much of the initial information. This attribute suggests that the repair plans are inherently *minimal* [9].

3.3.2 Extension of Preexisting Repair Plans

The extension of the old repair plans involves four key modifications: incorporating trace-back to literals, addressing positive literal sub-goals, accommodating functions, and adjusting conditions or heuristics tailored to each repair plan. A condensed overview is presented in Table 3.2.

Addressing positive sub-goals accounts for scenarios where the specific sub-goal under consideration for resolution within a resolution step is a positive literal. This is possible due to the inclusion of non-Horn clauses. The following sections describe remaining repairs in greater detail.

3.3.2.1 Trace-back algorithm

The trace-back algorithm serves the purpose of locating the original axiom within the theory that introduced the input clause in the current RS. This algorithm proves essential due to ancestor resolution, which can lead to the input clause not being an axiom directly from the theory. Thus, the trace-back process is indispensable for ensuring accurate propagation of changes to the appropriate axiom in the theory.

The algorithm identifies the original axiom by retracing the derivation steps and identifying an RS in which the new goal is identical to the input clause. Once this RS is located, the input clause of that RS is returned if it originates from the theory. Otherwise, the algorithm is recursively invoked on the new input clause. A comprehensive depiction of the algorithm is realized as `traceBackClause`, illustrated in Listing B.1 within Appendix B.

3.3.2.2 Functions

As explained in section 3.2.2, functions are portrayed in a format closely resembling that of predicates. However, within ABC's internal processing, this representation is transformed into a list structure. For instance, the function $f(apple, g(orange))$ is encoded as `[f,[apple],[g,[orange]]]` in the internal framework.

Given that functions can be nested to arbitrary depths, numerous repair plans necessitate corresponding adjustments to accommodate this aspect. Particularly noteworthy is the implementation of the `memberNested` function, designed to accurately identify constants and variables within a predicate. This function replaces multiple invocations of the original `member` function in the codebase. The precise implementation of `memberNested` is furnished in Listing B.2, presented within Appendix B.

3.3.2.3 Changes in specific repair plans

For comprehensive details and examples of each preexisting repair plan, we direct readers to section 5.1 of [22]. The following section exclusively outlines the introduced modifications and omits the entire operational theory to prevent redundancy.

CR6: Break $p(\vec{S}^n) \equiv p(\vec{T}^n)$ by adding an unprovable precondition $q(\vec{Z}^w)$.

Due to the incorporation of functions, it is important to note that not only constants but also functions can substitute a variable X. This necessitates an extension of the definition of the set $\mathbb{SC}(R, X)$ as outlined in equation (5.12) of [22]. Now, the set includes

both functions and constants that replace variable X in R. While the equation (5.12) remains unchanged in defining $\mathbb{SC}(R,X)$, the symbol c_{xi} can represent both functions and constants. For simplicity, the function arguments are disregarded, and solely the function name is considered for each c_{xi}. The same principle is applied to the definition of the domain $\mathcal{D}(q,i,\mathbb{T})$ outlined in equation (5.10).
The above adaptations result in minimal adjustments for equation (5.13), which pertains to the candidates for the precondition argument. A minor alteration is required in cases where v_i is set as $f(\vec{x})$, and f is a function ($f \in \mathcal{D}(q,i,\mathbb{T})$). Specifically:
For each argument x_i:

- If there is another call to the function f within the rule R, and its i-th argument is a variable ϕ, set $x_i := \phi$.
- Otherwise, if there is another call to f within the rule R, and its i-th argument is a function call $g(\vec{y})$, set $x_i := g(\vec{y})$.
- Otherwise, if there is another call to f within the rule R, and its i-th argument is a constant c_i, set $x_i := c_i$.
- Otherwise, set $x_i = \gamma$ where γ is a new independent variable.

Given that functions possess a broader capacity for generalization compared to constants, the preference is towards initializing with functions over constants.

SR2: Fix the failed RS $s_i \neq t_i$ by extending either to variable Z.
In ABC_Datalog, when t_i was selected for extension, a stipulation was applied that $p_2(\overrightarrow{T}^n)$ had to originate from a rule. This measure was implemented to prevent the formation of axioms containing *orphan variables* — variables solely present in the head but absent in the body. However, in the context of FOL, no constraints exist regarding orphan variables. Consequently, this restriction is eased, enabling the repair process to accommodate theorems in addition to rules.

SR3: Fix the failed RS $p_1(\vec{S^n}) \not\equiv p_2(\vec{T^m})$ by adding the assertion: $p_1(\vec{S^n})$.
In ABC_Datalog, a constraint was imposed to ensure that the added assertion should not involve a variable, primarily to prevent the emergence of orphan variables. This is now relaxed as it is fine for an FOL theory to contain orphan variables. Moreover, it is now possible to add a constraint axiom in this repair, as the sub-goal $p_1(\vec{S^n})$ can now be a positive literal.

SR4: Fix the failed RS $p_1(\vec{S^n}) \not\equiv p_2(\vec{T^m})$ by adding the rule shown in (5.26 [22]).

Given that $p_1(\vec{s}^n)$ can be a positive literal, this repair strategy is extended to accurately handle positive goals. This entails generating a rule, as outlined in equation (5.26), albeit with the head and body of the rule reversed.
Additionally, a heuristic that was formerly integrated into this repair plan mandates that the candidate for the precondition of the rule must share a constant with the goal clause (5.24). However, this requirement is now eased to permit an axiom containing only variables to also qualify as a valid candidate. This adjustment is exemplified by the following scenario:

Example 3.1: Humans Theory

$$\Longrightarrow human(X)$$
$$\mathcal{T}(\mathbb{PS}) = \{breathe(peter)\}$$

Example 3.2: Repair with SR4

$$human(Y) \Longrightarrow breathe(Y)$$
$$\Longrightarrow human(X)$$

In this case, the initial assertion posits that all entities are human — an assumption that aligns well with a knowledge base centered around human subjects. The objective is to establish the proposition $breathe(peter)$. However, there are no axioms within the theory exclusively pertaining to $peter$. Nonetheless, the theorem $human(X)$ is valid within the theory and can be unified with $human(peter)$. Consequently, it is reasonable to introduce the rule $human(Y) \Longrightarrow breathe(Y)$ in order to prove the required proposition.

SR5: Fix the failed RS $p_1(\vec{S}^n) \not\equiv p_2(\vec{T}^m)$ by deleting the precondition $p1(\vec{S}^n)$.
Similarly, it is possible to delete the literal $p_1(\vec{S}^n)$ in the body of a rule as it might be a positive literal.

3.3.3 New Repair Plans

CR7-10: Reuse CR1-4 to break unification of a function: $f_1(\vec{x}) \not\equiv f_2(\vec{y})$
CR7 - CR10 employ the same methodology as CR1 - CR4 to break unification. However, these plans target the unification between two functions rather than an entire predicate. Since predicates and functions share similar structures, the techniques that operate at the predicate level can be seamlessly adapted to function-level unification. Consequently, breaking the unification between functions will inherently break the unification between the predicates linked to those functions, effectively rectifying the incompatibility.

It is important to note that while functions and predicates share structural similarities, functions have the capacity to be nested to varying depths. This project's repair plans are adeptly devised to address unification between functions at any level within nested functions. This design consideration ensures the versatility of the repair plans when handling functions with diverse nesting depths.

Example 4.1: Faulty Dad Theory

$$\begin{aligned} dad(husOf(X),Y) \wedge dad(husOf(Z),Y) &\implies X = Z && (R1)\\ &\implies dad(husOf(lily),tina) && (A1)\\ &\implies dad(husOf(lily),victor) && (A2)\\ &\implies dad(husOf(anna),victor) && (A3) \end{aligned}$$

$\mathcal{F}(\mathbb{PS}) = \{anna = lily, lily = anna\}$

Consider the ”faulty dad theory,” which emulates the motherhood theory detailed in [22]. Suppose that the knowledge base lacks information about the names of all fathers, resulting in the representation of ”dad” through the function $husOf$ (indicating ”husband of”) in relation to mothers. The false set remains the same with the motherhood theory.

With this context, CR7 - CR10 now possess the capability to break the unification of the function $husOf$. This repair can occur, for instance, between the entities R1 and A2. Example 4.2 demonstrates a possible repair outcome generated by CR7. In this scenario, the placeholder term $dummyhusOf1$ could signify ”previous husband”, while the original $husOf$ represents the concept of the ”current husband”.

Example 4.2: Repaired Dad Theory

$$\begin{aligned} dad(husOf(X),Y) \wedge dad(husOf(Z),Y) &\implies X = Z\\ &\implies dad(husOf(lily),tina)\\ &\implies dad(dummyhusOf1(lily),victor)\\ &\implies dad(husOf(anna),victor) \end{aligned}$$

$\mathcal{F}(\mathbb{PS}) = \{anna = lily, lily = anna\}$

CR11: Break $p_1(\vec{S}^n) \equiv p(\vec{T}^n)$ by adding a variable Y to k

The condition of this repair is $(s_i = X \wedge t_i = k) \vee (s_i = k \wedge t_i = X)$, where k is either a constant or a function. Additionally, a prerequisite for this repair is the utilization of $Var\ Elim$ (as outlined in table 2.1), which unifies X with a variable Y within the same unification problem. This criterion aligns with the ”blocked doomed occurs-check actuations” heuristic [9]. Subsequently, the variable Y is added into k.

The repair is applied on the side where k is present. To implement this on the original axiom, trace-back is necessary. If k is a constant, it transforms into a function with a single argument Y, denoted as $k'(Y)$. The naming convention for k' involves appending $dummy \oplus k \oplus S$, with S denoting the serial number.

If k is originally a function, its arity is adjusted by appending Y as the final argument. To circumvent function overloading, any other instances of k are modified by introducing a new, independent variable as the last argument.

Example 5.1 Faulty equlity Theory

$$y = C \implies \qquad (C1)$$

$$\implies x = x \vee world(unstable) \quad (T1)$$

$$\mathcal{F}(\mathbb{PS}) = \{world(unstable)\}$$

This excerpt is cited from [9] to serve as an illustrative instance of the repair process. The initial axiom, denoted as C1, embodies a flawed assertion that can be translated to $\exists z, \forall y. y \neq z$. In this context, C represents a constant derived from the process of skolemization. The predicate $world(unstable)$ can be intrepreted as a discrepancy between the theory and real-world observations, provided it can be proven. The targeted unification to break in this example is that between $y = C$ from C1 and $x = x$ in T1. The repaired example turns C into a function as shown in example 5.2.

Example 5.2 Repaired equlity Theory

$$y = dummyC1(y) \implies \qquad (C1)$$

$$\implies x = x \vee world(unstable) \quad (T1)$$

$$\mathcal{F}(\mathbb{PS}) = \{world(unstable)\}$$

In the repaired version, C1 is amended to reorder the quantifiers into $\forall y, \exists z. y \neq z$, which now holds true.

CR12: Break $p_1(\vec{S}^n) \equiv p(\vec{T}^n)$ by adding an unprovable alternative $q(\vec{Z}^w)$

In contrast to CR6, which introduces a negative literal, this repair plan involves appending a positive literal. The selection of the candidate for $q(\vec{Z}^w)$ must also adhere to postulate 3 (P.78 in [22]), ensuring that it both blocks the necessary repair and does not introduce new insufficiencies. All the prerequisites for $q(\vec{Z}^w)$ remain consistent with those outlined in CR6, although the polarity of positive and negative literals is reversed.

The subsequent example serves to show a scenario in which the addition of a positive literal proves to be a more logical choice compared to introducing a negative literal:

Example 6.1 Repaired obesity Theory

$$eat(X, icecream) \implies obese(X) \quad (R1)$$
$$\implies eat(ernest, icecream) \quad (T1)$$
$$\mathcal{F}(\mathbb{PS}) = \{obese(ernest)\}$$

To repair this theory, an alternative can be added to the rule R1, as shown:

Example 6.2 Repaired obesity Theory

$$eat(X, icecream) \implies obese(X) \lor \color{red}{dummyPred(X)} \quad (R1)$$
$$\implies eat(ernest, icecream) \quad (T1)$$

The predicate $dummyPred(X)$ can be further repaired into a more contextually relevant concept, such as $happy(X)$. This adjustment implies that eating ice cream might lead to obesity, but it could also yield happiness. A possible interpretation of this new rule would be that: only if a person eats ice cream without feeling happiness can we ascertain that the person is obese (as they are too addicted into ice-cream, that no amount of ice-cream is enough to bring them happiness).

SR6: Fix the failed RS $s_i \neq t_i$ by removing all occurrences of a variable X

This repair operates similar to SR2 on a single-argument level. Its condition is the failure of unification solely due to the occurs check, with $(s_i = X \land t_i = k) \lor (s_i = k \land t_i = X)$, and the absence of other mismatches between the two literals: $p_1 = p_2$ and $s_j \equiv t_j$ for all $j \neq i$. This also hinges on the existence of a successful *Var Elim* operation that replaces X with another variable Y.

Without loss of generality, assume it is the case that $(s_i = X \land t_i = k)$. The algorithm for this repair is as follows:

- Locate the substitution Y/X that replaces variable X with Y within the argument list $\vec{T}$. If this substitution is not present, the repair cannot be applied.
- Remove all occurrence of Y from the function t_i. At the same time, record the removed argument positions and delete all those arguments from all other occurrences of the same function.

The flawed self-love theory is cited from [9] to illustrate the application of this repair, as shown in example 6.1. The theorem $T1$ states the proposition: ”everyone loves someone.” The intention is to utilize this proposition to demonstrate that someone loves themselves (R2). However, this proof is unattainable in the original theory.

Example 7.1 Faulty self-love Theory

$$\begin{aligned} &\Longrightarrow loves(y, loveOf(y)) \quad (T1) \\ loves(x,x) &\Longrightarrow world(stable) \quad (R1) \end{aligned}$$

$$\mathcal{T}(\mathbb{PS}) = \{world(stable)\}$$

To address this issue, a solution involves removing the variable y from the function $loveOf$, which was the root cause of the unsuccessful occurs check. The repaired theory will be as follows:

Example 7.2 Repaired self-love Theory

$$\begin{aligned} &\Longrightarrow loves(y, loveOf) \quad (T1) \\ loves(x,x) &\Longrightarrow world(stable) \quad (R1) \end{aligned}$$

The revised theory T1 transforms the function $loveOf$ into a constant. Consequently, T1 now implies that there exists an entity (represented by the constant $loveOf$, which could subsequently undergo further refinement) that is loved by everyone, fixing the insufficiency.

3.3.4 Heuristics

Readers are referred to section 6.5 of the original thesis of ABC [22] for a comprehensive list of heuristics already present in ABC. This section details the new heuristics in the extended ABC system:

- **datalogOnly**: This heuristic restricts the ABC system to run only ABC_Datalog, which is useful if a theory can be expressed entirely in Datalog logic.
- **noPruneSubopt**: With this heuristic, the system refrains from pruning suboptimal repairs. Although some faults might be fixable with a single repair step, the most optimal repair, which offers a more accurate representation of the environment, could require multiple steps. By not discarding suboptimal repairs, the system can explore repairs that involve more than one iteration. However, it is essential to set a round limit and incorporate other heuristics to prevent potential issues such as stack overflow or infinite execution.

Chapter 4

Evaluation and Analysis

4.1 Evaluation Methodology

4.1.1 Data Source

The data utilized in this section includes theory examples obtained from various sources:

1. **Self-curated examples**: These examples were specifically generated for this project, drawing upon the author and supervisor's personal experiences and insights.
2. **Examples from ABC**: These are the examples originally used in the evaluation of ABC theories. These examples are reused to compare against the old ABC system.
3. **Online dataset**: The FOLIO dataset [17] is an expert-annotated dataset consisting of theories expressed in both natural language and FOL, a conclusion and the truth value of the conclusion. Some of the examples from the dataset are rewritten into ABC's input format and used to evaluate the system, as detailed in appendix C. The chosen examples contains existential quantifiers, functions, negations, non-Horn clauses etc. to demonstrate the unique properties of FOL.

4.1.2 Test Plans

The following four tests will be conducted with regard to ABC_FOL.

- **Correctness test**: This test focuses on the extended fault detection module's performance, evaluating its accuracy in identifying faults within theories expressed in both Datalog and FOL. For this test, examples from ABC, along with specific instances from the FOLIO dataset, will be employed.
- **Superior Test**: This evaluation determines whether ABC_FOL can generate accurate and reasonable repairs for FOL theories that cannot be formulated in Datalog logic. The test will involve selected examples from the FOLIO dataset.

Theory Name	Sufficiency	Insufficiency	Incompatibility	Violations
Families	1/1	1/1	-	-
Tweety	3/3	-	1/1	-
Married Woman	-	1/1	1/1	-
Researcher	1/1	-	1/1	-
Super Penguin	1/1	-	1/1	1/1
Buy Stock	1/1	-	-	-
Working Student	2/2	-	-	1/1
Parent	-	3/3	-	-
Missing Parent	2/2	2/2	-	-
Load Car	-	2/2	-	-
Talent Show	-	1/1	-	-
Monkey Pox	-	2/2	1/1	-
Video Games	1/1	2/2	-	-
Turkey Types	-	1/1	1/1	-
Novel Writer	-	1/1	1/1	-

Table 4.1: Correctness test of fault detection module: *Each entry is of the form x/y where x denotes the number of correctly classified proposition and y denotes the total number of propositions in that category. Red theories denote new FOL examples as cited from [17] (More details for the FOL examples can be found in appendix C).*

- **Running Time Test**: This test measures the execution time of ABC_FOL, assessing its ability to conclude within a reasonable timeframe. Its performance will be compared to ABC_Datalog. Both prior ABC examples and the FOLIO dataset will be used in this evaluation.

- **Case Study**: This case study delves into an intriguing example referred to as the ”*Eggtimer*”. Its objective is to highlight differences between ABC_Datalog and ABC_FOL in terms of generated repair quantities, repair quality, the influence of heuristics on repairs, and any inherent limitations of the extended system.

4.2 Correctness test

The purpose of the correctness test is to ascertain the fault detection module’s precision in identifying all faults correctly. To carry out this evaluation, a combination of Datalog ABC examples and carefully chosen FOL examples from the FOLIO dataset are utilized. The definitive truth values for the Datalog ABC instances are extracted from [22], while the FOLIO dataset are already expert-annotated with their corresponding truth values. The outcomes of this examination are presented in Table 4.1.

The results demonstrate that the extended fault detection module retains backward compatibility, accurately identifying all faults in the Datalog ABC instances. Furthermore, it effectively extends to FOL examples.

4.3 Superior Test

The superior test evaluates ABC_FOL's performance on theories exclusively expressed in FOL. This assessment is conducted from two perspectives: firstly, by determining if a valid repair can be identified, and secondly, by checking if the repair aligns with the concept of a "*Gold Standard*."

Definition 4.3.1 (Gold Standard) *A gold standard is a repaired theory with the following properties: 1. They satisfy the preferred structure, 2. All the applied repair operations are necessary to repair the faults, 3. They embrace commonsense meanings.*

While there may exist multiple potential gold standards for a flawed theory, one specific gold standard is defined for each FOL example, detailed in Appendix C. They are unfortunately subjective in nature, and efforts to increase its robustness are described in the future work section. Nevertheless, the five FOL examples aim to demonstrate ABC's versatility to conceive FOL theories in different aspects:

- Example 1 (Talent show) incorporates non-Horn clauses and equivalence relations, which can lead to looping rules. This assesses ABC's capability to appropriately manage such scenarios.
- Example 2 (Monkey Pox) and 3 (Video Games) involves existential quantifiers resulting in skolem constants with undefined meanings. ABC's aptitude to assign meaningful names to these constants is evaluated.
- Example 4 (Turkey Types) introduces non-Horn clauses and multiple constraint axioms to eliminate implausible cases. This extensively tests ABC's ability to detect faults and generate repairs for positive literals.
- Example 5 (Novel Writer) introduces functions derived from skolemization and equality axioms for function evaluation. ABC's proficiency in accurately handling and repairing faults related to functions is assessed.

The outcomes of the superior test are presented in Table 4.2. Findings from table 4.2 reveal the successful generation of valid repairs for all the examined theories. Among the five theories, only one does not incorporate the gold standard within the solution. This shows that ABC_FOL is able to analyze and repair a wide range of constructs within FOL theories and often generate good quality repairs.

Theory Name	Number of repairs	Gold Standard Included?
Talent Show	6	Y
Monkey Pox	5	Y
Video Games	14	Y
Turkey Types	4	N
Novel Writer	36	Y

Table 4.2: Superior Test

4.4 Running Time Test

The running time test assesses the duration required for the system to generate repairs for flawed theories. In this evaluation, the execution time (cf. E-time in [22]) is employed as a metric to quantify the ABC system's runtime. The execution time records the interval from when the ABC system receives the input (initiation of the `abc` predicate) until the point when all fully repaired theories are produced. Given that the impact of theory size, fault count, and heuristics on ABC's performance have been extensively investigated and discussed in [22], this study primarily concentrates on comparing the runtime performance between ABC_Datalog and ABC_FOL. Additionally, the test aims to ascertain whether FOL theories exhibit significantly longer processing times compared to Datalog theories.

For each theory, the running time is calculated based on an average of three trials conducted on a single-thread computer. The results are compiled in Table 4.3. It is important to note that all the data is derived from the same computer configuration, without the application of any heuristics, and with consistent round and cost limits to ensure equitable comparisons.

From table 4.3, it becomes evident that ABC_FOL generally exhibits lengthier processing times than ABC_Datalog. This is understandable, as ABC_FOL contains additional procedures within the fault detection module, and more repair choices in the repair generation module. The most substantial increase in running time is about 17 times of the original system, with an average rises of 5-6 times, which still falls within an acceptable range.

It can also be concluded that FOL theories in general takes more time for processing than Datalog theories. This discrepancy is possibly attributed to the extended time required for retracing axioms and conducting function evaluations. It is worth noting that the ”Novel Writer” examples demonstrates notably extended running times compared to remaining theories, reaching the maximum round and cost limits. This could be due

Theory Name	Old running time (ms)	New running time (ms)
Families	36	42
Tweety	29	510
Married Woman	37	65
Researcher	8	33
Super Penguin	73	288
Buy Stock	62	367
Working Student	55	260
Parent	203	133
Missing Parent	519	3329
Load Car	150	487
Talent Show	-	238
Monkey Pox	-	36
Video Games	-	9404
Turkey Types	-	2113
Novel Writer	-	98214

Table 4.3: Running Time Test: *Old running time refer to the Datalog ABC system while New running time refers to FOL ABC system. All theories are tested with round limit set to 8 and cost limit set to 100.*

to the presence of functions within the theory, which induce an almost boundless search depth for a proof. This hints that supplementary heuristics might need to be developed to further limit the running time, while still ensuring the identification of proofs, if they exist.

4.5 Case Study

4.5.1 Problem Statement

The ”*eggtimer*” example, as introduced by Bundy [7], serves as a case study for ABC_FOL. This problem stands as one of the primary motivations behind the development of ABC_FOL. The problem is articulated as follows:

Euler’s formula posits that within a polyhedron, the equation $V - E + F = 2$ holds true, where V represents the count of vertices, E signifies the count of edges, and F the count of faces. However, this principle exclusively applies to convex polyhedra and does not universally include all types of polyhedra. To address this limitation, Lakatos [20] introduced a concept termed ”*monstar barring*”, involving the revision of object definitions to eliminate counterexamples. Specifically, the definition of a ”*Polygon*” can be altered to exclude non-convex polygons. The original definition of a polygon is presented as follows:

$$(\forall v \in vs. \exists! l_1, l_2 \in ls. l_1 \neq l_2 \wedge meet(v, l_1, l_2)) \implies polygon(n, ls, vs) \tag{4.1}$$

where $meet(p, l_1, l_2)$ means that lines l_1 and l_2 meet at a point p, vs is the set of vertices and ls is the set of lines.

The *eggtimer*, which is the target polygon to block, can then be formalized by the set $ls = \{l_1, l_2, l_3, l_4\}$, $vs = \{v_1, v_2, v_3, v_4\}$, and the meeting points $meet(v_1, l_1, l_2)$, $meet(v_2, l_2, l_3)$, $meet(v_3, l_3, l_4)$, $meet(v_4, l_4, l_1)$, $meet(x, l_1, l_3)$. A graphical illustration is provided in figure D.1 in appendix D.

With the current definition, a proof can be established that the "*eggtimer*" qualifies as a polygon. To obstruct such a proof, Bundy proposed the introduction of a precondition to equation 4.1, which is presented as equation 4.2:

$$\forall p.\forall l_1, l_2 \in ls.(meet(p, l_1, l_2) \implies p \in vs) \tag{4.2}$$

Equation 4.2 mandates that any meeting point must also serve as a vertex of the polygon. However, the "*eggtimer*" example breaks this precondition, given that point x functions as a meeting point while not being an element of vs. Yet, addressing this repair remains a challenge for ABC_FOL due to the subsequent factors:

- Equation 4.1 cannot be faithfully represented within the system, as $\exists!$ (uniquely exists) stands as a non-standard FOL construct.
- When negated and converted to clausal form, the precondition outlined in equation 4.2 becomes a conjunction of two literals: $meet(p, l_1, l_2) \wedge \neg vs(p)$. Integrating this with equation 4.1 directly would disrupt the clausal form, necessitating the division of equation 4.1 into two separate axioms after the re-conversion to standard clausal form. The current repair mechanisms within ABC do not facilitate such an operation.

That said, two attempts in solving the problem under current ABC's framework is shown in subsequent sections.

4.5.2 Attempt 1

In the first approach, the objective is not to transform equation 4.1 into clausal form. Instead, leveraging the available information, the formulation of the "eggtimer" in ABC is as follows (the complete Prolog formulation can be found in D.2):

$$meet(v_1, l_1, l_2) \wedge meet(v_2, l_2, l_3) \wedge meet(v_3, l_3, l_4) \wedge meet(v_4, l_4, l_1)$$
$$\implies polygon(eggtimer, setOf(v_1, v_2, v_3, v_4), setOf(l_1, l_2, l_3, l_4))]).$$

This formulation considers all meeting points related to the vertex set vs as the precondition for a polygon. Under this approach, the ability to prove that the ”*eggtimer*” is a polygon remains, with the only distinction from equation 4.1 being the explicit specification of all meet points.

When this theory is inputted into the ABC_FOL system without heuristics, a total of 50 distinct repairs are generated. Many of these repairs modify the predicate *meet* or alter constant names, which is undesirable since the aim is to avoid altering the ”*eggtimer*” definition. While plausible repairs do exist, the substantial number of possibilities reduces the likelihood of selecting and adopting the most desirable repairs. This emphasizes the necessity for heuristics, such as protecting the predicate *meet*, to curtail the search space and reduce the space of potential repairs.

The following repair, as generated by the system, is a viable option for adoption:

```
[+dummy_polygon_1(eggtimer,set_of(\v1,\v2,\v3,\v4),set_of(\l1,\l2,\
    l3,\l4)),-meet(\v1,\l1,\l2),-meet(\v2,\l2,\l3),-meet(\v3,\l3,\l4)
    ,-meet(\v4,\l4,\l1)].
```

Listing 4.1: Eggtimer repair formulation 1

In this repair, the predicate *polygon* is modified to *dummy_polygon_1* through the CR1 repair operation. The introduction of the *dummy_polygon_1* predicate could be interpreted to represent a generic polygon, while the original *polygon* predicate is retained to signify a convex polygon. This alteration allows for a more refined distinction between the types of polygons.

4.5.3 Attempt 2

Appendix D.3 shows the complete Prolog formuation of this approach. Notably, equation 4.1 is transformed into clausal form (excluding the uniqueness constraint and treating it as a conventional existential quantifier), as depicted in the following:

$$\neg vs(x) \vee \neg ls(A) \vee \neg ls(B) \vee eq(A,B) \vee \neg meet(x,A,B) \vee polygon(eggtimer)$$

The constant x within the equation originates from the skolem constant, designated as x to facilitate the repair of this example. The desired precondition (equation 4.2) is also introduced into the theory as a separate axiom, thus assuming to be true. Consequently, a fault emerges wherein x is presumed to be a point within the vertex set vs. This theory anticipates ABC's capability to challenge equation 4.2 (thus exposing its non-validity) and subsequently repair the fault.

A total of 56 repairs are generated, with many sensible repairs such as renaming *polygon* to *dummy_polygon_1* as in attempt 1. The desired repair is also successfully identified:

```
[+vs(\p),-dummyPred(\p),-meet(\p,\l1,\l2)].
```

Listing 4.2: Repair of Eggtimer formulation 2

This repair employs CR6 to introduce an unprovable precondition into equation 4.2, effectively disrupting the proof process. Nevertheless, the newly introduced predicate *dummyPred* lacks the essential semantics required for the rule to hold significance. In this context, *dummyPred* should be interlinked with specific mathematical definitions that establish point p as a vertex.

4.5.4 Analysis

The eggtimer example underscores several limitations of ABC_FOL. Namely:

- **Representation Challenges**: Complex FOL theories featuring distinctive constructs like uniqueness conditions struggle to be faithfully represented. Additionally, the proper naming of constants stemming from skolemization requires manual adjustment or repair.
- **Limited Repair Operations**: The current repertoire of repair operations primarily supports straightforward and minimal alterations, while more intricate modifications like the one presented in this case study remain unsupported.
- **Amplified Search Space**: The enlarged space of potential repairs demands a precise choice of heuristics and protected items to reduce the number of undesirable repairs.

Nonetheless, ABC_FOL contributes by enabling two distinct representations of the eggtimer example as valid inputs, a capability absent in ABC_Datalog due to the presence of functions and non-Horn clauses. While the generated repairs might fall short of optimality compared to the best repair, the system still effectively identifies a valid repair within a reasonable timeframe, offering a sound interpretation of the eggtimer scenario.

Chapter 5

Conclusions

5.1 Project Limitations

Several limitations were encountered during the course of this project:

1. **Insufficient Documentation**: The absence of comprehensive documentation for ABC_Datalog complicated code implementations. A significant portion of the project's duration was dedicated to reading and comprehending the codebase.
2. **Limited Literature**: The scarcity of literature on automated theory repair techniques posed challenges, particularly in devising novel repair strategies.
3. **Data Availability**: The absence of a sizable dataset containing FOL theorems in clausal form hindered the creation of numerous FOL examples for ABC testing. Manual conversion of examples to clausal form presented a formidable obstacle within the project's constrained timeframe.
4. **Evaluation Constraints**: The assessment of ABC_FOL's performance is restricted due to the subjective nature of determining the optimal repair output. Moreover, human subjects could not be involved in evaluation due to time constraints.

5.2 Future Work

Considering the evaluation results and the project's limitations highlighted earlier, the following avenues for future work are proposed:

1. **Dealing with Equalities**: Extend the unique name assumption with exceptions (UNAE) from ABC_Datalog to ABC_FOL. This enhancement would enable a more efficient representation of equality sets within the FOL context.

2. **Allow Non-Ground Assertions in $\mathbb{PS}$**: Relax the current constraint on $\mathbb{PS}$, allowing it to incorporate non-ground assertions. This expanded flexibility would accommodate more complex and expressive statements, such as those involving quantifiers like ”for all” or ”there exists,” as potential repair targets.
3. **Tailored Heuristics for FOL Theories**: Develop heuristics specifically designed for FOL theories. Given the increased search space in both fault detection and repair generation, custom heuristics could enhance the efficiency of ABC_FOL. For example, heuristics could be designed to restrict search within nested functions.
4. **Working with sorted logic**: Explore the extension of ABC_FOL to a sorted logic. Sorted logic introduces a concept similar to *types* in programming languages [27], in which quantifiers can bound variables over different domains. Employing sorted logic could potentially guide proof search more effectively and enhance the system's performance.
5. **Detailed Evaluation of ABC_FOL**: Conduct a more comprehensive evaluation of ABC_FOL. This could involve replicating tests from ABC_Datalog or introducing an expanded set of theory examples to assess and demonstrate the system's capabilities more comprehensively.

5.3 Summary

This thesis has proposed the design of algorithms and repair plans for extending ABC to first-order logic. The proposed modifications expand the system by introducing ancestor resolution and an occurs check to the fault detection module. Additionally, novel repair plans have been developed to address the unique characteristics of FOL, including functions and non-Horn clauses.

The extension was successfully implemented within the ABC codebase, as documented in the provided repository [14]. The evaluation of ABC_FOL provides supporting evidence for the research hypothesis. The system effectively generates numerous potential repairs that rectify identified faults, while maintaining semantic coherence. It is anticipated that ABC_FOL holds applicability across a range of domains, including decision systems, law enforcement, and knowledge graphs. ABC_FOL is expected to have a broader spectrum of use cases compared to ABC_Datalog.

Bibliography

[1] Chapter xii - automatic deduction. In Paul R. Cohen and Edward A. Feigenbaum, editors, *The Handbook of Artificial Intelligence*, pages 75–123. Butterworth-Heinemann, 1982.

[2] Carlos E. Alchourrón and David Makinson. On the logic of theory change: Safe contraction. *Studia Logica*, 44(4):405–422, 1985.

[3] Stephen Bay, Daniel Shapiro, and Pat Langley. Revising engineering models: Combining computational discovery with knowledge. 08 2002.

[4] Joachim Beer. The occur-check problem revisited. *The Journal of Logic Programming*, 5(3):243–261, 1988.

[5] Holden S.B. Paulson L.C. Bridge, J.P. Machine learning for first-order theorem proving. In *J Autom Reasoning 53, 141–172*, 2014.

[6] Alan Bundy. *The Computer Modelling of Mathematical Reasoning*. Academic Press Professional, Inc., USA, 1985.

[7] Alan Bundy. An ambiguous polygon. Blue Book Note 1887, July 2023.

[8] Alan Bundy and Xue Li. Representational change is integral to reasoning. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, January 2023.

[9] Alan Bundy and Boris Mitrovic. Reformation: A domain-independent algorithm for theory repair. Workingpaper, February 2016.

[10] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[11] P. T. Cox and T. Pietrzykowski. Causes for events: Their computation and applications. In Jörg H. Siekmann, editor, *8th International Conference on Automated Deduction*, pages 608–621, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.

[12] Marc Denecker and Antonis Kakas. *Abduction in Logic Programming*, pages 402–436. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[13] Jianfeng Du, Kunxun Qi, and Yuming Shen. Knowledge graph embedding with logical consistency. In Maosong Sun, Ting Liu, Xiaojie Wang, Zhiyuan Liu, and Yang Liu, editors, *Chinese Computational Linguistics and Natural Language Processing Based on Naturally Annotated Big Data*, pages 123–135, Cham, 2018. Springer International Publishing.

[14] Thomas Wong (forked from Xue Li). Abc_fol. https://github.com/tpmmthomas/ABC_FOL, 2023.

[15] Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*, chapter 9: SLD-Resolution and Logic Programming. Wiley, 1986.

[16] Peter Gärdenfors. *Belief Revision.* 05 1992.

[17] Simeng Han, Hailey Schoelkopf, Yilun Zhao, Zhenting Qi, Martin Riddell, Luke Benson, Lucy Sun, Ekaterina Zubova, Yujie Qiao, Matthew Burtell, David Peng, Jonathan Fan, Yixin Liu, Brian Wong, Malcolm Sailor, Ansong Ni, Linyong Nan, Jungo Kasai, Tao Yu, Rui Zhang, Shafiq Joty, Alexander R. Fabbri, Wojciech Kryscinski, Xi Victoria Lin, Caiming Xiong, and Dragomir Radev. Folio: Natural language reasoning with first-order logic. *arXiv preprint arXiv:2209.00840*, 2022.

[18] Suraj Kothawade, Vinaya Khandelwal, Kinjal Basu, Huaduo Wang, and Gopal Gupta. AUTO-DISCERN: autonomous driving using common sense reasoning. *CoRR*, abs/2110.13606, 2021.

[19] Robert Kowalski and Donald Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2(3):227–260, 1971.

[20] Imre Lakatos. *Proofs and Refutations: The Logic of Mathematical Discovery*. New York: Cambridge University Press, 1976.

[21] Xue Li. Abc_datalog. https://github.com/Xuerli/ABC_Datalog, 2022.

[22] Xue Li. *Automating the Repair of Faulty Logical Theories.* PhD thesis, University of Edinburgh, 2022.

[23] Xue Li and Alan Bundy. An overview of the abc repair system for datalog-like theories. In Alan Bundy and Denis Mareschal, editors, *Proceedings of 3rd International Workshop on Human-Like ComputingHLC2022 @ IJCLR*, volume 3227 of *Human-Like Computing Workshop 2022*, pages 11–17. CEUR Workshop Proceedings (CEUR-WS.org), October 2022.

[24] D. W. Loveland. *Mechanical Theorem-Proving by Model Elimination*, pages 117–134. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983.

[25] W. McCune. Prover9 and mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010.

[26] Jia Meng, Claire Quigley, and Lawrence C. Paulson. Automation for interactive proof: First prototype. *Information and Computation*, 204(10):1575–1596, 2006. Combining Logical Systems.

[27] Arnold Oberschelp. Order sorted predicate logic. In Karl Hans Bläsius, Ulrich Hedtstück, and Claus-Rainer Rollinger, editors, *Sorts and Types in Artificial Intelligence*, pages 7–17, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[28] Vladimir Pavlov, Alexander Schukin, and Tanzilia Cherkasova. Exploring automated reasoning in first-order logic: Tools, techniques and application areas. In Pavel Klinov and Dmitry Mouromtsev, editors, *Knowledge Engineering and the Semantic Web*, pages 102–116, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[29] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, jan 1965.

[30] Kazumi Saito, Pat Langley, Trond Grenager, Christopher Potter, Alicia Torregrosa, and Steven Klooster. Computational revision of quantitative scientific models. pages 336–349, 12 2001.

[31] Simon Schäfer and Stephan Schulz. Breeding theorem proving heuristics with genetic algorithms. In *Global Conference on Artificial Intelligence*, 2015.

[32] Raymond M. Smullyan. *First-Order Logic. Preliminaries*, pages 43–52. Springer Berlin Heidelberg, Berlin, Heidelberg, 1968.

[33] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[34] Kemas Wiharja, Jeff Z. Pan, Martin Kollingbaum, and Yu Deng. More is better: Sequential combinations of knowledge graph embedding approaches. In *Semantic Technology - 8th Joint International Conference, JIST 2018, Proceedings*, volume 11341 of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 19–35, Germany, November 2018. Springer Verlag. This work was supported by IBM Faculty Award and the EU Marie Currie K-Drive project (286348). Kemas Wiharja was also supported by the Lembaga Pengelola Dana Pendidikan (LPDP), the Ministry of Finance of Indonesia.; 8th Joint International Semantic Technology Conference, JIST 2018 ; Conference date: 26-11-2018 Through 28-11-2018.

Appendix A

Occurs Check Algorithm

Algorithm 1 Occurs check

```
Require: $\phi_1$, $\phi_2$ : Checks if $\phi_1$ occurs in $\phi_2$. $\phi_1$ is a variable, $\phi_2$ is a constant, variable, or function.
if $\phi_2$ is a constant then
    return False.
else if $\phi_2$ is a variable then
    if $\phi_1$ and $\phi_2$ are identical then
        return True.
    else
        return False.
    end if
else if $\phi_2$ is a function then
    for argument $i$ in $\phi_2$ do
        $S \leftarrow$ occurs($\phi_1$, $i$)
        if $S$ = True then
            return True.
        end if
    end for
    return False.
end if
```

Appendix B

Code Snippets

```
% TraceBackClause: Find back the original clause that introduced IC
    (in the case where IC is the ancestor.)
traceBackClause(IC,[],_,IC):- !.

traceBackClause(IC,_,TheoryIn,IC):-
    member(IC,TheoryIn),
    !.

traceBackClause(IC,Deriv,TheoryIn,OrgClause):-
    last(Deriv, (_,OrgClause,_,IC,_)),
    member(OrgClause,TheoryIn),
    !.

traceBackClause(IC,Deriv,TheoryIn,OrgClause):-
    last(Deriv, (_,OrgClauseT,_,IC,_)),
    \+member(OrgClauseT,TheoryIn),
    removeLast(Deriv,DerivNew),
    traceBackClause(OrgClauseT,DerivNew,TheoryIn,OrgClause),
    !.

traceBackClause(IC,Deriv,TheoryIn,OrgClause):-
    removeLast(Deriv,DerivNew),
    traceBackClause(IC,DerivNew,TheoryIn,OrgClause).
```

Listing B.1: traceBackClause function

```
memberNested(Elem,List):-
    member(Elem,List).

memberNested(Elem,[H|_]):-
```

```
    is_list(H),
    memberNested(Elem,H).

memberNested(Elem,[_|R]):-
    member([_|_],R),
    memberNested(Elem,R).

memberNested(_,[]):- fail.
```

Listing B.2: memberNested function

Appendix C

FOL Theory Examples

C.1 Example 1: Talent Show

Scenario in Natural Language

1. If people perform in school talent shows often, then they attend and are very engaged with school events.
2. People either perform in school talent shows often or are inactive and disinterested members of their community.
3. If people chaperone high school dances, then they are not students who attend the school.
4. All people who are inactive and disinterested members of their community chaperone high school dances.
5. All young children and teenagers who wish to further their academic careers and educational opportunities are students who attend the school.
6. Bonnie either both attends and is very engaged with school events and is a student who attends the school, or she neither attends and is very engaged with school events nor is a student who attends the school.

Representation in Clausal Form, Preferred Structure

```
axiom([-talentShows(\x),+engaged(\x)]).
axiom([+talentShows(\y),+inactive(\y)]).
axiom([-chaperone(\z),-students(\z)]).
axiom([-inactive(\a), +chaperone(\a)]).
axiom([-academicCareer(\b), +students(\b)]).
axiom([-engaged(bonnie),+students(bonnie)]).
axiom([-students(bonnie),+engaged(bonnie)]).

trueSet([engaged(bonnie)]).
falseSet([]).
protect([]).
heuristics([]).
```

Listing C.1: FOLIO Example 1

Fault State

Sufficiency: ∅

Insufficiency: $engaged(bonnie)$

Incompatability: ∅

Gold Standard

```
axiom([+academicCareer(bonnie)]).
axiom([-talentShows(\x),+engaged(\x)]).
axiom([+talentShows(\y),+inactive(\y)]).
axiom([-chaperone(\z),-students(\z)]).
axiom([-inactive(\a), +chaperone(\a)]).
axiom([-academicCareer(\b), +students(\b)]).
axiom([-engaged(bonnie),+students(bonnie)]).
axiom([-students(bonnie),+engaged(bonnie)]).
```

Listing C.2: FOLIO Example 1

An assertion, $academicCareer(bonnie)$, is added to the theory.

C.2 Example 2: Monkey Pox

Scenario in Natural Language

1. Monkeypox is an infectious disease caused by the monkeypox virus.
2. Monkeypox virus can occur in certain animals, including monkey.
3. Humans are mammals.
4. Mammals are animals.
5. Symptoms of Monkeypox include feeling tired, and so on.
6. People feel tired when they get a flu.

Representation in Clausal Form, Preferred Structure

```
axiom([+occurMonkeyPox(c1)]).
axiom([+getMonkeyPox(c1)]).
axiom([+animal(c2)]).
axiom([+occurMonkeyPox(c2)]).
axiom([-human(\x), +mammal(\x)]).
axiom([-mammal(\y), +animal(\y)]).
axiom([+getMonkeyPox(c3)]).
axiom([+tired(c3)]).
axiom([-human(\z), -flu(\z), +tired(\z)]).

trueSet([getMonkeyPox(human),animal(monkey)]).
falseSet([animal(c2)]).
protect([]).
heuristics([]).
```

Listing C.3: FOLIO Example 2

Fault State

Sufficiency: $\emptyset$

Insufficiency: $getMonkeyPox(human), animal(monkey)$

Incompatibility: $animal(c2)$

Gold Standard

```
axiom([+occurMonkeyPox(c1)]).
axiom([+getMonkeyPox(c1)]).
axiom([+animal(monkey)]).
axiom([+occurMonkeyPox(monkey)]).
axiom([-human(\x), +mammal(\x)]).
axiom([-mammal(\y), +animal(\y)]).
axiom([+getMonkeyPox(human)]).
axiom([+tired(human)]).
axiom([-human(\z), -flu(\z), +tired(\z)]).
```

Listing C.4: FOLIO Example 2 Gold Standard

The constants $c2$ and $c3$ from skolemization are renamed accordingly.

C.3 Example 3: Video Games

Scenario in Natural Language

1. A video game company created the game the Legend of Zelda.

2. All games in the Top 10 list are made by Japanese game companies.
3. If a game sells more than one million copies, then it will be selected into the Top 10 list.
4. The Legend of Zelda sold more than one million copies.

Representation in Clausal Form, Preferred Structure

```
axiom([+videoGameCompany(c1)]).
axiom([+game(thelegendofzelda)]).
axiom([+created(c1,thelegendofzelda)]).
axiom([-game(\x), -intop10(\x), -created(\y,\x),-videoGameCompany(\y
    ), +japanese(\y)]).
axiom([-game(\z), -sellmorethan(\z,onemillioncopies), +intop10(\z)])
    .
axiom([+sellmorethan(thelegendofzelda,onemillioncopies)]).

trueSet([intop10(thelegendofzelda),created(nintendo,thelegendofzelda
    ),japanese(nintendo)]).
falseSet([]).
protect([]).
heuristics([]).
```

Listing C.5: FOLIO Example 3

Fault State

Sufficiency: $intop10(thelegendofzelda)$

Insufficiency: $created(nintendo, thelegendofzelda), japanese(nintendo)$

Incompatibility: $\emptyset$

Gold Standard

```
axiom([+videoGameCompany(nintendo)]).
axiom([+game(thelegendofzelda)]).
axiom([+created(nintendo,thelegendofzelda)]).
axiom([-game(\x), -intop10(\x), -created(\y,\x),-videoGameCompany(\y
    ), +japanese(\y)]).
axiom([-game(\z), -sellmorethan(\z,onemillioncopies), +intop10(\z)])
    .
% axiom([+sellmorethan(thelegendofzelda,onemillioncopies)]).
```

Listing C.6: FOLIO Example 3 Gold Standard

The constants $c1$ from skolemization are renamed accordingly.

C.4 Example 4: Turkey types

Scenario in Natural Language

1. There are six types of wild turkeys: Eastern wild turkey, Osceola wild turkey, Gould's wild turkey, Merriam's wild turkey, Rio Grande wild turkey, and Ocellated wild turkey.
2. Tom is not an Eastern wild turkey.
3. Tom is not an Osceola wild turkey.
4. Tom is also not a Gould's wild turkey, or a Merriam's wild turkey, or a Rio Grande wild turkey.
5. Tom is a wild turkey.

Representation in Clausal Form, Preferred Structure

```
axiom([-wildTurkey(\x),+eastern(\x),+osceola(\x),+goulds(\x),+
    merriams(\x),+riogrande(\x),+ocellated(\x)]).
axiom([-wildTurkey(tom), -eastern(tom)]).
axiom([-wildTurkey(tom), -osceola(tom)]).
axiom([-wildTurkey(tom), -goulds(tom)]).
axiom([-wildTurkey(tom), -merriams(tom)]).
axiom([-wildTurkey(tom), -riogrande(tom)]).
axiom([+wildTurkey(tom)]).

trueSet([eastern(tom)]).
falseSet([ocellated(tom)]).
protect([]).
heuristics([]).
```

Listing C.7: FOLIO Example 4

Fault State

Sufficiency: $\emptyset$

Insufficiency: $eastern(tom)$

Incompatibility: $ocellated(tom)$

Gold Standard

```
axiom([-wildTurkey(\x),+eastern(\x),+osceola(\x),+goulds(\x),+
    merriams(\x),+riogrande(\x),+ocellated(\x)]).
axiom([-wildTurkey(tom), -ocellated(tom)]).
axiom([-wildTurkey(tom), -osceola(tom)]).
axiom([-wildTurkey(tom), -goulds(tom)]).
axiom([-wildTurkey(tom), -merriams(tom)]).
axiom([-wildTurkey(tom), -riogrande(tom)]).
axiom([+wildTurkey(tom)]).
```

Listing C.8: FOLIO Example 4 Gold Standard

The constraint axiom that asserts Tom is not an eastern turkey is instead renamed to be an ocellated turkey.

C.5 Example 5: Novel writer

Scenario in Natural Language

1. A podcast is not a novel.
2. If a person is born in American City, the person is American.
3. If a book is novel and it is written by a person, then the person is a novel writer.
4. Dani Shapiro is an American writer.
5. Family History is written by Dani Shapiro.
6. Family History is a novel written in 2003.
7. Dani Shapiro created a podcast called Family Secrets.
8. Boston is an American city.

Representation in Clausal Form, Preferred Structure

```
axiom([-isPodCast(\x), -isNovel(\x)]).
axiom([-bornIn(\y,f(\y)),-isCity(f(\y)),-isAmerican(f(\y)),+
    isAmerican(\y)]).
axiom([-isNovel(\a),-writtenBy(\a,\b),+writesNovel(\b)]).
axiom([+isAmerican(dani)]).
axiom([+isWriter(dani)]).
axiom([+writtenBy(familyHistory,dani)]).
axiom([+isNovel(familyHistory)]).
axiom([+writtenIn(familyHistory,y2003)]).
axiom([+isPodCast(familySecrets)]).
axiom([+createdBy(familySecrets,dani)]).
axiom([+isCity(boston)]).
axiom([+isAmerican(boston)]).
```

```

eqAxiom([f(dani),boston]).

trueSet([bornIn(dani,boston)]).
falseSet([writesNovel(dani)]).
protect([]).
heuristics([]).
```

Listing C.9: FOLIO Example 5

Fault State

Sufficiency: $\emptyset$

Insufficiency: $bornIn(dani,boston)$

Incompatibility: $writesNovel(dani)$

Gold Standard

```
axiom([+bornIn(dani,f(dani))]).
axiom([-isPodCast(\x), -isNovel(\x)]).
axiom([-bornIn(\y,f(\y)),-isCity(f(\y)),-isAmerican(f(\y)),+
    isAmerican(\y)]).
axiom([-isNovel(\a),-writtenBy(\a,\b),+writesNovel(\b)]).
axiom([+isAmerican(dani)]).
axiom([+isWriter(dani)]).
axiom([+writtenBy(familyHistory,dani)]).
axiom([+writtenIn(familyHistory,y2003)]).
axiom([+isPodCast(familySecrets)]).
axiom([+createdBy(familySecrets,dani)]).
axiom([+isCity(boston)]).
axiom([+isAmerican(boston)]).
```

Listing C.10: FOLIO Example 5 Gold Standard

The fact that Dani is born in Boston is asserted with a new theorem, using the function f for better generalization. The assertion $isNovel(familyHistory)$ is deleted to repair the incompatibility.

Appendix D

Eggtimer Illustration, Prolog Theory Files

D.1 Illustration

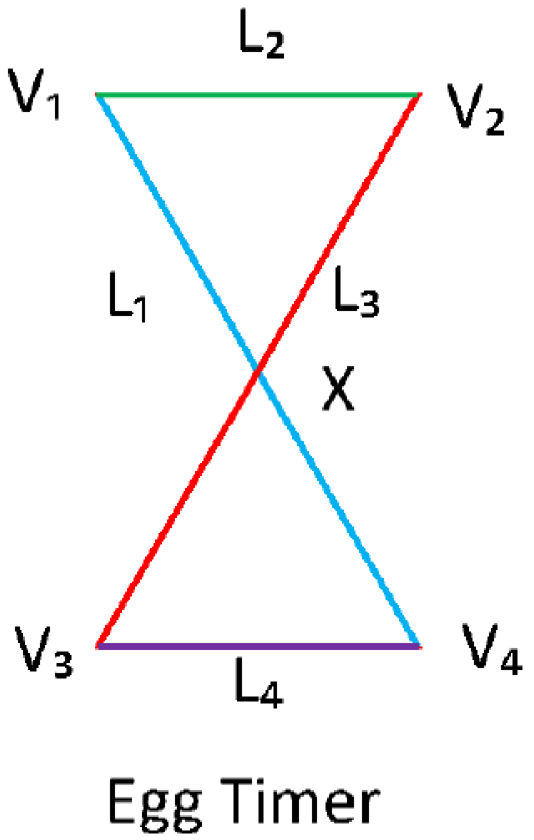

Figure D.1: Visualization of eggtimer [7]

D.2 Attempt 1 - Prolog Theory File

```
axiom([+meet(v1,l1,l2)]).
axiom([+meet(v2,l2,l3)]).
axiom([+meet(v3,l3,l4)]).
axiom([+meet(v4,l4,l1)]).
axiom([+meet(x,l1,l3)]).
axiom([-meet(\v1,\l1,\l2),-meet(\v2,\l2,\l3),-meet(\v3,\l3,\l4),-
    meet(\v4,\l4,\l1),+polygon(eggtimer,set_of(\v1,\v2,\v3,\v4),
    set_of(\l1,\l2,\l3,\l4))]).

trueSet([]).
falseSet([polygon(eggtimer,set_of(v1,v2,v3,v4),set_of(l1,l2,l3,l4))
    ]).
```

Listing D.1: Eggtimer formulation 1

D.3 Attempt 2 - Prolog Theory File

```
axiom([+meet(v1,l1,l2)]).
axiom([+meet(v2,l2,l3)]).
axiom([+meet(v3,l3,l4)]).
axiom([+meet(v4,l4,l1)]).
axiom([+meet(x,l1,l3)]).
axiom([-eq(l1,l3)]).
axiom([-eq(l3,l1)]).
% More constraints on equality omitted...
axiom([+ls(l1)]).
axiom([+ls(l2)]).
axiom([+ls(l3)]).
axiom([+ls(l4)]).
axiom([-meet(\p,\l1,\l2),+vs(\p)]). %Equation 4.2
axiom([-vs(x),-ls(\l1),-ls(\l2),+eq(\l1,\l2),-meet(x,\l1,\l2),+
    polygon(eggtimer)]). %equation 4.1

trueSet([]).
falseSet([polygon(eggtimer)]).
protect([ls,eq,arity(ls),arity(eq)]).
```

Listing D.2: Eggtimer formulation 2