Explanation of Counterexamples in the Context of Formal Verification

Alexander Ek
Abstract

Explanation of Counterexamples in the Context of Formal Verification

Alexander Ek

With the current rapid computerisation and automation of systems, which were previously controlled manually, a growing demand for measures to ensure correctness of systems is arising. This can be achieved with formal verification. With formal verification, systems can be proved to satisfy, or not satisfy, a set of given properties. In cases where a system does not satisfy the given properties, a counterexample is presented, but counterexamples are, more often than not, hard to interpret manually. Thus, automatic procedures for explaining counterexamples are needed but unfortunately few.

This thesis will focus on the explanation of counterexamples from failed formal verifications of systems. An algorithm for doing such is presented in this thesis. This algorithm is shown to give desired explanations for some cases whilst just the bare minimum for the majority of cases. A number of procedures to obtain the latter are already known. However, a small problem, left unsolved, can bring desired explanations to the majority of cases if solved. The presented algorithm is, unfortunately, not free from flaws nor errors, but it is still unclear how severe these are.
1 Introduction

With the current rapid computerisation and automation of systems, which were previously controlled manually, comes growing demand for measures to ensure correctness of systems. Simple thorough testing can not ensure correctness since it can only show presence of errors, not absence. Correctness becomes especially important with critical systems such as systems for controlling nuclear power plants, and systems for train scheduling and signalling. An error in these kinds of systems can cause disasters. This is where formal verification is necessary.

Formal verification is using formal methods of mathematics to prove or disprove the correctness of the underlying structure and meaning of algorithms and systems with respect to formal specifications \[1, 4\]. A system and its behaviour, as well as the requirements of that system, can be expressed formally in predicate logic (or temporal logic). The requirements of a system are properties that must hold in every valid configuration of the system in order for the system to be considered correct (i.e. a formal specification). Let the formula \( \text{system} \) be a logical description of some system and the formula \( \text{requirements} \) be a logical description of the requirements of \( \text{system} \). Now, the formula \( \text{system} \rightarrow \text{requirements} \) expresses the property that the system is considered correct; that in every allowed configuration of the system, the requirements are fulfilled. The aforementioned formula can be automatically proved using formal methods of mathematics. That is done by first translating the formula into propositional logic, thereafter running it through a satisfiability (SAT) solver. If the implication is valid, the system will always satisfy its requirements. If the formula is falsifiable, however, then the SAT solver will produce a counterexample; it will produce an interpretation that falsifies the formula.

1.1 The problem

In essence, given a formula, along with a counterexample, the task is to explain why the formula is not satisfiable in the provided counterexample, as well as possible. Counterexamples are truth-value configurations that interprets the formula as false. Explanations for interpreted formulae consists of an atom such that, if the atom changes truth value, then the interpreted formula will also change truth value.

However, it is relatively hard to manually pin-point a causal factor for the generated counterexample and, unfortunately, automatic procedures for doing such are few and incomplete \[3\]. Therefore manual work is frequently needed. Although an automatic debugging process is hard to construct, constructing a system that thoroughly helps the manual debugging process and explains the error, automatically, is feasible.

Although many systems may be easier to model with temporal logic, it was disregarded for this thesis due to the limited time frame. Thus, only predicate logical descriptions of systems and requirements will be considered. Furthermore, with predicate logic, only finite domains will be used.

\[1\] This is not generally true; it only holds for certain logics.
Ideally, the input would only be the formula of the system and its requirements. A counterexample would then be constructed (if one exists) and then the explanation process would start. The output would be a list of atoms such that, if at least one of them would change truth value, then the whole input formula would change truth value. This list would be an overview of the result. More detailed explanations of the atoms (where and why it was found) would also be presented. For example, take the system \( p(a) \leftrightarrow p(b) \) (call it \( sys \)) with the requirements \( p(a) \) (call it \( req \)), where \( p(a) \) and \( p(b) \) are atoms. It could look like this:

**Input:** \( sys \rightarrow req \)

**Output:** Final explanation: \( p(b) \),
related explanations: \( p(a) \)

Here, simply writing the formula would start the proving process. The output would remark that the input formula is falsifiable and would give a final explanation (\( p(b) \) in this case) and a list of related explanations (\( p(a) \) in this case). However, it might be interesting to see what truth value each atom is assigned to in the current counterexample. In that case, the command `/list` would be used. The command would list all true atoms in the current interpretation. It could look like this:

**Input:** `/list`

**Output:** All atoms are false

Here, all atoms are false. This is in fact the only false interpretation of this formula. In case further details of why a certain explanation was found, would be desired, a command where a question mark immediately followed the explanation in question, would be used. It could look like this:

**Input:** \( p(b)? \)

**Output:** \( p(b) \) is false,
found in true subformulae: \( p(a) \leftrightarrow p(b) \),
because related explanation to: \( p(a) \),

**Input:** \( p(a)? \)

**Output:** \( p(a) \) is false,
Found in false subformulae: \( req \),
because 'p(a)' is false, and
because 'req' is false, and
because 'sys \rightarrow req' (root) is false with 'sys' true

Here, detailed information about each of the two explanations would be provided. It would show how the formula was traversed and what led to the atom being considered an explanation.
One big problem with automated explanation of counterexamples of formulae is that there are, more often than not, no clear or direct answers. This is a result of the fact that there is generally an external underlying meaning attached to the system and the requirements which is not accessible from the abstract formulae itself. This creates a guessing-game for an automatic explanation procedure. There may be multiple atoms that can change value of the input formula. In such cases, the best atom to chose as an explanation may be apparent to a system designer but seem completely arbitrary to an automatic procedure.

1.2 Prior work

This thesis is mainly built upon, and inspired by, the GTO toolset by L.-H. Eriksson [2, 3]. The GTO toolset provides great means for simulating and proving formal systems expressed in a simple form of temporal first-order predicate logic with finite domains (only a previous-moment temporal operator and without function symbols). Because the domains are finite, problems can be, and are, reduced to propositional satisfiability problems, and solved by an independent SAT solver which is interfaced to GTO. Eriksson outlines the functions of the GTO toolset and shows how it can be used for specification and verification work. It is noted that analysis of counterexamples is important. However, it is a task which has not attracted much research and has very little automation support. The means of the GTO toolset to analyse counterexamples of failed proof attempts are limited and only finds one direct reason. This leaves a lot of work to be done by hand. It is not believed that the process of explaining counterexamples can be automated completely, but a tool providing more detailed information, thus substantial support to the user finding explanations, is possible and also desired.

Assume a counterexample, where \( p(a) \) and \( p(b) \) are False, to the formula:

\[
\text{system} \quad (p(a) \leftrightarrow p(b)) \rightarrow p(a) \\
\text{requirements}
\]

Here, GTO would simply give \( p(a) \) as the explanation whilst \( p(b) \) seems to be an equally good explanation. Note, however, that \( p(b) \) only exists in the system, as opposed to \( p(a) \), which appears in the requirements as well. Note, also, that \( p(b) \) and \( p(a) \) seem to have some connection since they occur as direct subformulae of the same connective. The question here is whether changing the truth value of \( p(b) \) is more desirable than changing the truth value of \( p(a) \). Giving both as separate, but somehow connected, explanations seems to be the best choice in this case. For instance, \( p(b) \) could be given as the first explanation and \( p(a) \) as the second explanation.

In the implementation of the algorithm that will be presented, and the infrastructure, syntax almost identical to the syntax of the GTO toolset will be used. This is mainly because GTO will be used as a platform for finding counterexamples as well as finding out if a formula is valid. Thus bringing focus to the main point of this thesis.
1.3 Contributions

The aim of this thesis is to construct an algorithm that produces more comprehensive and further detailed explanations than GTO: taking it one step further. In this thesis, an attempt at designing an algorithm for explaining counterexamples is presented (described in Section 3.2). This algorithm uses concepts called explanations and related explanations, detailed in Section 3.1 and uses an unproved simplification, discussed in Section 6.1. There are some cases where the algorithm produces better explanations than GTO (described in Section 5). However, the algorithm has some flaws and errors as well (described in Section 6). Luckily, the most prominent documented error is considered rare. There are still a lot of work left, some of which are presented in Section 8.1.

Together with this algorithm, an infrastructure that can parse files and input data, both syntactically and semantically is presented (described in Section 4.1). The infrastructure has a user interface and a back-end. There are some non-critical flaws with the infrastructure that are also mentioned.

2 Background

First, notations and definitions need to be established. Basic mathematical logic is presumed to be familiar to the reader.

2.1 Notation

Notation 1. All sets are written in boldface (example, Example). Names of sets that are also metavariables, begin with lowercase letters while names of constant sets begin with uppercase letters. Well known sets such as the natural numbers (N₀, N>₀) are exceptions to this.

Notation 2. All metavariables (except sets) are written in italics (example) and begin with a lowercase letter, or written with a single Greek letter (α, Φ).

Notation 3. All keywords are written in italics with initial capitalisation (Nothing, True).

Notation 4. All functions are written in sans serif (function).

Notation 5. Program code, program output and program input are written in typewriter font (example).

Notation 6. To not cause confusion, for every boolean metavariable foo, foo is written to denote its negation.

2.2 Relevant terminology of logic

In this section, definitions and explanations will be presented to clarify how the logic looks like and how it is used. For boolean constants, True will be used to denote constant truth and False will be used to denote constant falsity.
Definition 2.1. (Constant) A constant is simply an object that does not change meaning. Two constants can be checked for equality and inequality (denoted = and \(\neq\) respectively). Let the set of all constants be denoted \textbf{Const}.

Definition 2.2. (Variable) Variables can take on the value of some constant. Let the set of all variables be denoted \textbf{Var}.

Definition 2.3. (Variable domain) A variable domain is a finite set of constants that variables can range over (assume any element from). Every variable must range over exactly one variable domain; however, any number of variables can range over the same variable domain. Furthermore, no constant can belong to more than one variable domain that has variables ranging over it (in this thesis, neither sub- nor superdomains are regarded). Also, only constants belonging to a variable domain that some variable ranges over can be used.

Let \(x\) be a variable and \(\mathbf{x}\) be a set of constants; \(\mathbf{x} : x\) denotes that \(\mathbf{x}\) is the variable domain of \(x\). Let the set of all variable possible domains be denoted \(\text{Domain}\). Note that \(\text{Domain} \subseteq \mathcal{P}(\text{Const})\) (only finite subsets).

Definition 2.4. (Predicate and atom) A predicate is a boolean-valued function (can assume either the value \textit{True} or \textit{False}). A predicate can have any non-negative number of arguments; this number is called the arity of the predicate. Let \(p\) be a predicate with arity \(n \in \mathbb{N}_0\). The declaration of \(p\) is denoted: \(p(x_1, \ldots, x_n)\), where \(x_1, \ldots, x_n\) are non-empty elements of \textbf{Domain}, such that each variable domain has some variable ranging over it. (Parentheses are omitted in the declaration if \(n = 0\).)

In the declaration of \(p\), \(x_j\) denotes the domain of the \(j\)th argument of \(p\), for each \(0 < j \leq n\). When using \(p\), only constants in \(x_j\) and variables ranging over \(x_j\) can be used as the \(j\)th argument, for every \(0 < j \leq n\).

A predicate, together with a constant or variable for each of its arguments, is called an atom. The set of all predicates is denoted \textbf{Pred} and the set of all atoms is denoted \textbf{Atom}. Note that every predicate in \textbf{Pred} contains information about the arity of the predicate and the domains of its arguments.

Definition 2.5. (Ground atom) An atom is also a ground atom if and only if it has no variables as arguments; if and only if all of its arguments are constants. Atoms that are not ground atoms can not be evaluated. The set of all ground atoms is denoted \textbf{Atom}_G. Note that \(\text{Atom}_G \subset \text{Atom}\).

Remark. Some clarifications on Definitions 2.4 and 2.5: First, for each predicate there may exist multiple possible atoms, some of which are ground atoms. Furthermore, if a predicate \(p\) is 0-ary (i.e. no arguments), then \(p\) is a predicate, an atom and a ground atom (and the only atom of \(p\)).

Definition 2.6. (Literal) Literals are introduced to simplify notation. A ground literal is either a ground atom or the negation of a ground atom. The set of all ground literals is denoted \textbf{Lit}_G. \footnote{\(\mathcal{P}\) denotes the powerset function. The powerset of a set \(s\) (denoted \(\mathcal{P}(s)\)) is the set of all possible subsets of \(s\).}
Definition 2.7. (Formula) Let \( c_1 \) and \( c_2 \) be constants, let \( x_1 \) and \( x_2 \) be variables ranging over the same variable domain. Let \( \alpha \) be a ground atom and \( \beta \) be a non-ground atom. Note that, \( \beta \) must have at least one argument that is a variable. Every formula is constructed, from Form, in the following manner:

\[
\text{Rel} := c_1 = c_2 \mid c_1 = x_1 \mid x_1 = c_1 \mid x_1 = x_2 \mid c_1 \neq c_2 \mid c_1 \neq x_1 \mid x_1 \neq c_1 \mid x_1 \neq x_2
\]

\[
\text{Form} := \text{Form} \land \text{Form} \mid \text{Form} \lor \text{Form} \mid \text{Form} \rightarrow \text{Form} \mid \text{Form} \leftrightarrow \text{Form} \mid \neg \text{Form}
\]

where, \( \land \) denotes conjunction (and), \( \lor \) denotes disjunction (or), \( \rightarrow \) denotes implication (if then), \( \leftrightarrow \) denotes equivalence (if and only if), \( \neg \) denotes negation, \( \forall \) denotes universal quantification (for all), \( \exists \) denotes existential quantification (for at least one).

A formula \( \Phi \) is closed if each occurrence (in \( \Phi \)) of a variable \( x \) is in some formula \( \Psi \) such that \( \forall x \Psi \) or \( \exists x \Psi \) is a subformula to \( \Phi \), for every variable. Variables not satisfying this property are called unbound variables. If a formula is not closed, it is an open formula, and hence has some unbound variable. Only closed formulae can be evaluated.

Let Form denote the set of all closed formulae, and let Form\(_O\) denote the set of all (closed and open) formulae. Let Conn\(_B\) = \{\( \land, \lor, \rightarrow, \leftrightarrow \)\} (i.e. the set of all binary connectives).

Note. In this thesis, \( \land \) is considered to have a higher tightness of binding than \( \lor \).

Definition 2.8. (Variable substitution) In order to create a closed formula from an open formula, variable substitution is necessary. Let \( \Phi \in \text{Form}\(_O\) \), \( c \in x \in \text{Domain} \), and \( x \in \text{Var} \) such that \( x : x \). The formula identical to \( \Phi \), but with every unbound instance of \( x \) in \( \Phi \) replaced by \( c \), is denoted \( \Phi[x \mapsto c] \); and that \( x \) is replaced by \( c \) is similarly denoted \( x[x \mapsto c] \).

Let \( \Psi \in \text{Form}\(_O\) \), and \( \circ \in \text{Conn}\(_B\) \). Let \( p(a_1, \ldots, a_n) \in \text{Atom} \) be an \( n\)-ary atom, where \( a_1, \ldots, a_n \in \text{Var} \cup \text{Const} \). Also, let \( y \in \text{Var} \). The process of variable substitution can be described as:

\[
\begin{align*}
a_1[x \mapsto c] &= c, \text{ if } a_1 \text{ is a variable and is } x, a_1 \text{ otherwise.} \\
(\neg \Phi)[x \mapsto c] &= \neg(\Phi[x \mapsto c]) \\
(\Phi \circ \Psi)[x \mapsto c] &= \Phi[x \mapsto c] \circ \Psi[x \mapsto c] \\
(a_1 = a_2)[x \mapsto c] &= a_1[x \mapsto c] = a_2[x \mapsto c] \\
(a_1 \neq a_2)[x \mapsto c] &= a_1[x \mapsto c] \neq a_2[x \mapsto c] \\
p(a_1, \ldots, a_n)[x \mapsto c] &= p(a_1[x \mapsto c], \ldots, a_n[x \mapsto c]) \\
(\forall y \Phi)[x \mapsto c] &= \forall y(\Phi[x \mapsto c]), \text{ if } y \text{ and } x \text{ are different variables} \\
(\forall x \Phi)[x \mapsto c] &= \forall x \Phi \\
(\exists y \Phi)[x \mapsto c] &= \exists y(\Phi[x \mapsto c]), \text{ if } y \text{ and } x \text{ are different variables} \\
(\exists x \Phi)[x \mapsto c] &= \exists x \Phi \\
\Phi[x \mapsto c] &= \Phi, \text{ if no other rules apply.}
\end{align*}
\]
Definition 2.9. *(Predicate definition)* A predicate can be defined to represent a (possibly open) formula. Let \( p \in \text{Pred} \) be a predicate with arity \( n \in \mathbb{N}_0 \). If \( p \) is defined to represent a formula \( \Phi \in \text{Form}_0 \), \( p \) is called a *defined* predicate (or defined atom).

The definition of \( p \) representing \( \Phi \) is denoted \( p(x_1, \ldots, x_n) := \Phi \), where \( x_1, \ldots, x_n \) are distinct elements of \( \text{Var} \). Let \( x_1 : x_1, \ldots, x_n : x_n \) where \( x_1, \ldots, x_n \in \text{Domain} \) (not necessarily distinct domains) be the domains of the arguments as set by the declaration of \( p \). Moreover, the set of all variables in the formula \( \Phi \) must be a subset of \( \{x_1, \ldots, x_n\} \).

And, when evaluating \( p \) as a ground atom, with some constant arguments \( c_1, \ldots, c_n \in \text{Const} \), where \( c_1 \in x_1, \ldots, c_n \in x_n \); the formula \( \Phi[x_1 \mapsto c_1] \ldots [x_n \mapsto c_n] \) is evaluated.

Predicates that are not defined are simply called *undefined* predicates (or undefined atoms). Let the set of all possible combinations of predicate definitions be denoted \( \text{Def} = \text{Pred} \mapsto (\text{Var}^n \times \text{Form}_0) \), where \( n \) is the arity of the corresponding predicate. Note that, a predicate definition is a mapping from a predicate to a tuple of variables and a possibly open formula.

Let \( \text{def} : \text{Atom}_G \times \text{Def} \mapsto \text{Form} \) be a function that gives the closed formula definition of a ground atom according to a set of predicate definitions. If the ground atom is undefined, the output formula is the input atom.

**Definition 2.10.** *(Unrolling quantifiers)* To transform formulae of predicate logic into propositional logic, the quantifiers needs to be replaced with something else. This is referred to as unrolling the quantifiers. Let \( \text{unroll} : \text{Form} \mapsto \text{Form} \) denote the unrolling function. Let \( \Phi, \Psi \in \text{Form}, \odot \in \text{Conn}_B \), and let \( x \in \text{Var} \) and \( x : x \) such that \( x = \{c_1, \ldots, c_n\} \in \text{Domain} \). Also, let \( \text{def} \in \text{Def} \) be a set of predicate definitions and let \( \alpha \in \text{Atom}_G \). The function \( \text{unroll} \) can be defined as follows:

\[
\begin{align*}
\text{unroll}(\neg \Phi) & = \neg(\text{unroll}(\Phi)) \\
\text{unroll}(\Phi \odot \Psi) & = \text{unroll}(\Phi) \odot \text{unroll}(\Psi) \\
\text{unroll}(\forall x \Phi) & = \text{unroll}(\Phi[x \mapsto c_1]) \land \cdots \land \text{unroll}(\Phi[x \mapsto c_n]) \\
\text{unroll}(\exists x \Phi) & = \text{unroll}(\Phi[x \mapsto c_1]) \lor \cdots \lor \text{unroll}(\Phi[x \mapsto c_n]) \\
\text{unroll}(\alpha) & = \text{unroll}(\text{def}(\alpha, \text{def})), \text{ if } \alpha \neq \text{def}(\alpha, \text{def}). \\
\text{unroll}(\Phi) & = \Phi, \text{ if no other rules apply.}
\end{align*}
\]

**Definition 2.11.** *(Interpretation and counterexample)* An interpretation \( i \) of a formula \( \Phi \) is a truth value assignment to all the ground atoms in \( \Phi \). The truth value of the of \( \Phi \) when interpreted with \( i \) is denoted \( \Phi^i \). That \( \Phi^i \) is true is written \( i \models \Phi \) (and \( i \not\models \Phi \), if it is false). That \( \Phi \) is true for every possible interpretation is denoted \( \models \Phi \) (with a specific interpretation omitted). Furthermore, every possible truth value assignment is a unique interpretation, and a counterexample is an interpretation that makes the formula false.

In this thesis, an interpretation is represented as a set of ground atoms where included atoms are true and excluded atoms are false. Let the set of all possible interpretations be denoted \( \text{Interp} \). Note that \( \text{Interp} = \mathcal{P}(\text{Atom}_G) \).

**Definition 2.12.** *(Environment)* In order for a formula to be properly examined, an interpretation, a set of predicate definitions and, a mapping from variables to variable
domains (i.e. what each variable ranges over) are needed. A collection these is called an environment. The set of all environments is denoted $\text{Env}$. Note that $\text{Env} = \text{Interp} \times \text{Def} \times (\text{Var} \rightarrow \text{Domain})$.

**Definition 2.13.** (Comparison of formulae) Let $\Phi, \Psi \in \text{Form}$ be two formulae.

- **(Identity)** If $\Phi$ and $\Psi$ are identical (denoted $\Phi = \Psi$), $\Phi$ and $\Psi$ have exactly the same structure. Parentheses that does not alter the truth values of a formula have no effect on the structure of the formula.

- **(Semiidentity)** If $\Phi$ and $\Psi$ are semiidentical (denoted $\Phi \approx \Psi$), it is possible to add some number of negations in front of either $\Phi$ or $\Psi$ such that the two becomes identical.

- **(Equivalence)** If $\Phi$ and $\Psi$ are (logically) equivalent (denoted $\Phi \leftrightarrow \Psi$, not to be confused with $\leftrightarrow$), $\Phi$ will assume the same truth value as $\Psi$, for every interpretation. This is simply a more convenient way of writing $|\models \Phi \leftrightarrow \Psi$.

**Definition 2.14.** (Direct subformula) Let $\Phi, \Psi, \Omega \in \text{Form}$ be formulae, and $\odot \in \text{Conn}_B$. Let $x \in \text{Var}$ such that $x : x$, where $x \in \text{Domain}$. The formula $\Psi$ is a direct subformula to $\Phi$ if and only if any of the following properties is true:

- $\Phi = \neg \Psi$
- $\Phi = \Psi \odot \Omega$
- $\Phi = \Omega \odot \Psi$
- $\Phi = \forall x \Omega$ and $\Omega[x \mapsto c] = \Psi$, for some constant $c \in x$.
- $\Phi = \exists x \Omega$ and $\Omega[x \mapsto c] = \Psi$, for some constant $c \in x$.
- $\Psi = \text{def}(\Phi, \text{def})$ and $\Psi \neq \Phi$.

where $\text{def} \in \text{Def}$ is a set of predicate definitions. Furthermore, $\Phi$ is a direct superformula to $\Psi$ if and only if $\Psi$ is a direct subformula to $\Phi$.

**Remark.** Let $\Phi \in \text{Form}$. It follows from Definition 2.13 that $\Phi$ is identical to the parenthesised formula ($\Phi$), and since no formula is a direct subformula to itself, $\Phi$ is not a direct subformula to ($\Phi$).

**Definition 2.15.** (Top-level connective) Let $\Phi, \Psi_1, \Psi_2 \in \text{Form}$, and let $\odot \in \text{Conn}_B$. The top-level connective of $\Phi$ is $\odot$ if $\Phi = \Psi_1 \odot \Psi_2$, and it is $\neg$ if $\Phi = \neg \Psi_1$. If $\Phi$ is a ground atom defined as $\Psi_1$, then the top-level connective of $\Phi$ is the top-level connective of $\Psi_1$. Otherwise, $\Phi$ does not have a top-level connective.

**Definition 2.16.** (Classification of formulae) Let $\Phi \in \text{Form}$.

- If $\Phi$ is true in every interpretation (i.e. $|\models \Phi$), then $\Phi$ is said to be valid or a valid formula.
• If $\Phi$ is true in at least one interpretation, then $\Phi$ is said to be *satisfiable* or a satisfiable formula.

• If $\Phi$ is false in every interpretation, then $\Phi$ is said to be *unsatisfiable* or an unsatisfiable formula.

• If $\Phi$ is false in at least one interpretation, then $\Phi$ is said to be *falsifiable* or a falsifiable formula.

• If $\Phi$ is valid or unsatisfiable (in other words, $\Phi$ assumes the same truth value in every interpretation), then $\Phi$ is said to be *invariable* or an invariable formula.

3 Theory

The theoretical principles of the algorithm, as well as the concepts surrounding it, will be explained in this section.

3.1 Concepts

Understanding of the underlying concepts of the algorithm is crucial for a thorough understanding of the algorithm itself. These concepts will, therefore, be defined and explained in this section.

Let $i \in \text{Interp}$ be an interpretation, let $\Phi \in \text{Form}$, and let $\alpha \hookrightarrow \beta \in \text{Atom}_G$ be ground atoms.

**Definition 3.1.** (*Modification*) The modification of an interpreted ground atom $\alpha^i$ is another interpretation $i'$ where $i'$ is the symmetric difference\(^3\) of $i$ and $\{\alpha\}$. Hence, $\alpha^i \neq \alpha^{i'}$. Note that only one such $i'$ exists for each $i$.

Similarly, the modification of an interpreted formula $\Phi^i$ is another interpretation $i'$, such that $\Phi^i \neq \Phi^{i'}$, but where $i'$ is the symmetric difference between $i$ and a set of ground atoms $a$. In this case, there may exist multiple interpretations $i'$ for each $i$, all with possibly different cardinality of $a$. $\Phi^i$ is said to be $k$-explainable if there exists a set $a$ with cardinality $k$.

**Remark.** An interpreted formula may be $k$-explainable but not necessarily $(k + 1)$-explainable or $(k - 1)$-explainable, for any $k \in \mathbb{N}_{>0}$.

**Definition 3.2.** (*Independence*) An interpreted formula $\Phi^i$ is said to be independent of $\alpha$ if the modification of $\alpha^i$ is not a modification of $\Phi^i$.

**Definition 3.3.** (*Explanation*) $\alpha$ is an explanation for $\Phi^i$ if the modification of $\alpha^i$ is a modification of $\Phi^i$.

---

\(^3\) The symmetric difference of two sets is the set of elements which are in the union but not in the intersection.
Remark. It follows from Definition 3.3 that $\Phi^i$ must be 1-explainable to have an explanation. The definition could, however, be extended to $k$-explainable interpretations of formulae (for any $k \in \mathbb{N}_0$), but making use of that lies beyond the scope of this thesis.

In order to find a relevant series of explanations to a system, some notion of related explanations needs to be established. In this thesis, two explanations are defined to be related if they are different ways of “fixing” the same problem. There is some sort of semantic connection. A more formal definition of related explanations now follows.

**Definition 3.4. (Related explanation)** Let $\alpha$ be an explanation for $\Phi^i$. $\beta$ is a related explanation to $\alpha$ in $\Phi^i$ if:

- $\alpha \neq \beta$, and
- $\beta$ is an explanation for $\Phi^i$ and $\Phi^i'$ is independent of $\alpha$, where $i'$ is the modification of $\beta$ (and hence a modification of $\Phi^i$).

**Remark.** If $\beta$ is a related explanation to $\alpha$ in $\Phi^i$, then $\alpha$ will be a related explanation to $\beta$ in $\Phi^i$ as well. This follows from Definitions 3.1, 3.3 and 3.4. A detailed proof has been omitted since the result is not used in this thesis.

**Definition 3.5. (Candidate formula)** Let $\alpha$ be an explanation for $\Phi^i$, let $\odot \in \text{Conn}_B$, and let $\Psi \in \text{Form}$. Let the formula $\alpha'$ be constructed by negating $\alpha$ any (non-negative) number of times. Now, a subformula $\alpha' \odot \Psi$ (or $\Psi \odot \alpha'$) to $\Phi$ is a candidate formula to $\alpha$ in $\Phi^i$ if both the following conditions are true:

- There exists a modification $i'$ of $\Psi^i$ that is a modification of $\Phi^i$ such that $\Phi^i'$ is independent of $\alpha'$.
- The instance of $\alpha$ (in $\alpha'$) in the candidate formula is in a different (in another part of $\Psi$) from the instance of $\alpha$ that is identified as an explanation.

The last conditions is important because if the instances were identical, there would be a reason to why the explanation in $\alpha'$ was prioritised over an explanation in $\Psi$. Hence, an explanation in $\Psi$ would not be as interesting.

**Remark.** In Definition 3.5, if $\Psi$ is 1-explainable, it follows from Definitions 3.1 and 3.3 that there would exists a $\beta$ that would be an explanation to $\Psi^i$ and an explanation to $\Phi^i$. Furthermore, by Definition 3.4 $\beta$ would be a related explanation to $\alpha$ in $\Phi^i$.

Candidate formulae are used to enforce a syntactical connection between the explanations, in addition to the semantic connection. The syntactical connection comes from the conditions on structure of the system.

### 3.2 Explanation algorithm

The algorithm is given a formula and an environment. Let $i \in \text{Interp}$ be the provided interpretation (counterexample), and let $\text{def} \in \text{Def}$ be the predicate definitions.
The main idea of the algorithm is first finding an explanation, which is an relatively easy task. If and when an explanation is found, a search for a candidate formula begins. This is done by backtracking in the formula. If and when a potential candidate formula is found, the direct subformula of it that has a potential related explanation is searched for a regular explanation. This explanation can result in several related explanations found during the explanation process. If an explanation is found, it is then considered to be a possible related explanation to the previous one. In any case, it is still only known that this is a related explanation in the subformula currently explored. Further backtracking will then confirm (or deny) that the related explanation holds in superformulae. When no more traversing is possible by the algorithm, a list of related explanations will be outputted as the result.

The algorithm is divided into six functions and two phases, the explanation phase and the candidate formula phase. There are three functions in each phase. The explanation phase identifies potential explanations and the candidate formula phase identifies potential candidate formulae. These are potential in the sense that they may only be explanations and candidate formulae to some subformulae of the main formula but not the main formula itself. The explanation phase includes the functions expl, expl’ and, expl’’, whilst the candidate formula phase includes find, find’ and, find’’. Of these, expl’’ is the only function that has no recursive calls to any of these six functions. There is a seventh function, prio, that controls the order of exploration in the formula.

The negations need to be pushed down to atom-level in the formula (with De Morgan’s laws). This is because there was not enough time to investigate how negations propagated throughout the formula when returning explanations and related explanations. There is an exception with implication regarding this which may create a problem (see Section 6.2).

### 3.2.1 Finding an explanation to a counterexample

Finding an explanation is simply done by traversing the formula inwards. Both direct subformulae of a binary connective may be visited if at least one of the subformulae have no explanation. A list (ordered set) of all previous explanations is included (called prevs), this is mainly used when finding related explanations. Note that the environment will be omitted as arguments to the recursive formulae (and seen as global) since they never change between recursive calls.

Despite that explanations are defined as atoms, literals are returned as explanations by the recursive formulae defining the algorithm. This is since some simplified evaluations depends on whether an explanation has the same truth value, or the opposite truth value, as the interpreted formula it is an explanation for.

*Note.* When negating a formula that is already negated, the negation is assumed to be removed; thus, no double negation is introduced by the algorithm. However, an input formula may have double (or more) negations.

The first function, expl, is used to identify atoms that are explanations or can not be
explanations (in case of boolean constants or (in)equalities). Let

$$\text{expl} : \text{Form} \times \mathcal{P}(\text{Lit}_G) \rightarrow (\text{Lit}_G \cup \{\text{Nothing}\}) \times \mathcal{P}(\text{Lit}_G)$$

This function is also responsible for unrolling quantifiers, expanding definitions and traversing negations. If a binary connective is reached, expl' is called.

The function expl can now be defined:

$$\text{expl}(\Phi, \text{prevs}) = \begin{cases} 
\{\text{Nothing}, \text{prevs}\} & \text{if any} \begin{cases} 
\Phi \in \{\text{True}, \text{False}\} \\
\Phi \text{ is } (c_1 = c_2) \\
\Phi \text{ is } (c_1 \neq c_2) \\
\Phi \in \text{prevs}
\end{cases} \quad (1a) \\
\langle \Phi, \text{prevs}\rangle & \text{if } \begin{cases} 
\Phi \text{ is an undefined atom, and} \\
\neg \Phi, \Phi \not\in \text{prevs}
\end{cases} \quad (1b) \\
\text{expl}(\Psi, \text{prevs}) & \text{if } \begin{cases} 
\text{def}(\Phi, \text{def}) = \Psi, \Phi \neq \Psi, \text{ and} \\
\neg \Phi, \Phi \not\in \text{prevs}
\end{cases} \quad (1c) \\
\langle \neg \alpha, \text{prevs}_\alpha\rangle & \text{if } \begin{cases} 
\Phi = \neg \Psi, \text{ and} \\
\langle \alpha, \text{prevs}_\alpha\rangle = \text{expl}(\Psi, \text{prevs})
\end{cases} \quad (1d) \\
\text{expl}(\Psi, \text{prevs}) & \text{if } \begin{cases} 
\Phi \text{ is a quantifier and} \\
\text{unroll}(\Phi) = \Psi.
\end{cases} \quad (1e) \\
\text{expl}'(\Psi_1, \text{left}_1, \Psi_2, \text{hasAct}_2, \odot, \text{prevs}) & \text{if } \begin{cases} 
\Phi = \Phi_L \odot \Phi_R, \text{ and} \\
\langle \Psi_1, \text{left}_1, \Psi_2, \text{hasAct}_2\rangle = \text{prio}(\Phi)
\end{cases} \quad (1f) \\
\{\text{Nothing}, \text{prevs}\} & \text{if } \begin{cases} 
\Phi = \Phi_L \odot \Phi_R, \text{ and} \\
\text{Nothing} = \text{prio}(\Phi)
\end{cases} \quad (1g)
\end{cases}$$

where $\odot \in \text{Conn}_B$, and $c_1, c_2 \in \text{Const}$.

**Case 1a** The first case of expl is for boolean constants, equality, inequality and ground atoms in prevs. In this case Nothing will be returned since $\Phi^i$ either has no modification or have already been examined.

**Case 1b** In the second case of expl, an explanation has been identified. The found undefined ground atom will be returned as an explanation.

**Case 1c** In the third case of expl, a defined atom (not in prevs) is reached. Since explanations for the definition of $\Phi^i$ are explanations for $\Phi^i$, simply explaining the definition is sufficient.

**Case 1d** The fourth case of expl is for negations, which are handled in a quite simple way. The direct subformulae of $\Phi$ is simply explained. Thereafter, the explanation is
negated to indicate that its behind a negation. (This case may not work as intended if the negation is not at atom level.)

Case 1e: In the fifth case of expl quantifiers are unrolled. The formula is first unrolled, then the unrolled formula is explained.

Case 1f: The sixth case of expl is used when a binary connective is reached and it may be 1-explainable (since prio does not return Nothing). Here, expl' is used.

Case 1g: The seventh and last case of expl is used when a binary connective is reached but it is not 1-explainable (since prio does return Nothing). Hence, Nothing is returned.

Let \( \text{prio} : \text{Form} \rightarrow (\text{Form} \times \text{Bool} \times \text{Form} \times \text{Bool}) \cup \{\text{Nothing}\} \). The function prio is used to select which branch (i.e. which of the two direct subformulae) to explore first, and if the second branch (or both branches) can be ignored.

From a call to \( \text{prio} \), a 4-tuple is returned where the first element is the priority one formula and second element denotes whether the priority one formula is the left direct subformula of the input formula. The third element is the priority two formula and the fourth and last element denotes whether the priority two formula can affect the value of the input formula. Note, that this last element, the boolean variable hasAct\(_2\), is theoretically not needed, it is just used to practically speed up runtime. If the input formula appears to not be 1-explainable, then Nothing is returned. The function prio can be defined as:

\[
\text{prio}(f_L \odot f_R) = \begin{cases} 
\langle f_L, \text{True}, f_R, \text{True} \rangle & \text{if } \text{any} \\
\langle f_L, \text{True}, f_R, \text{False} \rangle & \text{if } \text{any} \\
\langle f_R, \text{False}, f_L, \text{True} \rangle & \text{if } \text{any} \\
\langle f_R, \text{False}, f_L, \text{False} \rangle & \text{if } \text{any} \\
\{\text{Nothing}\} & \text{otherwise}
\end{cases}
\]

To make \( \text{prio} \) total, let Nothing be returned if a formula without a binary top-level connective is given as argument.

Cases 2b and 2d are reached when no modifications of one of the direct subformulae is a modification of the input formula (assuming none of those modifications affect the
other direct subformula via shared ground atoms). Hence, the other direct subformula is the prioritised formula and the first one is deemed uninteresting.

In Cases \[2a\] and \[2c\] modifications of either direct subformula may be a modification of the input formula. Hence, the prioritisation is quite arbitrary. In the case of equivalence, the false formula gets priority over the true formula if an equivalence between a true and a false formula is given as argument. This is because it is assumed to be more desirable to have an equivalence between two true formulae than between two false formulae. With implication, the right direct subformula (the false one), is prioritised over the left. Likewise, this is because it is assumed that an implication between two true formulae is more desired than an implication between two false formulae. In other cases, simply the left direct subformula is prioritised over the right one.

Finally, Case \[2e\] is reached when the input formula is not 1-explainable (assuming that no explanation is shared between the two direct subformulae).

The second function, expl', takes care of the two direct subformulae of a binary connective, given which one is prioritised first, and whether the other one is relevant. Either none of the two formulae have an explanation, the first has or, the second but not the first. Let:

\[
\text{expl}' : \text{Form} \times \text{Bool} \times \text{Form} \times \text{Bool} \times \text{ConnB} \times \mathcal{P}(\text{Lit}_G) \\
\rightarrow (\text{Lit}_G \cup \{\text{Nothing}\}) \times \mathcal{P}(\text{Lit}_G).
\]

The function expl' can now be defined as:

\[
\text{expl}'(\Phi_1, \Phi_2, \{\text{prevs}\}, \text{tlc}, \text{hasAct}_1) =
\begin{cases}
\langle \text{Nothing}, \text{prevs} \rangle & \text{if any } \langle \text{expl}(\Phi_1, \text{prevs}) = \langle \text{Nothing}, \ast \rangle \text{ and } \\
 & \text{hasAct}_2 = \text{False}
\end{cases}
\]

\[
\begin{cases}
\text{expl}(\Phi_1, \{\text{prevs}\}) = \langle \text{Nothing}, \ast \rangle \\
= \text{expl}(\Phi_2, \{\text{prevs}\}) = \langle \text{Nothing}, \ast \rangle
\end{cases}
\]

\[
\text{expl}'(\Phi_1, \Phi_2, \{\text{prevs}\}, \text{tlc}, \text{hasAct}_2) =
\begin{cases}
\langle \text{expl}(\Phi_1, \text{prevs}), \text{expl}(\Phi_2, \text{prevs}) \rangle & \text{if } \langle \text{expl}(\Phi_1, \text{prevs}) = \langle \text{ex}_1, \text{prevs}_1 \rangle, \\
& \text{ex}_1 \neq \text{Nothing}
\end{cases}
\]

\[
\begin{cases}
\text{expl}(\Phi_1, \{\text{prevs}\}) = \langle \text{expl}(\Phi_2, \text{prevs}) = \langle \text{ex}_2, \text{prevs}_2 \rangle, \\
& \text{ex}_2 \neq \text{Nothing}, \text{ and}
\end{cases}
\]

\[
\text{expl}(\Phi_1, \{\text{prevs}\}) = \langle \text{Nothing}, \ast \rangle
\]
where:

\[
\langle rex_1, rPrevs_1 \rangle = \text{find}(\Phi_2, ex_1, \leftarrow_1, tlc, \text{prevs}_1)
\]
\[
\langle rex_2, rPrevs_2 \rangle = \text{find}(\Phi_1, ex_2, \leftarrow_1, tlc, \text{prevs}_2)
\]

let \( \circ = tlc \) in:

\[
eForm_1 = \begin{cases} 
ex_1 \circ rex_1 & \text{if } \leftarrow_1 \text{ is true} \\
rex_1 \circ ex_1 & \text{otherwise}
\end{cases}
\]

\[
eForm_2 = \begin{cases} 
rex_2 \circ ex_2 & \text{if } \leftarrow_1 \text{ is true} \\
ex_2 \circ rex_2 & \text{otherwise}
\end{cases}
\]

**Case [3a]**  The first case of \( \text{expl} \) is reached when there is no explanation found (or can be found). Here \( \text{Nothing} \) is returned to denote this.

**Case [3b]**  The second case of \( \text{expl} \) is reached when the first formula has an explanation. This explanation and its potential related explanation will be further examined in \( \text{expl}'' \).

**Case [3c]**  The third and last case of \( \text{expl} \) is reached when the first formula does not have any explanation but the second formula does. This explanation and its potential related explanation will be further examined in \( \text{expl}'' \).

The third function, \( \text{expl}'' \), is responsible for accepting or denying potential related explanations. In some cases, ground atoms identified as related explanations in subformulae are not applicable in a superformula; thus, not a related explanation. Let:

\[
\text{expl}'' : \text{Lit}_G \times \text{Lit}_G \times \text{Form} \times \mathcal{P}(\text{Lit}_G) \times \mathcal{P}(\text{Lit}_G) \rightarrow (\text{Lit}_G \cup \{ \text{Nothing} \}) \times \mathcal{P}(\text{Lit}_G).
\]

The function \( \text{expl}'' \) can now be defined:

\[
\text{expl}'' \left( \begin{array}{c}
ex, \\
rex, \\
eForm, \\
\text{prevs,} \\
rPrevs
\end{array} \right) = \begin{cases}
\langle \text{Nothing}, \text{prevs} \rangle & \text{if } e\text{Form is invariable} \\
\langle \text{ex, prevs} \rangle & \text{if } \begin{cases} e\text{Form} \leftrightarrow ex, \text{ or} \\
rex = \text{Nothing}
\end{cases} \\
\langle \neg \text{ex, prevs} \rangle & \text{if } e\text{Form} \leftrightarrow \neg \text{ex} \\
\langle \text{rex, rPrevs} \rangle & \text{if } \begin{cases} rex \not\equiv ex, \text{ and} \\
rex \neq \text{Nothing}
\end{cases}
\end{cases}
\]

If no related explanation was found, \( rex = \text{Nothing} \) or \( rex \approx \text{ex} \). This creates three cases: when the original explanation (\( ex \)) is denied, when \( ex \) is still correct and, when \( ex \) must be negated. The original explanation is denied if \( ex \) is an explanation to both direct subformula to some superformula such that it is not an explanation to this superformula. In this case \( e\text{Form} \) will be invariable, and Case [4a] takes care of this. The original explanation is still correct if \( rex = \text{Nothing} \) or if \( e\text{Form} \leftrightarrow ex \). Thus, \( rex \) can be disregarded. Case [4d] takes care of this. Finally, in Case [4c] \( ex \) needs to be negated since \( e\text{Form} \leftrightarrow \neg \text{ex} \), creating the effect as if \( ex \) was behind a negation. The remaining case, Case [4d] is reached when \( rex \) is a related explanation.
3.2.2 Identifying a related explanation

Every time an explanation is found the algorithm will start looking for a related explanation. This procedure consists of two major steps: finding a candidate formula and finding an explanation to the candidate formula (that is not the original explanation).

Let:

\[
\text{find} : \text{Form} \times \text{Lit}_G \times \text{Bool} \times \text{Conn}_B \times \mathcal{P}(\text{Lit}_G)
\rightarrow (\text{Lit}_G \cup \{\text{Nothing}\}) \times \mathcal{P}(\text{Lit}_G).
\]

The function \(\text{find}\) can now be defined as:

\[
\begin{align*}
\text{find} & \left( \Phi, \text{ex}, \text{left}_\text{ex}, \text{tlc}, \text{prevs} \right) = \\
& \begin{cases} \\
\langle \text{Nothing}, \text{prevs}\rangle & \text{if any} \quad \begin{cases} \\
\Phi \in \{\text{True}, \text{False}\} \\
\Phi \text{ is } (c_1 = c_2) \\
\Phi \text{ is } (c_1 \neq c_2) \\
\Phi \text{ is an undefined atom, and} \\
\Phi \neq \text{ex} \\
\end{cases} \quad (5a) \\
\langle \text{ex}, \text{prevs}\rangle & \text{if } \Phi = \text{ex} \quad (5b) \\
\text{find} & \left( \Psi, \text{ex}, \text{left}_\text{ex}, \text{tlc}, \text{prevs} \right) \text{ if } \begin{cases} \\
\text{def}(\Phi, \text{def}) = \Psi, \text{ and} \\
\Phi \neq \text{ex} \\
\end{cases} \quad (5c) \\
\text{find} & \left( \Psi, \text{ex}, \text{left}_\text{ex}, \text{tlc}, \text{prevs} \right) \text{ if } \begin{cases} \\
\Phi \text{ is a quantifier, and} \\
\text{unroll}(\Phi) = \Psi \\
\end{cases} \quad (5d) \\
\langle \lnot \text{ex}, \text{prevs}_{\lnot \text{ex}}\rangle & \text{if } \begin{cases} \\
\Phi = \lnot \Psi, \text{ and} \\
\Psi, \text{ex}, \text{left}_\text{ex}, \text{tlc}, \text{prevs} \\
\end{cases} \quad (5e) \\
\text{find}' & \left( \Psi_1, \text{left}_1, \Psi_2, \text{prevs} \right) \text{ if } \begin{cases} \\
\Phi = \Phi_L \odot \Phi_R, \text{ and} \\
\langle \Psi_1, \text{left}_1, \Psi_2, \text{hasAct}_2 \rangle = \text{prio}(\Phi) \\
\end{cases} \quad (5f) \\
\langle \text{Nothing}, \text{prevs}\rangle & \text{if } \begin{cases} \\
\Phi = \Phi_L \odot \Phi_R, \text{ and} \\
\text{Nothing} = \text{prio}(\Phi) \\
\end{cases} \quad (5g)
\end{cases}
\end{align*}
\]

where \(\odot \in \text{Conn}_B\), and \(c_1, c_2 \in \text{Const}\).
Case 5a  In the first case of $\text{find}$, $\text{Nothing}$ is returned. This case is reached when $\Phi$ is not further traversable except when $\Phi$ is an undefined ground atom that is identical to $ex$. If $\Phi$ is not identical to $ex$, then the $\alpha'$ from Definition 3.5, the definition of candidate formulae, has not been found.

Case 5b  The second case of $\text{find}$ is reached when $ex$ has been found, meaning that a superformula to $\Phi$ may be candidate formula. In this case the $\alpha'$ from Definition 3.5, the definition of candidate formulae, may have been found.

Case 5c  The third case of $\text{find}$ is used for expanding predicate definitions. A simple recursion on the definition of $\Phi$ is needed.

Case 5d  The fourth case of $\text{find}$, quantifiers are unrolled. The formula is first unrolled, then the unrolled formula is recursed on.

Case 5e  The fifth case of $\text{find}$ is reached when the top-level connective of $\Phi$ is negation. Here, simply recursing on the direct subformulae of $\Phi$ is sufficient.

Case 5f  The sixth case of $\text{find}$ is reached when $\Phi$ is a binary connective and can have an explanation (since $\text{prio}$ does not return $\text{Nothing}$), and hence a related explanation. In such cases $\text{find}'$ is used to further investigate the formula.

Case 5g  The seventh and last case is reached when $\Phi$ does not have an explanation. Hence, no related explanation can exist, and $\text{Nothing}$ is returned.

In $\text{find}'$, candidate formulae are identified. Let:

$$\text{find}' : \text{Form} \times \text{Bool} \times \text{Form} \times \text{Conn}_B \times \text{Lit}_G \times \text{Conn} \times \mathcal{P}(\text{Lit}_G) \rightarrow (\text{Lit}_G \cup \{\text{Nothing}\}) \times \mathcal{P}(\text{Lit}_G).$$

The function $\text{find}'$ can now be defined as follows:

$$\text{find}'(\Phi_1, \Phi_2, \text{left}_1, \text{mlc}, \text{ex}, \text{left}_{\text{ex}}, \text{tlc}, \text{prevs}) = \begin{cases} (\text{Nothing, } \text{prevs}) & \text{if } \text{rex}_1 = \text{rex}_2 = \text{Nothing} \\ \text{find}''(\Phi_1, \text{left}_1, \text{mlc}, \text{ex}, \text{left}_{\text{ex}}, \text{tlc}, \text{prevs}) & \text{if } \text{rex}_1 \neq \text{Nothing} \\ \text{find}''(\Phi_2, \text{ex}, \text{left}_{\text{ex}}, \text{tlc}, \text{prevs}) & \text{if } \text{rex}_1 = \text{Nothing} \end{cases}$$

where:

$$\langle \text{rex}_1, \text{prevs}_1 \rangle = \text{find}(\Phi_1, \text{ex}, \text{left}_{\text{ex}}, \text{tlc}, \text{prevs})$$
$$\langle \text{rex}_2, \text{prevs}_2 \rangle = \text{find}(\Phi_2, \text{ex}, \text{left}_{\text{ex}}, \text{tlc}, \text{prevs})$$

23
and where \( eForm_1 \) and \( eForm_2 \) are formulae consisting of the connective \( tlc \) with \( ex \) as a direct subformula on the left side if \( left_{ex} \) is true, otherwise on the right side. Furthermore, on the other side (e.g. right side if \( left_{ex} \) is true) there is a direct subformula consisting of the connective \( mlc \). In the case of \( eForm_1 \), this connective has a direct subformula \( rex_1 \) to left if \( left_1 \) is true, otherwise to the right, and a ground atom representing \( \Phi_1 \) on the other side. In the case of \( eForm_2 \), the connective \( mlc \) has a direct subformula \( rex_2 \) to right if \( left_1 \) is true, otherwise to the left, and a ground atom representing \( \Phi_1 \) on the other side. The formulae \( eForm_1 \) and \( eForm_2 \) are simplified versions of the main formula. This simplification may cause problems, as discussed in Section 6.1.

Case 6a: The first case of \( find' \) is reached when neither \( \Phi_1 \) nor \( \Phi_2 \) has an instance of \( ex \) reachable or a related explanation to \( ex \). In this case, \textit{Nothing} is returned since nothing can be done.

Case 6b: The second case of \( find' \) is reached when \( \Phi_1 \) has a reachable instance of \( ex \) (i.e. \( \Phi_1 \) may be a candidate formula) or a related explanation to \( ex \). Here, \( find'' \) is used to further investigate \( \Phi_1 \).

Case 6c: Finally, the third and last case of \( find' \) is reached when \( \Phi_2 \) but not \( \Phi_1 \) has a reachable instance of \( ex \) or a related explanation to \( ex \). Here, \( find'' \) is used as well.

Finally, the function \( find'' \) tries to evaluate whether two given explanations are related. Let:

\[
find'' : \text{Lit}_G \times \text{Lit}_G \times \text{Form} \times \text{Form} \times \mathcal{P} (\text{Lit}_G) \times \mathcal{P} (\text{Lit}_G) \rightarrow (\text{Lit}_G \cup \{\text{Nothing}\}) \times \mathcal{P}(\text{Lit}_G).
\]

The function \( find'' \) can now be defined as:

\[
find'' (ex, ex', \Phi, eForm, prevs, prevs') = \begin{cases} 
\exp (\text{ex} \cup \text{prevs'}) & \text{if } \{ \begin{array}{l} eForm_i \text{ is not independent of} \\
\text{its atom representing } \Phi, \text{ and} \\
\text{ex} \approx ex' 
\end{array} \} \\
\langle \text{Nothing, prevs} \rangle & \text{otherwise}
\end{cases}
\]

Case 7a: The first case of \( find'' \), a recursive call to \( \exp \) is returned. The result of this recursive call is a related explanation of \( ex \), if one exists. If one does not exists, the recursive call will return \textit{Nothing}. This case is reached when a candidate formula is found, in this case \( \Phi \) and \( ex' \) are direct subformulae of the candidate formula. Note, that an explanation found here may be denied further up in the main formula.

Case 7b: The second and final case of \( find'' \) is reached when the constructed formula \( eForm \) is independent of the ground atom representing \( \Phi \). Hence, no related explanation can exist to \( ex \); thus, \textit{Nothing} is returned. Case 7b case is also responsible for making sure that a related explanation to another ground atom than \( ex \) is returned correctly (or if it can be returned). However, this case is incomplete; thus, \textit{Nothing} is returned here, as well. More on this is explained in Section 8.1.1.
3.2.3 Concrete example

In this section a concrete example of a system and its requirements will be used to show how the explanation algorithm works. This will help to create a thorough understanding of the algorithm. To keep it short, a simple formula will be used (and rather unrealistic); nevertheless, it will significantly improve the understanding of the overarching principles of the algorithm. Let \( p(a) \) and \( p(b) \) be undefined ground atoms. Let \( sys \) and \( req \) be ground atoms with the definitions
\[
\text{sys} := p(a) \iff p(b) \quad \text{and} \quad \text{req} := p(a).
\]
The formula \( \text{sys} \rightarrow \text{req} \) can be visualised in Figure 3.1. This formula is false only when all ground atoms are false; thus, it is the counterexample to be examined. Let \( i \) be this counterexample; hence, \( i = \emptyset \).

![Figure 3.1: Tree structure of the example](image)

Now, \( \text{expl}(\text{sys} \rightarrow \text{req}, \emptyset) \) will give a list of an explanation and related explanations for \( (\text{sys} \rightarrow \text{req}) \). Since \( \text{sys} \rightarrow \text{req} \) has a binary connective as top-level connective, only Case 1f or Case 1g of \( \text{expl} \) are applicable. To find out exactly which one, \( \text{prio}(\text{sys} \rightarrow \text{req}) \) is evaluated. Since \( \rightarrow \) is the top-level connective and the left direct subformula is true but the right one is false, only Case 2c of \( \text{prio} \) is applicable. Thus, \( \langle \text{req}, \text{False}, \text{sys}, \text{True} \rangle \) is returned, which means that \( \text{req} \) is the priority one formula and is to the right of the connective, and \( \text{sys} \) may have an explanation. Hence, Case 1f must be used and the following can be deduced:

\[
\text{expl}(\text{sys} \rightarrow \text{req}, \emptyset) = \text{expl'} \left( \begin{array}{c}
\text{req}, \\
\text{False}, \\
\text{sys}, \\
\text{True}, \\
\rightarrow, \\
\emptyset
\end{array} \right) \quad (A1)
\]

In order to know how to proceed from Equation (A1), \( \text{expl}(\text{req}, \emptyset) \) needs to be evaluated to find out the applicable case of \( \text{expl}' \). Since, \( \text{req} \) is a defined ground atom not in \( \emptyset \), by Case 1c of \( \text{expl} \), a recursion on its definition is performed. The following is now known:

\[
\text{expl}(\text{req}, \emptyset) = \text{expl}(p(a), \emptyset) = \langle p(a), \emptyset \rangle
\]

Hence, Case 3b is the case of \( \text{expl}' \) that is applicable. However, to evaluate Case 3b, \( \text{find}(\text{sys}, p(a), \text{True}, \rightarrow, \emptyset) \) needs to be evaluated. Let \( \langle \alpha, \text{prevs}_\alpha \rangle \) be the value of that.
Thus:

\[ \langle \alpha, \text{prevs}_\alpha \rangle = \text{find}(\text{sys}, p(a), \text{True}, \to, \emptyset) \]  

(B1)

Since \( \text{sys} \) is a defined ground atom, not in \( \emptyset \), only Case 5c of find is applicable. By proceeding from Equation (B1), the following can be derived:

\[ \langle \alpha, \text{prevs}_\alpha \rangle = \text{find} \left( \begin{array}{c}
\text{sys}, \\
p(a), \\
\text{True}, \\
\to, \\
\emptyset
\end{array} \right) = \text{find} \left( \begin{array}{c}
p(b) \leftrightarrow p(a), \\
p(a), \\
\text{True}, \\
\to, \\
\emptyset
\end{array} \right) \]  

(B2)

Only Case 5f and Case 5g are applicable here. To know how to proceed, it is necessary to use the \( \text{prio} \) function again. Case 2a of \( \text{prio} \) is used since both \( p(a) \) and \( p(b) \) are false, and the following can be derived:

\[ \text{prio}(p(a) \leftrightarrow p(b)) = (p(a), \text{True}, p(b), \text{True}) \]

Thus, only Case 5f of find is applicable and it is possible to proceed from Equation (B2) as follows:

\[ \langle \alpha, \text{prevs}_\alpha \rangle = \text{find} \left( \begin{array}{c}
p(b) \leftrightarrow p(a), \\
p(a), \\
\text{True}, \\
\to, \\
\emptyset
\end{array} \right) = \text{find}' \left( \begin{array}{c}
p(a), \\
\text{True}, \\
p(b), \\
\leftrightarrow, \\
\to, \\
\emptyset
\end{array} \right) \]  

(B3)

Before proceeding from Equation (B3) with the definition of find', it is necessary to evaluate \( \text{find}'(p(a), p(a), \text{False}, \to, \emptyset) \). By Case 5b that will evaluate to \( (p(a), \emptyset) \). Hence, Case 6b of find' will be used when continuing from Equation (B3) giving:

\[ \langle \alpha, \text{prevs}_\alpha \rangle = \text{find'} \left( \begin{array}{c}
p(a), \\
\text{True}, \\
p(b), \\
\leftrightarrow, \\
p(a), \\
\text{False}, \\
\to, \\
\emptyset
\end{array} \right) = \text{find''} \left( \begin{array}{c}
p(a), \\
p(b), \\
p(a) \leftrightarrow q \to p(a), \\
\emptyset, \\
\emptyset
\end{array} \right) \]  

(B4)

where \( q \) is a ground atom representing \( p(b) \). Now, in \( \text{find''} \), since \( p(a) \approx p(a) \) and \( (p(a) \leftrightarrow q \to p(a))^4 \) is not independent of \( q \) (easily checked), Case 7a is applicable. By
proceeding from Equation [B4] with Case 7a of find\(^\prime\), the following can be deduced:

\[
\langle \alpha, \text{prevs}_\alpha \rangle = \text{find}''(p(a), p(a), p(b), p(a) \leftrightarrow q \rightarrow p(a), \emptyset, \emptyset)
\]

\[
= \text{expl}(p(b), [p(a)] \cup \emptyset)
\]

\[
= \text{expl}(p(b), [p(a)])
\]

\[
= \langle p(b), [p(a)] \rangle \quad \text{(by Case 1b)}
\]

(B5)

Thus, by Equation [B5] \(\alpha = p(b)\) and \(\text{prevs}_\alpha = [p(a)]\). Hence, it is now possible to proceed from Equation [A1] since only Case 3b of \(\text{expl}''\) is applicable. Recall that the original explanation is \(p(a)\). The following is derived:

\[
\text{expl}'' \left( \begin{array}{c}
\text{req}, \\
\text{sys}, \\
\rightarrow, \\
\emptyset
\end{array} \right) = \text{expl}'' \left( \begin{array}{c}
p(a), \\
p(b), \\
p(b) \leftrightarrow p(a), \\
\emptyset, \\
[p(a)]
\end{array} \right)
\]

(A2)

Now, there is not much left to do, since \(\text{expl}''\) does not do any recursive calls. Since \(p(a) \not\approx p(b)\), Case 4d of \(\text{expl}''\) is the only case applicable. That gives:

\[
\text{expl}'' \left( \begin{array}{c}
p(a), \\
p(b), \\
p(b) \leftrightarrow p(a), \\
\emptyset, \\
[p(a)]
\end{array} \right) = \langle p(b), [p(a)] \rangle
\]

(A3)

Finally, Equations [A1], [A2] and [A3] gives:

\[
\text{expl}(\text{sys} \rightarrow \text{req}, \emptyset) = \langle p(b), [p(a)] \rangle
\]

Thus, \(p(b)\) is the final explanation and is a related explanation to \(p(a)\) in \((\text{sys} \rightarrow \text{req})\).

4 Implementation

Although this is thesis mainly focuses on theoretical aspects of computer science, implementation of the algorithm was inevitable for efficient testing. To test the algorithm, a functioning infrastructure to run the algorithm on was needed. Everything, including the algorithm, was implemented in Haskell. However, parts of the infrastructure was left to GTO (see Section 1.2 for more on this).

In this section, implementation details of the algorithm as well as an overview of how the infrastructure works will be explained.

The implemented algorithm and the algorithm described in Section 3.2 is, in principle, identical. However, the latter is a little different for the sake of brevity and readability.
4.1 Infrastructure

In the infrastructure, data structures with formulae, interpretations, predicate definitions and variable domains were needed. In order to efficiently use and modify these data structures a parsing module was created, which uses the Haskell library Parsec. The parsing module can parse expressions and statements written with GTO syntax. There are, however, small syntactical discrepancies. Furthermore, the parsing module can read input from both files and user input. The parsing module and the explanation algorithm are both accessed through a user interface.

4.1.1 Data structures

Most things are implemented as described in Section 3. However, in the implementation of the algorithm, evaluated formula trees are used instead of conventional formulae (as described in Definition 2.7).

Definition 4.1. (Evaluated formula tree) Every node in the tree representation of the formula is evaluated. However, the quantifiers are left unevaluated due to the fact that they need to be unrolled for that. Nevertheless, when the algorithm needs to traverse a quantifier, the quantifier is unrolled and translated into an evaluated formula tree.

Evaluated formula trees (and formula trees) are constructed in a manner where every node is one of either:

- Childless nodes:
  - Boolean constants (True, False).
  - Undefined atoms (p(x), q(c), A, φ).
  - Equality / Inequality (b = e, f ≠ a)

- One-child nodes:
  - Negation (¬).
  - Universal quantification (∀) (unevaluated).
  - Existential quantification (∃) (unevaluated).
  - Defined atoms (p(x), q(c), A, φ).

- Two-child nodes
  - Conjunction (∧).
  - Disjunction (∨).
  - Implication (→).
  - Equivalence (↔).

In the implementation, the input formula is converted into an evaluated formula tree when a counterexample is to be examined. And, evaluated formula trees are traversed and used as formulae are in Section 3.2.
4.1.2 Parsing

In this section, how the syntactical and semantic analysis in the parsing module of the infrastructure works will be shown. Backus-Naur form will be used to express the formal grammars. First let:

\[
\text{Char ::= } [a–z] \mid [A–Z] \quad \text{Alphanum ::= Char | [0–9]}
\]

Closed formulae (\(\text{Form}\)) and possibly open formulae (\(\text{Form}_O\)) are constructed as follows:

\[
\begin{align*}
\text{Form} & ::= \text{Equiv} \quad \text{Form}_O ::= \text{Form} \\
\text{Equiv} & ::= \text{Impl} \ \text{Equiv}' \\
\text{Impl} & ::= \text{Disj} \ \text{Impl}' \\
\text{Disj} & ::= \text{Conj} \ \text{Disj}' \\
\text{Conj} & ::= \text{Neg} \ \text{Conj}' \\
\text{Neg} & ::= \text{“~” T} \mid T \\
\text{T} & ::= \text{Exist} \mid \text{Univ} \mid \text{Rel} \\
& \quad \mid \text{Atom} \mid \text{“(” Form “)”} \\
\text{Exist} & ::= \text{“SOME” Name “(” Form “)”} \\
\text{Univ} & ::= \text{“ALL” Name “(” Form “)”} \\
\text{Rel} & ::= \text{Equal} \mid \text{Ineq} \\
\text{Equal} & ::= \text{Name “=” Name} \\
\text{Ineq} & ::= \text{Name “<>” Name} \\
\text{Prop} & ::= \text{“TRUE”} \mid \text{“FALSE”} \mid \text{Name} \\
\text{Pred} & ::= \text{Name “(” Args “)”} \\
\text{Args} & ::= \text{Name “,” Args} \mid \text{Name} \\
\text{Name} & ::= \text{Char Alphanum}^* \\
\end{align*}
\]

such that Definition 2.7 is followed for all \(\text{Form}\) and \(\text{Form}_O\).

Furthermore, statements can be either constant declarations, variable declarations, variable domain declarations, predicate and atom declarations, predicate definitions, or truth statements. Statements (\(\text{Stmt}\)) are constructed in the following manner:

\[
\begin{align*}
\text{Stmt} & ::= \text{Consts} \mid \text{Vars} \mid \text{Types} \mid \text{Preds} \mid \text{Defs} \mid \text{Facts} \\
\text{Variable domain declarations (Types)} & \text{ are constructed and defined as follows:}
\end{align*}
\]

\[
\begin{align*}
\text{Types} & ::= \text{“TYPES” Name Types'} \\
\text{Types'} & ::= \text{“,” Name Types'} \mid \text{“;”} \\
\end{align*}
\]

such that Definition 2.3 is followed. Variable domains must be declared via these statements in order to be used.
Constants (\textit{Consts}) can only be introduced by including them into some existing variable domain, and their declarations are constructed and defined as follows:

\begin{align*}
\text{Consts} & ::= \ "\text{CONSt}\_\" \text{\underline{Name} Consts}' \\
\text{Consts}' & ::= \ "\," \text{\underline{Name} Consts}' \mid \ "\," \text{\underline{Name} \ ";"}
\end{align*}

such that Definitions \ref{def:const} and \ref{def:const2} are followed. Before constants can be used, they must be declared via these statements.

Variables (\textit{Vars}) can only be introduced by setting them to range over some variable domain, and their declarations are constructed and defined as follows:

\begin{align*}
\text{Vars} & ::= \ "\text{VAR}\_\" \text{\underline{Name} Vars}' \\
\text{Vars}' & ::= \ "\," \text{\underline{Name} Vars}' \mid \ "\," \text{\underline{Name} \ ";"}
\end{align*}

such that Definitions \ref{def:var} and \ref{def:var2} are followed. Variables must be declared via these statements before they can be used.

Predicate and atom declarations (\textit{Preds}) are constructed and defined as follows:

\begin{align*}
\text{Preds} & ::= \ "\text{PRED}\_\" \text{\underline{Atom} Preds}' \\
\text{Preds}' & ::= \ "\," \text{\underline{Atom} Preds}' \mid \ "\," \text{\underline{Atom} \ ";"}
\end{align*}

such that Definition \ref{def:pred} is followed (e.g. no \text{Atom} is either \text{TRUE} or \text{FALSE}). Atoms must be declared via these statements before they can be used.

Predicate definitions (\textit{Defs}) are constructed and defined as follows:

\begin{align*}
\text{Defs} & ::= \text{\underline{Atom} \ "=\" \text{\underline{Form}_O} \ ";"}
\end{align*}

such that Definition \ref{def:def} is followed.

Lastly, the interpretation can be partially (or fully) fixed by forcing a subset of atoms to be true. Truth statements (\textit{Facts}) are constructed and defined as follows:

\begin{align*}
\text{Facts} & ::= \ "\text{FACTS}\_\" \text{\underline{Atom} Facts}' \\
\text{Facts}' & ::= \ "\," \text{\underline{Atom} Facts}' \mid \ "\," \text{\underline{Atom} \ ";"}
\end{align*}

such that every \text{Atom} is a declared atom.
5 Results

The algorithm has been tested on several examples. The most interesting ones will be presented here along with the results of each example. The algorithm will in most cases find the same explanations as the GTO toolset; however, in some cases better explanations is provided.

First, one example that the presented algorithm gives better results than the GTO toolset does, will be shown. Assume that the formula \((b \leftrightarrow (c \leftrightarrow ((d \leftrightarrow a) \rightarrow d)) \rightarrow c)) \rightarrow b\) is the input formula, where \(a, b, c, d \in \text{Atom}_G\). This formula can be more easily visualised in Figure 5.1. One counterexample to this is when all ground atoms are false. In this case the algorithm will produce an expected result, namely: \(a, \{d, c, b\}\). This result is expected since: all explanations are listed, they are all related explanations of each other and, they are ordered by depth of occurrence in the formula. It first identifies \(b\) as an explanation. Thereafter, a candidate formula \(b \leftrightarrow (c \leftrightarrow ((d \leftrightarrow a) \rightarrow d)) \rightarrow c\) is found and a related explanation is sought out in the right direct subformula. Now, the formula is \((c \leftrightarrow ((d \leftrightarrow a) \rightarrow d)) \rightarrow c\), \(c\) is found as an explanation and the pattern continues. This, however, is not a commonly structured input formula, but the algorithm give similar results on similar structures.

![Figure 5.1: Formula tree of the first example.](image)

A common way of structuring a system is to have a number of properties (which could include all kinds of connectives and quantifiers) in conjunction or disjunction on the left-hand side of an implication. The following formula is an example of a common structure:

\[
\frac{\text{system}}{((\text{property}_0 \lor \text{property}_1') \land \text{property}_1 \land \text{property}_2)} \rightarrow \text{requirements}
\]

This formula will be transformed to the tree in Figure 5.2. Assuming that \(\text{property}_2\) is not an undefined atom, the marked node (thick dotted circle) in Figure 5.2 will not be fully traversable regarding related explanations (see Section 8.1.1). This means that
no connection to any of these properties, from requirements, can be done. This, of course, severely limits the number of formulae where related explanations inside the system can be found.

The first example that was investigated in this project was the battery example. The battery example is a rather small system compared to numerous practical systems. Nevertheless, one of the larger examples used to test this algorithm.

Intuitively, it describes a system with two lights, two light switches and three batteries. Each light is either lit or unlit, each switch is either on or off, and each battery is either charged or discharged. The first light is connected to the first battery via the first light switch and the last light is connected to the two other batteries via the remaining light switch. A light will be lit if and only if its light switch is on and at least one of its batteries is charged. Each light switch and battery can arbitrarily assume either state. The requirements of this system states that every light is correct. A light is correct if and only if, whenever there is sufficient battery power, the light is lit if all of its light switches are on. Furthermore, if there are at most one discharged battery, then there is sufficient battery power.

The GTO code for this battery example can be found in Listing 5.1. When running system \( \rightarrow \) requirements this translates to the formula shown in tree form in Figure 5.3. Assume that a SAT solver then gives the following counterexample (which is a legitimate counterexample and the counterexample outputted when running this example in the GTO toolset):

\[
i = \{ \text{correct}(L2), \quad \text{charged}(B2), \quad \text{charged}(B3), \quad \text{system}, \quad \text{on}(S1), \quad \text{sufficientbatteries}, \quad \text{connected}(S1,L1), \quad \text{connected}(S2,L2) \}
\]

The described algorithm here returns \( \text{lit}(L1) \), [], this means that no related explanation to \( \text{lit}(L1) \) was found. The explanation \( \text{lit}(L1) \) is, however, a correct explanation; running \( \text{system & lit}(L1) \rightarrow \text{requirements} \) shows that there are, in fact, no counterexamples. The question is whether no related explanation exists, or whether the

---

4 Running in the GTO toolset.
algorithm is incorrect or incomplete. By doing some manual work, a related explanation can be found. The formula system has a subformula lit(L1) ↔ charged(B1) ∧ on(S1). In this subformula, according to the counterexample, on(S1) is True whilst charged(B1) is False. Thus, a related explanation because running system & charged(B1) → requirements will show that the formula is valid (hence the whole formula has the same truth value regardless of the value of lit(L1)). It is at the marked (dotted) node in Figure 5.3 this problem occurs. The cause of this is discussed in Section 8.1.1.

When running the algorithm on some examples, one big potential error was discovered. In the early stages of the algorithm, it was decided that if a modification of a formula with a binary top-level connective required modifications of both its direct subformulae, and hence was 2-explainable (assuming no shared explanations), the algorithm would just arbitrarily choose one of the direct subformulae to recurse on. Furthermore, this incomplete explanation would be treated as a regular explanation. This gave some correct results and, sometimes, required the algorithm to run twice to solve a problem. However, when identifying candidate formulae, a distinction between regular explanations and the aforementioned kind of explanations was not made. Thus, a possibility for identified related explanations to not be actual related explanations were present.

---

5 Running in the GTO toolset.
It was never proven to cause errors, but following the adage “better safe than sorry”, this was removed. Instead of returning a possibly incorrect explanation, Nothing is now returned.

6 Discussion and reflection

The errors and (non critical) flaws of the algorithm and the implementation of the algorithm will be discussed in this section. Moreover, other, earlier, approaches will be discussed as well. Lastly, a reflection of the work in this thesis is provided.
6.1 Unproven techniques

The presented algorithm uses simplified formulae when identifying candidate formulae. It is, however, not proved whether this simplification will create an unwanted behaviour. Despite this, no tests have shown any falsely identified related explanations. This may be a result of the fact that the algorithm is incomplete; that the incompleteness is preventing any (or a large portion of) falsely identified related explanations, resulting from this simplification, to be returned. More on this is described in Section 8.1.1.

6.2 Errors

The most prominent known error in the algorithm is the exception of the negation push-down procedure. Recall that: \( a \rightarrow b \Leftrightarrow \neg a \vee b \). By this property, the negation in the implication should be pushed down. However, this is not the case because of the desire to preserve the implications, since implications and disjunctions are handled rather differently. This retains the special case of implications but introduces a bug. This bug can occur when, in the aforementioned formula, \( b \) has no explanation but \( a \) has. Thus, an explanation from \( a \) will be returned. This may cause problems due to the fact that \( a \) is behind a negation not taken into account. Hence, candidate formulae will most likely be improperly identified. Therefore, the algorithm will, in some niche cases, produce erroneous explanations.

A possible solution to this exception is to start a negation push-down whenever the right hand side of an implication is searched for an explanation.

There are very likely more errors and bugs that has not been discovered yet, as in most projects of this size.

6.3 Flaws

In addition to errors there are also some properties that are unnecessary or create runtime (and manual) overhead. These properties will be presented here.

6.3.1 Result print-out

Aside from the program’s quite unreadable print-out\(^6\), the print-out gives no real context as to where the explanations are located or why they were identified. These details could give additional information that is essential to find some errors. One atom (that is the returned explanation) may occur many times throughout the whole formula and knowing which one that was identified by the algorithm may make a big difference. Different instances of the atom identified by the algorithm may give different preferred solutions. Knowing why would also be helpful.

\(^6\) In the program’s current state, solely the returned explanation and the prevs list is printed.
6.3.2 Flaws in the infrastructure

In addition to the user interface being very clunky, predicate names cannot start with a \texttt{A} or \texttt{S}. To the parser, that will be confused with the keyword \texttt{ALL} or \texttt{SOME}. Moreover, \texttt{ALL} and \texttt{SOME} always need parentheses; thus, \texttt{ALL x p(x)} is not allowed but \texttt{ALL x (p(x))} is, for instance. Because of this problem quantifiers can't be chained without blocks of messy parentheses. For example, \texttt{SOME x SOME y (p(x,y))} is not allowed but \texttt{SOME x (SOME y (p(x,y)))} is allowed.

6.3.3 Redundant searches

In many cases, subformulae of the main formula will be unnecessarily searched for explanations or related explanations. Notwithstanding that these subformulae does not have explanations identifiable by the algorithm, they will be searched. Either the subformula is not 1-explainable or it is 1-explainable only by a shared explanation. The problem with shared explanations is explained in Section 6.3.4. More often than not, certain constant-time checks can be performed to avoid this overhead searching.

6.3.4 Missed explanations

By the greediness of \texttt{prio}, shared explanations are ignored. When a modification of a formula requires modifications of both its direct subformulae, the formula is considered not 1-explainable by the algorithm. In some cases, however, both the direct subformulae may share an explanation, and hence a modification of that ground atom is a modification of both direct subformulae, and as a result a modification of the formula itself. In \texttt{prio}, this is, as stated, ignored and \texttt{Nothing} will always be returned.

This will, of course, limit the number of explanations and related explanations that can be found by the algorithm. It is unclear to what degree this is a limitation and whether shared explanations is an uncommon phenomenon in practice.

6.4 Other approaches

The first approach to this problem was not making use of backtracking. This first approach revisited the whole formula top-down every time an undefined ground atom had been identified as an explanation. The revisitation of the formula would then be a complete search for a candidate formula (at this time the definition of candidate formulae was still vague). This idea was abandoned because of two main reasons. First, the whole formula had to be at hand at all times. This created problems because a functional (as in programming) approach did not coincide with this practice; it was more suited for imperative style. And because a major part of the infrastructure had already been implemented in Haskell at that point, I decided to look for another approach. Secondly, the number of times some subformulae were visited could easily explode, figuratively. To solve this, certain subformula had to be marked in some manner. In the final approach presented in this thesis, the number of visitations should be much lower on average that
what it would have been with this previous approach. Although, there is no evidence to support this claim.

Another, perhaps less complex approach than the one presented in this thesis, is built on the notion of first finding all explanations to the main formula, then connecting them in chains of related explanations. This could be done by, for instance, comparing evaluations of the main formula in the four possible interpretations obtained by combining the two modifications of each pair of explanations. This approach may be less time efficient than the one presented in this thesis, but may also be easier to comprehend, thus making it easier to define and implement.

6.5 Reflection

To attack this problem I went for a mostly abstract and general approach. This made me focus on the whole picture instead of the most common cases. Nevertheless, in the end, the result is not as useless as I sometimes thought it would become. If only the unsolved problem regarding related explanations (described in Section 8.1.1) is solved (even partially) this algorithm could produce an array of explanations for the majority of common cases. Time limit was also a factor, there were certain errors and flaws that I wanted to take a closer look at but there was not enough time; I had to prioritise. And I believe that I made the correct prioritisations.

Creating and implementing the infrastructure was the easiest part. It was the theoretical aspect of the algorithm, especially defining related explanations and candidate formulae, that took the most time and thinking.

I implemented everything in Haskell which I still feel was a good choice. I learnt a lot from doing this, in both theoretical and practical aspects of computer science.

7 Related work

This thesis has a rather strong connection to the GTO toolset by L.-H. Eriksson. For things covering the relation with the GTO toolset, see Section 1.2.

M. Siegel, A. Maggiore and C. Pichler describes a software that seems to have a similar function as the algorithm described in this thesis [5]. More detailed information on how the procedure works is, however, left out. They describe two complementary approaches: structural assertion debugging and temporal fan-in analysis. Keep in mind that this work is essentially working with temporal logic. This work also focuses on more concrete situations. The structural assertion debugging process is very similar to the approach of finding an explanation that is presented in this thesis. In contrast, the structural assertion debugging takes time into account, it annotates all objects at every clock-cycle with the corresponding value from the counterexample. Thereafter, it finds the sub-expressions that causes the assertion to fail, essentially finding an explanation. The temporal fan-in analysis basically finds what certain values (those found by structural assertion debugging) depends on, where and how it was changed. This is not as similar to the notion of finding related explanations as structural assertion is to finding
explanations. That has much to do with the fact that, again, time is a factor. This method can identify both design bugs and property errors. They state that these techniques enable efficient working with complex assertions.

A. Süßflow, G. Fey, R. Bloem and R. Drechsler describes a completely different method for error explanations in the context of verification of logic circuits [6]. They propose a debugging framework that exploits something called unsatisfiable cores. Unsatisfiable cores are unsatisfiable subsets of clauses of an unsatisfiable formula in CNF\(^7\) (note that not all clauses are necessarily unsatisfiable). They say that breaking an unsatisfiable core is an alteration of its clauses such that it is satisfiable. The procedure described in the paper is to first calculate an unsatisfiable core, thereafter breaking it, and repeating until the problem no longer is unsatisfiable. Lastly, standard SAT-based debugging is used.

8 Conclusions and future work

An algorithm for explaining counterexamples have been constructed and described in Section 3.2. As shown in Section 5, the algorithm can, and will in some cases, find better explanations than the GTO toolset. However, details of the process (where and why explanations were found in the formula) is not as thorough as those provided by the GTO toolset. One big reason to this fact is that the presented algorithm needs negations to be pushed down to atom level. Hence, the original structure of the formula is not retained and, as a result, providing details becomes problematic. In some rare cases, the algorithm will most likely produce erroneous explanations. This is due to the exception of the negation push-down explained in Section 6.2. Furthermore, some hypothesised errors have been proposed, which may or may not be of real concern.

The runtime complexity of this algorithm is currently not known, and large enough examples have not been used to make impractical runtimes appear.

8.1 Future work

There are plenty of interesting work and research left to be done. For example, what the implications of handling and explaining 2-explainable (and beyond) formulae is and if it could be solved by using non-atom explanations. Furthermore, how to properly and efficiently identify shared explanations is also an unsolved problem.

Another interesting topic is finding a way to not need negation push-down. This would make details of explanations much easier to provide.

Two of the most interesting topics will be more thoroughly explained in the following sections.

\(^7\) Conjunctive normal form, a formula in CNF is a conjunction of clauses where each clause is a disjunction of atoms (or its negation).
8.1.1 Forwarding of related explanations

Determining whether a related explanation is one to a direct superformula or if it can be altered so that it becomes one, was intended to be handled in a separate case of find. However, due to time limitations, how to do it was never studied. This special case had to be bundled in with Case 7b and simply return Nothing. It would have been a great expansion of the algorithms current explanation capacity if this case would have been implemented.

In order to explain this concept, let the atom \( \alpha \) be an explanation to \( \Phi^i \), where \( \Phi \) is a formula and \( i \) an interpretation. The result from a recursive call of find on any direct subformulae (of a binary connective) is either Nothing; the explanation \( \alpha \), meaning that this is a candidate formula or; another (related) explanation. In the last case, where another explanation is returned, the question is whether this new explanation can be returned as is, needs to be altered or, simply is not an explanation to the superformula. Since this case is extremely common, solving this problem is an important step towards making this algorithm generally useful.

This problem is likely related to the problem of the unproven simplification used to identify candidate formulae (explained in Section 6.1). Proving whether the simplification is valid or showing how the related explanations need to be altered, when returned, to keep the simplification valid, will give huge insights to how the algorithm can be improved.

8.1.2 Broadening candidate formulae

By extending the definition of candidate formulae (Definition 3.5) to allow \( \alpha' \) be a formula such that \( \alpha \) is an explanation to \( (\alpha')^i \), the number of candidate formulae may tremendously increase. The problem is how the algorithm can be extended to utilise such a definition of candidate formulae. In the presented algorithm, this case is bundled with Case 7b of find.

References


