Verified Rewriting of MFOTL Formulas

Dawit Tirore

January 2020

Contents

1 Introduction

Runtime monitoring is a technique to verify properties about the execution of a program. This is useful in scenarios where there is a low tolerance for errors. Runtime monitoring works by setting up a monitor that continuously receives inputs from the monitored application. The monitor is configured to evaluate this stream of inputs, also called a *trace*, against a policy written in a policy specification language. If the monitored application was an HTTP-server, the policy might specify the following: If a request event is observed, a response event must be observed within 1 second.

Metric first-order temporal logic (MFOTL) is one such example of a policy specification language. It is based on the first-order-logic, which it extends with future and past temporal operators that reference either forward or backward in the trace relative to the current index. Additionally, all indices in the trace are annotated with a time-stamp, allowing one to specify time-windows. From now on we will refer to a policy as a *formula*, since MFOTL policies are logical formulas. Basin et al. [1] describe an efficient monitoring algorithm which has been implemented in the monitoring tool MonPoly [2] which can monitor a fragment of MFOTL. Scneider et al. [3] formally verified this monitoring algorithm (called *VeriMon*) using the Isabelle proof-assistant. Isabelle is an interactive theorem-prover, mechanically checking human written proofs while assisting the user in constructing these proofs. All inferences must pass through a small well understood trusted kernel, providing the highest level of trustworthiness. VeriMon has been integrated into MonPoly, replacing an unverified implementation of the monitoring algorithm. Even though the core monitoring algorithm of MonPoly has been verified, the tool still relies on some unverified procedures such as the rewriting of formulas. The rewriting of formulas is necessary because VeriMon monitors formulas efficiently by using finite tables. Storing the intermediate results of evaluated sub-formulas with finite tables does however inherit the problem of *domain independence* also seen in relational calculus [4]. To see the problem, consider the formula below where R, P and Q are finite relations:

$$R(x, y) \wedge (\neg P(x) \vee Q(y))$$

The formula as a whole is only satisfied for a finite set of assignments for both x and y and is therefore called *domain independent*, as the assignments depend on R and not the domain x and y is defined over. On the other hand, the right conjunct does not restrict either of the variables to values present in the tables P and Q and is therefore *domain dependent*. Domain dependence is therefore a problem when one is dealing with finite tables. Although the example formula as a whole is domain independent, it can not be monitored directly by MonPoly's monitoring algorithm because it contains a domain dependent sub-formula. It is undecidable whether a formula is domain independent but a decidable subset of the domain independent formulas can be defined syntactically. These formulas are called *range restricted*. In addition to being domain independent, the

formula above is also range restricted. Evident by this example formula, a range restricted formula can contain a domain dependent sub-formula if its variables are restricted in other parts of the formula. Only a subset of the range restricted formulas can be monitored and these are the *safe* formulas. A safe formula can be evaluated in a bottom-up manner with finite tables because intermediate results are ensured to be finite. Although this is not the case for a range restricted formula, some range restricted formulas can become safe formulas by rewriting them. The range restricted formula above becomes safe if $R(x, y)$ is distributed:

$$(R(x, y) \wedge \neg P(x)) \vee (R(x, y) \wedge Q(y))$$

The goal of rewriting formulas is to maximize the number of range restricted formulas that can be rewritten to safe formulas. MonPoly currently performs rewriting with an unverified rewrite function. The goal in this project is to implement and verify a rewrite function in Isabelle, using the unverified rewrite function as a starting point. The rewrite rules that the unverified function implements are based on equivalences from the doctoral thesis Theory and applications of runtime monitoring metric first-order temporal logic [5]. From now on this will simply be referred to as Müller's thesis. The equivalences from Müller's thesis have not been verified. This project therefore also includes their formal verification in Isabelle.

Eight equivalences were found to be unsound, some of which are directly used in the current unverified rewrite function of MonPoly. The source of the unsoundness is the same for the eight equivalences and they share the same solution. This will be addressed in Section 2. The structure of the report is the following. This introduction will continue into a subsection showing the syntax and semantics of MFOTL and will end with a precise definition of range restriction. Section 2 will show the equivalences that the rewrite function is based on and describe relevant aspects of how they were verified. Section 3 will describe implementation details of the rewrite function and Section 4 will go through the main challenges in proving its correctness.

1.1 Metric First Order Temporal Logic

Below the syntax and semantics of MFOTL is given. The paragraphs **Syntax** and **Definition 2.4.16** are taken literally from Müller's Thesis. The definition number is provided as it appears in the thesis.
In the paragraph **Syntax**, C is a set of constant symbols, R a set of relation symbols and a is an arity function defined over the relation symbols. In the paragraph **Definition 2.4.16**, the interval set $\mathbb{I}$ is defined as $[a, b] \in \mathbb{I}$ if $a \in \mathbb{N}$, $b \in \mathbb{N} \cup \infty$ and $a \leq b$.

Syntax Let $S = (C, R, a)$ be a signature and V denote a countably infinite set of variables, where we assume $V \cap R = \emptyset$ and $V \cap C = \emptyset$.

Definition 2.4.16 *The (MFOTL) formulas over S are inductively defined.*
(i) For $t,' t \in V \cup C$, $(t \approx t')$and $(t \prec t')$ are formulas.
(ii) For $r \in R$ and $t_1, ..., t_{a(r)} \in V \cup C$, $r(t_1, ..., t_{a(r)})$ is a formula.
(iii) For $x \in V$ if ϕ_1 and ϕ_2 are formulas then $(\neg\phi_1), (\phi_1 \wedge \phi_2)$ and $\exists x.\ \phi_1$ are formulas.
(iv) For $I \in \mathbb{I}$, if ϕ_1 and ϕ_2 are formulas then $\bullet_I \phi_1$ and $\bigcirc_I \phi_1$, $\phi_1 \mathcal{S}_I\ \phi_2$ and $\phi_1 \mathcal{U}_I \phi_2$ are formulas.

Additionally for the formula $\phi_1 \mathcal{U}_{[a,b]} \phi_2$, the interval $[a, b]$ is required to be future bound so it must hold that $b \in \mathbb{N}$ (i.e. b cannot be ∞).

Satisfiability The satisfiability of a formula is defined by the sat predicate seen on Figure 1. The sat predicate is defined in the Formula theory, which is part of the Isabelle entry MFODL_Monitor_Optimized which in turn is part of the Archive of Formal Proofs, an Isabelle proof library. A theory in Isabelle is equivalent to a module in most programming languages and an entry is a collection of theories. Looking at the definition of the sat predicate, the type of σ is trace, which is a tuple of two infinite sequences, the first being an infinite sequence of sets of events (accessed by $\Gamma\ \sigma$) and the second being an infinite sequence of timestamps (accessed by $\tau\sigma$). The sat predicate is defined for MFOTL extended with aggregation, let-bindings and regular expressions. These extensions will generally be glanced over in this report as no equivalence uses them. V should be ignored as it is related to the semantics of let-bindings. The type of v is env, which is a list of event data and it is used as a partial function on the variables. The syntax in 2.4.16 is mapped to the sat predicate by (i) referring to the cases where the top-constructor of a formula is Eq, Less or LessEq. (ii) refers to Pred. (iii) refers to Neg, And and Exists. (iv) refers to Prev, Next, Since and Until.

```
qualified fun sat :: "trace ⇒ (name ⇀ nat ⇒ event_data list set) ⇒
                      env ⇒ nat ⇒ formula ⇒ bool" where
  "sat σ V v i (Pred r ts) = (case V r of
      None ⇒ (r, map (eval_trm v) ts) ∈ Γ σ i
    | Some X ⇒ map (eval_trm v) ts ∈ X i)"
| "sat σ V v i (Let p b φ ψ) =
    sat σ (V(p ↦ λi. {v. length v = nfv φ - b ∧ (∃zs. length zs = b ∧
                                                 sat σ V (zs @ v) i φ)})) v i ψ"
| "sat σ V v i (Eq t1 t2) = (eval_trm v t1 = eval_trm v t2)"
| "sat σ V v i (Less t1 t2) = (eval_trm v t1 < eval_trm v t2)"
| "sat σ V v i (LessEq t1 t2) = (eval_trm v t1 ≤ eval_trm v t2)"
| "sat σ V v i (Neg φ) = (¬ sat σ V v i φ)"
| "sat σ V v i (Or φ ψ) = (sat σ V v i φ ∨ sat σ V v i ψ)"
| "sat σ V v i (And φ ψ) = (sat σ V v i φ ∧ sat σ V v i ψ)"
| "sat σ V v i (Ands l) = (∀φ ∈ set l. sat σ V v i φ)"
| "sat σ V v i (Exists φ) = (∃z. sat σ V (z # v) i φ)"
| "sat σ V v i (Agg y ω b f φ) =
    (let M = {(x, ecard Zs) | x Zs. Zs = {zs. length zs = b ∧
                                          sat σ V (zs @ v) i φ ∧
                                          eval_trm (zs @ v) f = x} ∧
                                          Zs ≠ {}}
    in (M = {} ⟶ fv φ ⊆ {0..<b}) ∧ v ! y = eval_agg_op ω M)"
| "sat σ V v i (Prev I φ) = (case i of 0 ⇒ False
                                    | Suc j ⇒ mem (τ σ i - τ σ j) I ∧ sat σ V v j φ)"
| "sat σ V v i (Next I φ) = (mem (τ σ (Suc i) - τ σ i) I ∧ sat σ V v (Suc i) φ)"
| "sat σ V v i (Since φ I ψ) = (∃j≤i. mem (τ σ i - τ σ j) I ∧ sat σ V v j ψ ∧
                                   (∀k ∈ {j <.. i}. sat σ V v k φ))"
| "sat σ V v i (Until φ I ψ) = (∃j≥i. mem (τ σ j - τ σ i) I ∧ sat σ V v j ψ ∧
                                   (∀k ∈ {i ..< j}. sat σ V v k φ))"
| "sat σ V v i (MatchP I r) = (∃j≤i. mem (τ σ i - τ σ j) I ∧ Regex.match (sat σ V v) r j i)"
| "sat σ V v i (MatchF I r) = (∃j≥i. mem (τ σ j - τ σ i) I ∧ Regex.match (sat σ V v) r i j)"
```

Figure 1: Sat predicate

Abbreviations Some relevant abbreviations given in Müller's thesis are:

$$
\begin{aligned}
(\Diamond_I \phi) &:= (true\ \mathcal{U}_I\ \phi) \\
(\blacklozenge_I \phi) &:= (true\ \mathcal{S}_I\ \phi) \\
(\Box_I \phi) &:= (\neg(\Diamond_I(\neg\phi)) \\
(\blacksquare_I \phi) &:= (\neg(\blacklozenge_I(\neg\phi)) \\
\beta \mathcal{R}_I \gamma &:= \neg(\neg\beta \mathcal{U}_I\ \neg\gamma) \\
\beta \mathcal{T}_I \gamma &:= \neg(\neg\beta \mathcal{S}_I\ \neg\gamma)
\end{aligned}
$$

Intuitively $\alpha \mathcal{U}_I \beta$ means that α is satisfied now and at all future indices until β is satisfied where as $\alpha \mathcal{S}_I \beta$ means that β is satisfied at some index in the past and α is satisfied for all succeding indices up until and including the current index.

The abbreviation ($\Diamond_I \phi$), means "sometime in the future", while ($\blacklozenge_I \phi$) means "sometime in the past". On the other hand ($\Box_I \phi$) means "at all times in the future" and similarly for ($\blacksquare_I \phi$) "at all times in the past".
The operators $\mathcal{R}$ (Release) and $\mathcal{T}$ (Trigger) will be referred to as dual temporal operators, as they can be seen as the duals of $\mathcal{U}$ (Until) and $\mathcal{S}$ (Since).

Strictness In Müller's thesis a strict version of all temporal operators (including those that are abbreviations) is derived by replacing $\mathcal{S}$ and $\mathcal{U}$ with $\dot{\mathcal{S}}$ and $\dot{\mathcal{U}}$, whose semantics differ by the latter having strict inequalities between indices i and j. Intuitively strictness enforces the left operand of the temporal operator to be satisfied for a sequence of indices.

Range restriction The set of range restricted variables in a formula α is $RR(\alpha)$, defined in Figure 2 where x, x' range over V and c over C.

$$
\mathsf{RR}(\alpha) := \begin{cases}
\{x\} & \textit{if } \alpha = x \approx c,\ \alpha = c \approx x, \textit{or } \alpha = x \prec c, \\
\{t_i \mid t_i \in \mathsf{V} \wedge 1 \leq i \leq a(r)\} & \textit{if } \alpha = r(t_1, \ldots, t_{a(r)}), \\
\emptyset & \textit{if } \alpha = \neg\beta, \alpha = c \prec x, \alpha = x \approx x', \\
 & \quad \alpha = x \prec x', \textit{or } \alpha = x \prec c, \\
\mathsf{RR}(\beta) \setminus \{x\} & \textit{if } \alpha = \exists x.\, \beta \textit{ and } x \in \mathsf{RR}(\beta), \\
\mathsf{RR}(\beta) & \textit{if } \alpha = \otimes\beta \textit{ with } \otimes \in \{\bullet_I, \dot{\blacklozenge}_I, \blacklozenge_I, \dot{\blacksquare}_I, \blacksquare_I\} \\
 & \qquad \cup \{\bigcirc_I, \dot{\Diamond}_I, \Diamond_I, \dot{\Box}_I, \Box_I\}, \\
\mathsf{RR}(\beta) \cup \mathsf{RR}(\gamma) & \textit{if } \alpha = \beta \wedge \gamma,\ \alpha = \beta \dot{\mathcal{S}}_I \gamma, \textit{ or } \alpha = \beta \dot{\mathcal{U}}_I \gamma, \\
\mathsf{RR}(\beta) \cap \mathsf{RR}(\gamma) & \textit{if } \alpha = \beta \vee \gamma, \alpha = \beta \dot{\mathcal{T}}_I \gamma, \textit{ or } \alpha = \beta \dot{\mathcal{R}}_I \gamma, \\
\mathsf{RR}(\gamma) & \textit{if } \alpha = \beta\, \mathcal{S}_I\, \gamma, \alpha = \beta\, \mathcal{U}_I\, \gamma, \alpha = \beta\, \mathcal{T}_I\, \gamma, \\
 & \quad \textit{or } \alpha = \beta\, \mathcal{R}_I\, \gamma.
\end{cases}
$$

Figure 2: Definition of Range Restriction. The figure is taken from Müller's thesis.

2 Soundness of Equivalences

In this section, a set of equivalences originally presented in Müller's thesis, will be shown. These are later used in proving the correctness of the rewrite function. Some of the equivalences are unsound, in which case the corrections also will be shown. Some of the equivalences presented in Müller's thesis use a strict variant of the (dual) temporal operators, which have not been defined in the Formula theory. Strictness has in this project been enforced in another way which will be described. The Formula theory also represents variables with De Bruijn indices and the consequence of this in proving equivalences will also be shown. The section will end with a walk-through of the proof for an equivalence involving a dual temporal operator.

2.1 Overview

We divide the equivalences that have been proved into two sets. The first set is called the auxiliary equivalences and can be seen in Figure 3. This set is not directly used in the rewrite function but shows how operands of temporal operators can be distributed from left to right and right to left. Moreover it shows how to push a conjunct inside the arguments of a temporal operator.

1. $\alpha \dot{\mathcal{U}}_I \beta \equiv \alpha \dot{\mathcal{U}}_I (\dot{\blacklozenge}_I \alpha \wedge \beta)$
2. $\alpha \dot{\mathcal{U}}_I \beta \equiv (\alpha \wedge \dot{\Diamond}_I \beta) \dot{\mathcal{U}}_I \beta$
3. $\alpha \dot{\mathcal{S}}_I \beta \equiv \alpha \dot{\mathcal{S}}_I (\dot{\Diamond}_I \alpha \wedge \beta)$
4. $\alpha \dot{\mathcal{S}}_I \beta \equiv (\alpha \wedge \dot{\blacklozenge}_I \beta) \dot{\mathcal{S}}_I \beta$
5. $\alpha \wedge (\beta \dot{\mathcal{U}}_I \gamma) \equiv \alpha \wedge (\beta \dot{\mathcal{U}}_I (\dot{\blacklozenge}_I \alpha \wedge \gamma))$
6. $\alpha \wedge (\beta \dot{\mathcal{U}}_I \gamma) \equiv \alpha \wedge ((\beta \wedge \blacklozenge_I \alpha) \dot{\mathcal{U}}_I \gamma)$
7. $\alpha \wedge (\beta \dot{\mathcal{S}}_I \gamma) \equiv \alpha \wedge (\beta \dot{\mathcal{S}}_I (\dot{\Diamond}_I \alpha \wedge \gamma))$
8. $\alpha \wedge (\beta \dot{\mathcal{S}}_I \gamma) \equiv \alpha \wedge ((\beta \wedge \Diamond_I \alpha) \dot{\mathcal{S}}_I \gamma)$

Figure 3: Auxiliary equivalences

The main equivalences which are directly used as rewrite rules can be seen in Figure 4. From now on, rather than writing for example second auxiliary equivalence and third main equivalence, I will instead write AE2 and ME3. For each of the auxiliary equivalences, there is a related main equivalence. Some equivalences such as AE6 and ME11 are the same.[1] Other equivalences such as AE1 and ME7 differ only because the latter requires an argument of the temporal operator to be a conjunction. The reason for requiring an argument to be a conjunction is heuristical and will be explained in Section 3. Finally, since strict

[1] AE6 and ME11 are the same equivalence after conjunction in the left temporal operator of ME11 has been commuted.

operators are not used, rule 13 through 17 duplicates rule 18 through 21.

1.	$\alpha \wedge (\beta \vee \gamma)$	$\mapsto$	$(\alpha \wedge \beta) \vee (\alpha \wedge \gamma)$	(push range restriction into $\vee$)
2.	$\alpha \wedge (\beta \dot{\mathcal{R}}_I \gamma)$	$\mapsto$	$\alpha \wedge ((\beta \wedge \blacklozenge_I \alpha) \dot{\mathcal{R}}_I (\gamma \wedge \blacklozenge_I \alpha)$	(push range restriction into $\dot{\mathcal{R}}_I$)
3.	$\alpha \wedge (\beta \dot{\mathcal{T}}_I \gamma)$	$\mapsto$	$\alpha \wedge ((\beta \wedge \dot{\Diamond}_I \alpha) \dot{\mathcal{T}}_I (\gamma \wedge \dot{\Diamond}_I \alpha)$	(push range restriction into $\dot{\mathcal{T}}_I$)
4.	$\alpha \wedge \exists x.\beta$	$\mapsto$	$\alpha \wedge \exists x.(\alpha \wedge \beta)$	(push range restriction into $\exists$)
5.	$\alpha \wedge \neg\beta$	$\mapsto$	$\alpha \wedge \neg(\alpha \wedge \beta)$	(push range restriction into $\neg$)
6.	$(\alpha \wedge \gamma) \dot{\mathcal{S}}_I \beta$	$\mapsto$	$(\alpha \wedge \gamma) \dot{\mathcal{S}}_I (\dot{\Diamond}_I \alpha \wedge \beta)$	(distribute from left to right in $\dot{\mathcal{S}}_I$)
7.	$(\alpha \wedge \gamma) \dot{\mathcal{U}}_I \beta$	$\mapsto$	$(\alpha \wedge \gamma) \dot{\mathcal{U}}_I (\blacklozenge_I \alpha \wedge \beta)$	(distribute from left to right in $\dot{\mathcal{U}}_I$)
8.	$\beta \dot{\mathcal{S}}_I (\alpha \wedge \gamma)$	$\mapsto$	$(\blacklozenge_I \alpha \wedge \beta) \dot{\mathcal{S}}_I (\alpha \wedge \gamma)$	(distribute from right to left in $\dot{\mathcal{S}}_I$)
9.	$\beta \dot{\mathcal{U}}_I (\alpha \wedge \gamma)$	$\mapsto$	$(\dot{\Diamond}_I \alpha \wedge \beta) \dot{\mathcal{U}}_I (\alpha \wedge \gamma)$	(distribute from right to left in $\dot{\mathcal{U}}_I$)
10.	$\alpha \wedge (\beta \dot{\mathcal{S}}_I \gamma)$	$\mapsto$	$\alpha \wedge ((\Diamond_I \alpha \wedge \beta) \dot{\mathcal{S}}_I \gamma)$	(push into $\dot{\mathcal{S}}_I$, left side)
11.	$\alpha \wedge (\beta \dot{\mathcal{U}}_I \gamma)$	$\mapsto$	$\alpha \wedge ((\blacklozenge_I \alpha \wedge \beta) \dot{\mathcal{U}}_I \gamma)$	(push into $\dot{\mathcal{U}}_I$, left side)
12.	$\alpha \wedge (\gamma \dot{\mathcal{S}}_I \beta)$	$\mapsto$	$\alpha \wedge (\gamma \dot{\mathcal{S}}_I (\dot{\Diamond}_I \alpha \wedge \beta)$	(push into $\dot{\mathcal{S}}_I$, right side)
13.	$\alpha \wedge (\gamma \dot{\mathcal{U}}_I \beta)$	$\mapsto$	$\alpha \wedge (\gamma \dot{\mathcal{U}}_I (\blacklozenge_I \alpha \wedge \beta)$	(push into $\dot{\mathcal{U}}_I$, right side)
14.	$\alpha \wedge \blacklozenge_I \beta$	$\mapsto$	$\alpha \wedge \blacklozenge_I (\dot{\Diamond}_I \alpha \wedge \beta)$	(push into $\blacklozenge_I$)
15.	$\alpha \wedge \dot{\Diamond}_I \beta$	$\mapsto$	$\alpha \wedge \dot{\Diamond}_I (\blacklozenge_I \alpha \wedge \beta)$	(push into $\dot{\Diamond}_I$)
16.	$\alpha \wedge \blacksquare_I \beta$	$\mapsto$	$\alpha \wedge \blacksquare_I (\dot{\Diamond}_I \alpha \wedge \beta)$	(push into $\blacksquare_I$)
17.	$\alpha \wedge \dot{\Box}_I \beta$	$\mapsto$	$\alpha \wedge \dot{\Box}_I (\blacklozenge_I \alpha \wedge \beta)$	(push into $\dot{\Box}_I$)
18.	$\alpha \wedge \blacklozenge_I \beta$	$\mapsto$	$\alpha \wedge \blacklozenge_I (\Diamond_I \alpha \wedge \beta)$	(push into $\blacklozenge_I$)
19.	$\alpha \wedge \Diamond_I \beta$	$\mapsto$	$\alpha \wedge \Diamond_I (\blacklozenge_I \alpha \wedge \beta)$	(push into $\Diamond_I$)
20.	$\alpha \wedge \blacksquare_I \beta$	$\mapsto$	$\alpha \wedge \blacksquare_I (\Diamond_I \alpha \wedge \beta)$	(push into $\blacksquare_I$)
21.	$\alpha \wedge \Box_I \beta$	$\mapsto$	$\alpha \wedge \Box_I (\blacklozenge_I \alpha \wedge \beta)$	(push into $\Box_I$)
22.	$\alpha \wedge \bullet_I \beta$	$\mapsto$	$\alpha \wedge \bullet_I (\bigcirc_I \alpha \wedge \beta)$	(push into $\bullet_I$)
23.	$\alpha \wedge \bigcirc_I \beta$	$\mapsto$	$\alpha \wedge \bigcirc_I (\bullet_I \alpha \wedge \beta)$	(push into $\bigcirc_I$)

Figure 4: Main equivalences. Taken from Müller's thesis, where $\mapsto$ is used rather than $\equiv$ because they are presented as rewrite rules.

2.2 Strictness of Temporal Operators

All unsound equivalences use strict temporal operators. The Formula theory, which defines the syntax and semantics of MFOTL only contains the non-strict temporal operators. Therefore, before the unsoundness of the incorrect equivalences can be shown, their non-strict representations in Isabelle must be ad-

dressed.

Most of the equivalences that use strict temporal operators, remain sound when strictness is dropped and in this case the strict operator is simply replaced by its non-strict counterpart. The arguments of temporal operators will also be referred to as *temporal operands*. The remaining equivalences where dropping strictness would be unsound are those that distribute temporal operands from left to right. One example is $\alpha \dot{\mathcal{U}}_I \beta \equiv \alpha \dot{\mathcal{U}}_I (\blacklozenge_I \alpha \wedge \beta)$ which is only sound in the strict case. Intuitively this is because, strictness forces α to be satisfied at some index within the interval. An analogy can be drawn to regular expressions, where $\alpha \dot{\mathcal{U}}_I \beta$ is similar to $\alpha^+ \ \beta$ and $\alpha \mathcal{U}_I \beta$ is similar to $\alpha^* \ \beta$. It is therefore only in the strict case where α is guaranteed to have been satisfied before β within the interval I.
For these kinds of equivalences, strictness is enforced by assuming 0 is not in the interval I. Intuitively this means that time has increased between the current index and the time where β is satisfied. Stated more formally, the timestamps in τ is a monotonically increasing sequence. Therefore for positions i, j where $i \leq j$, if $\tau_j - \tau_i \neq 0$, then $i < j$. It can be noted that requiring time to have increased, is more strict than requiring $i < j$, because τ is not necessarily *strictly* increasing. Figure 5 shows a small Isabelle proof, showing that when assuming I excludes 0, the non-strict Until operator and the strict one are equivalent. Here `mem` is the membership function on intervals.

```
lemma strict_until:
    "excl_zero I ⟹ Formula.sat σ V v i (Formula.Until φ I ψ) =
    (∃j>i. mem (τ σ j - τ σ i) I ∧
           Formula.sat σ V v j ψ ∧
           (∀k ∈ {i..< j}. Formula.sat σ V v k φ))"
  using le_eq_less_or_eq by auto
```

Figure 5: Simulating strictness in Isabelle

The *excl_zero I* assumption is used in AE1, AE3, ME6 and ME7. As an example AE1 is shown in Figure 6.

```
lemma equiv_1: "excl_zero I ⟹
Formula.sat σ V v i (Formula.Until α I β) =
Formula.sat σ V v i (Formula.Until α I (Formula.And (diamond_b I α) β))"
  by fastforce
```

Figure 6: Representing AE1 in Isabelle

2.3 Unsound Equivalences and Their Corrections

The unsound equivalences are AE2, AE4, ME2, ME3 and ME8-ME11. The unsoundness stems from either distributing temporal operands from right to left or pushing a conjunct into a left temporal operand. The first reason is exemplified by AE4: $\alpha \mathcal{S}_I \beta \equiv (\alpha \wedge \blacklozenge \beta) \mathcal{S}_I \beta$ and the second reason is exemplified by ME10: $\alpha \wedge (\beta \mathcal{S}_I \gamma) \equiv \alpha \wedge ((\diamond_I \alpha \wedge \beta) \mathcal{S}_I \gamma)$.[2] To see why they are unsound, consider the left-hand-side of ME10 where $\alpha = S(y)$, $\beta = \neg P(y) \wedge Q(x)$, $\gamma = R(x, y)$ and $I = [10, 12]$.

$$\phi := S(y) \wedge ((\neg P(y)) \wedge Q(x)) \mathcal{S}_{[10,12]} R(x, y))$$

And the sequence of time stamps and sets of events

```
@0  R(1,2)
@8  Q(1)
@10 S(2) Q(1)
@20
```

Using zero-based indexing to refer to a line, the n-th line contains the n-th timestamp followed by the n-th set of events. By evaluating ϕ at index 2 where $\tau_2 = 10$, the left conjunct is satisfied by $y = 2$. The right conjunct is also satisfied at index 2 by $x = 1$ and $y = 2$. This is because the right temporal operator is satisfied at index 0, the left temporal operator is satisfied at both index 1 and 2 and $\tau_3 - \tau_0 = 10$ is in the interval $[10, 12]$.

Now consider the right-hand-side of the equivalence

$$\psi := Sy \wedge ((\diamond_{[10,12]} Sy \wedge (\neg Py \wedge Qx)) \mathcal{S}_{[10,12]} R(x, y))$$

For ψ to be satisfied at index 2, $(\diamond_{[10,12]} Sy \wedge (\neg Py \wedge Qx))$ must be satisfied at indices 1 and 2, but it is satisfied at neither of the indices. This is because only index 3 is in the interval of $\diamond_{[10,12]} S(y)$ and since index 3 contains no events, it does not satisfy $S(y)$. To solve this problem we must replace the lower bound of the interval by 0. The corrected ME10 proved in Isabelle can be seen in Figure 7. Here `diamond_w` is $\diamond$ and `init_int`$([a, b]) = [0, b]$.

[2] These two equivalences have been written non-strictly, because strictness can be dropped if temporal operands are not distributed left to right, as described in Section 2.2

```
lemma sat_main_10:
"Formula.sat σ V v i (Formula.And α (Formula.Since β I γ)) =
 Formula.sat σ V v i (Formula.And α
      (Formula.Since (Formula.And (diamond_w (init_int I) α) β) I γ))"
(is "?L = ?R")
proof(rule iffI)
  assume L:?L
  then obtain j where j: "j≤i" "mem (τ σ i - τ σ j) I"
                         "Formula.sat σ V v j γ"
                         "(∀k∈{j<..i}. Formula.sat σ V v i α ∧
                                       Formula.sat σ V v k β)" by auto
  then have "∀k∈{j<..i}. mem (τ σ i - τ σ k) (init_int I)"
    using nat_less_mem_of_init[OF _ j(2)] by fastforce
  then show ?R using L j by fastforce
qed auto
```

Figure 7: Proof of corrected ME10

$$\alpha \wedge (\beta \mathcal{S}_I \gamma) \equiv \alpha \wedge ((\Diamond_{init\_int(I)} \alpha \wedge \beta) \mathcal{S}_I \gamma)$$

Note the use of an additional lemma `nat_less_mem_of_init` in Figure 7, near the bottom. This lemma is used in all the corrected equivalences. Its definition and proof can be seen on Figure 8.

```
lemma nat_less_mem_of_init: "⋀i j:: nat. k ∈ {i..j} ∧ k' ∈{i..j} ⟹
                                         mem (τ σ j - τ σ i) I ⟹
                                         mem (τ σ k - τ σ k') (init_int I)"
proof -
  fix i j :: nat assume A:"k ∈ {i..j} ∧ k' ∈{i..j}" "mem (τ σ j - τ σ i) I"
  then have "(τ σ k - τ σ k') ≤ (τ σ j - τ σ i)" using nat_less_than_range by auto
  then show " mem (τ σ k - τ σ k') (init_int I)" using A(2) mem_of_init by blast
qed
```

Figure 8: Lemma capturing why the corrected lemma is sound

The lemma states that if $\tau_j - \tau_i \in I$ then $\forall k, k' \in [i, j]. \tau_k - \tau_{k'} \in init\_int(I)$. This is clearly not the case if $init\_int(I)$ is replaced by I. Proving `nat_less_mem_of_init` relies on the fact that $k, k' \in [i, j] \implies k - k' \leq j - i$. By the monotone property of time stamps, this is lifted to $\tau_k - \tau_{k'} \leq \tau_j - \tau_i$. The lemma further relies on the intuitive fact that for a natural number n, if $n \in I$, then $\forall n' \leq n.\ n' \in init\_int(I)$. These two facts combined is enough to prove the lemma.

2.4 De Bruijn Indices and Existential Variables

Another source of unsoundness in the main equivalences is ME4 which does not ensure capture avoidance. To ensure capture avoidance, it should be formulated

as $x \notin fv(\alpha) \implies \alpha \wedge \exists x.\beta \equiv \alpha \wedge \exists x\ (\alpha \wedge \beta)$. The Formula theory represents MFOTL variables with De Bruijn indices, which makes the proof significantly more challenging. De Bruijn indices is a way to represent variables used for example in lambda calculus that avoids the naming of variables. A binder is a term that introduces a bound variable, such as lambda abstraction or existential quantification. A De Bruijn index represents a variable occurrence by a natural number indicating the number of binders that were introduced between this variable occurrence and its own binder. In MFOTL both aggregations and the existential quantifier is a binder, but we shall focus on the existential quantifier. Take an example with two existential variables and a free variable, $\exists\exists P(0, 1, 2)$. Here 0 is bound to the inner-most quantifer, 1 is bound to the other and 2 is a free variable because the index is too large to refer to a binder inside its scope. Remembering the event-data list in the sat predicate, it is because variables are represented as De Bruijn indices, that their mapping to event-data can be represented by a list.

It is known that $x \notin fv(\alpha) \implies \alpha \equiv \exists x.\alpha$. When using De Bruijn indices however, all the free variables must be incremented to ensure capture avoidance. Related to this is that the unverified rewrite function, although not using De Bruijn indices, has suffered the error of applying this equivalence without ensuring capture-avoidance, resulting in unsoundness. Incrementing the free variables in a formula, i.e. *shifting* the formula, is implemented in Isabelle by the function $shiftI :: nat \to formula \to formula$, where the first argument is the number of binders in scope. To illustrate its definition, the case for Exists is shown in Figure 9.

```
fun shiftTI :: "nat ⇒ Formula.trm ⇒ Formula.trm" where
 "shiftTI k (Formula.Var i) = (if i < k then Formula.Var i
                                        else Formula.Var (i + 1))"
primrec shiftI :: "nat ⇒ Formula.formula ⇒ Formula.formula" where
  "shiftI k (Formula.Pred r ts) = Formula.Pred r (map (shiftTI k) ts)"
| "shiftI k (Formula.Exists a) = Formula.Exists (shiftI (Suc k) a)"
```

Figure 9: Excerpt of the shift function, incrementing the free variables in a formula. (Top) The case where a term is a variable. (Bottom) The Exists and Pred case where r is a relation and ts is a list of terms.

To prove the corrected ME4, a separate lemma is needed which addresses the satisfiability of a shifted formula. Naturally, if the free variables of a formula are shifted, the event-data list that contains their mappings must be shifted as well. Shifting the list is achieved by prefixing it with an arbitrary value z. The equivalence is seen by the lemma in Figure 10. Here the lemma is defined for when the first argument of shiftI is 0. `sat_shift` is a generalized lemma where the first argument may be any arbitrary natural number. `sat_shift` was stated in this general form to allow the proof to be by induction over the formulas. The proof of `sat_shift_z` is simply an instantiation of `sat_shift`.

```
lemma sat_shift_z:
"Formula.sat σ V (z # v) i (shiftI 0 φ) = Formula.sat σ V v i φ"
  using sat_shift[of "[]"] by auto
```

Figure 10: Equivalence between a formula and the shifted formula

The proof of ME4 instantiates `sat_shift` the same way. This can be seen in Figure 11.

```
lemma sat_main_4: "Formula.sat σ V v i (Formula.And α (Formula.Exists β)) =
                   Formula.sat σ V v i (Formula.Exists (Formula.And (shift α) β ))"
  using sat_shift[of "[]"] by auto
```

Figure 11: Proof for ME4

2.5 Proving Equivalences Using Dual Temporal Operators

Some of the more complicated equivalences to prove were ME2 and ME3 involving the dual temporal operators. To see why, compare the semantics of Until with Release in Figure 12.

```
lemma sat_until_def:
"Formula.sat σ V v i (Formula.Until φ I ψ) =
 (∃j≥i. mem (τ σ j - τ σ i) I ∧
        Formula.sat σ V v j ψ ∧
        (∀k ∈ {i ..< j}. Formula.sat σ V v k φ))" by auto

lemma sat_release_abb:
"Formula.sat σ V v i (release φ I ψ) =
 (∀j≥i. (¬(mem (τ σ j - τ σ i) I) ∨
        Formula.sat σ V v j ψ ∨
        (∃k ∈ {i ..< j}. Formula.sat σ V v k φ)))" by auto
```

Figure 12: Semantics of Until (Top) and dual semantics of Until (Bottom)

Here `release` is the definition of $\mathcal{R}$. Comparing the two, the semantics of Until has an existentially bound quantifier and a sub-expression consisting of three conjuncts. The dual semantics of Release on the other hand has a universally bound quantifier and a sub-formula of three disjuncts.

Proving equivalences where the right-hand-side consists of conjuncts is easier to prove in both directions. From left to right, each conjunct can be derived individually. From right to left, each conjunct becomes a separate assumption. For disjuncts on the other hand, one must consider all the cases that would satisfy them. To see this, a walkthrough for the proof of ME2 is now given.

Proving ME2 The lemma to show is stated below. The last line abbreviates the left-hand-side by the schematic variable $?L$ and right-hand-side by $?R$.

```
lemma sat_main_2:
"Formula.sat σ V v i (Formula.And α (release β I γ)) =
 Formula.sat σ V v i (Formula.And α (release (Formula.And β (diamond_b (init_int I) α))
                                             I
                                             (Formula.And γ (diamond_b I α))))"
(is "?L = ?R" )
```

The first rule to apply is iff-introduction, stating that showing an equivalence, reduces to showing $?L \implies ?R$ and $?R \implies ?L$. This is seen by the first line in the image below. $?R \implies ?L$ trivially holds, so we proceed to prove $?L \implies ?R$ by now assuming $?L$. We simplify this assumption into a list of two facts, which we name `split_A`. In the second fact, $\bigwedge j.$ introduces j as a fixed but arbitrary natural number, within the scope of the binder. To easily refer to the second fact of split_A it will be abbreviated here in the report as $\bigwedge j.\ j \geq i \implies a \vee b \vee c$.

```
proof(rule iffI)
  assume ass: "?L"
  then have split_A: "Formula.sat σ V v i α"
                     "(⋀j. j≥i ⟹ (¬ mem (τ σ j - τ σ i) I) ∨
                                  Formula.sat σ V v j γ ∨
                                  (∃k∈{i..<j}. Formula.sat σ V v k β))"
    by auto
```

From these two facts, we now want to derive that α can be propagated into both operands of Release. This is seen below by stating an intermediate lemma, similar to the second fact in split_A, except that the second and third disjunct now include the distributed α. Again for reference, this will here in the report be abbreviated as $\bigwedge j.\ j \geq i \implies a \vee (b \wedge b') \vee (c \wedge c')$

```
then have "(⋀j. j≥i ⟹ ¬ mem (τ σ j - τ σ i) I ∨
                        (Formula.sat σ V v j γ ∧
                           (∃ja≤j. mem (τ σ j - τ σ ja) I)) ∨
                        (∃k∈{i..<j}. (Formula.sat σ V v k β ∧
                            (∃j≤k. mem (τ σ k - τ σ j) (init_int I) ∧
                                   Formula.sat σ V v j α ))))"
```

We show $\bigwedge j.\ j \geq i \implies a \vee (b \wedge b') \vee (c \wedge c')$ by creating a new proof context, seen by the first line in the image below. In this proof context we fix j to an arbitrary value greater or equal to i. In this proof context $\bigwedge j.\ j \geq i \implies a \vee b \vee c$ yields $a \vee b \vee c$.

```
  proof -
  fix j assume le:"j≥i"
  then have "¬ mem (τ σ j - τ σ i) I ∨
             Formula.sat σ V v j γ ∨
             (∃k∈{i..<j}. Formula.sat σ V v k β)"
    using ass by auto
```

We now state three cases, preparing us for a proof of the main lemma $\bigwedge j.\ j \geq i \implies a \vee (b \wedge b') \vee (c \wedge c')$ by cases. Again, in the current proof context $\bigwedge j.\ j \geq i \implies a \vee (b \wedge b') \vee (c \wedge c')$ yields $a \vee (b \wedge b') \vee (c \wedge c')$. Intuitively

there is a case for when each of the disjuncts are true. For case (b) and (c) we additionally need the interval membership of $\tau_j - \tau_i$.

```
then consider (a) "¬ mem (τ σ j - τ σ i) I" |
              (b) "(Formula.sat σ V v j γ) ∧  mem (τ σ j - τ σ i) I" |
              (c) "(∃k∈{i..<j}. Formula.sat σ V v k β)"
                  "mem (τ σ j - τ σ i) I" by auto

then show "(¬ mem (τ σ j - τ σ i) I ∨
             (Formula.sat σ V v j γ ∧
                (∃ja≤j. mem (τ σ j - τ σ ja) I)) ∨
             (∃k∈{i..<j}. (Formula.sat σ V v k β ∧
                 (∃j≤k. mem (τ σ k - τ σ j) (init_int I) ∧
                        Formula.sat σ V v j α ))))"
```

The proof is over the cases (a), (b) and (c) and each case is simple to prove. Note that (c) needs to appeal to `nat_less_mem_of_init` because of $init\_int(I)$.

```
proof(cases)
    case a
    then show ?thesis by auto
  next
    case b
    then show ?thesis using le by auto
  next
    case c
    then show ?thesis
      using split_A(1) nat_less_mem_of_init[OF _ c(2)]
      by auto
  qed
```

This proves $?L \implies ?R$ and as mentioned, $?R \implies ?L$ trivially holds, so Isabelle proves this directly. With both implications shown, the equivalence has been proved.

3 Implementing the Rewrite Function

This section addresses two essential concerns about the implementation of the rewrite function. These are: When rewrite rules should be applied and how they can be implemented in an Isabelle-friendly way.

3.1 The Propagate Condition

The purpose of the rewrite function is to enlarge the set of safe formulas by propagating range restrictions heuristically. The heuristics are guided by a

propagate condition related to each main equivalence, introduced in Müller's thesis, which pushes α into β if there is a range restricted variable x in α (i.e. $x \in RR(\alpha)$) which is free but not range-restricted in β (i.e. $x \in FV(\beta) \setminus RR(\beta)$). In Müller's thesis the sub-formulas in the main equivalences have been named intentionally such that it is assumed that α contains the range restricted x, which is unrestricted in β. For the equivalences where the distributed α is part of a conjunction, our rewrite function additionally tries to distribute the other conjunct if the propagate condition fails for α. This can be seen in Figure 13 on the next page. This explains why some of the main equivalences are nearly identical to some of the auxiliary equivalences except for the former requiring an argument of a temporal operator to be a conjunction. The benefit is that if one of the conjuncts contain a range restriction, only a conjunct and not an entire conjunction needs to be pushed into β. It avoids unnecessarily bloating up a formula. Its drawback is that there are cases where it would be useful to apply the less restrictive rewrites as they appear in the auxiliary equivalences.

The propagate condition for ME1 differs from the remaining propagate conditions. The original propagate condition was *prop_cond* α β, which in this project was improved to *prop_cond* α $\beta \vee$ *prop_cond* α γ. To see their difference consider instantiating the left-hand-side of ME1 to $R(x, y) \wedge (P(x) \vee \neg Q(y))$, making $\alpha = R(x, y)$, $\beta = P(x)$ and $\gamma = \neg Q(y)$. The original propagate condition *prop_cond* α β would not trigger as β is already range restricted. However the improved propagate condition *prop_cond* α $\beta \vee$ *prop_cond* α γ would trigger as y is not range restricted in γ. A further improvement that could have been made on this propagate condition was discussed with the project advisors but was not been implemented due to time constraints. This improvement was the propagate condition *prop_cond* α $(\beta \vee \gamma)$. To see why this is the most useful propagate condition of the three, consider instansiating the left-hand-side of ME1 to $R(x, y) \wedge (P(x) \vee Q(y))$, making $\alpha = R(x, y)$, $\beta = P(x)$ and $\gamma = Q(y)$. Only the propagate condition *prop_cond* α $(\beta \vee \gamma)$ would trigger the rewrite as it correctly considers the intersection of the range restricted variables in β and γ, whose set is empty. This is the best propagate condition of the three because it aligns with the definition of range restriction for disjunction (i.e. $RR(\beta \vee \gamma) = RR(\beta) \cap RR(\gamma)$).

3.2 Embedings and Projections

A function in Isabelle containing overlapping pattern-match cases, is internally parsed into an equivalent function of a larger set of non-overlapping cases. Quoting an example found in the Isabelle documentation [6], consider the type P3 modeling booleans but also allowing the unknown value, represented by X.

```
P3 = T | F | X
```

Implementing conjunction for this datatype with overlapping cases could then look like

```
And T p = p
And p T = p
And p F = F
And F p = F
And X X = X
```

In Isabelle this would internally be represented by the non-overlapping patterns

```
And T p = p
And F T = F
And X T = X
And F F = F
And X F = F
And F X = F
And X X = X
```

It can be seen the number of cases has increased. The rewrite function pattern matches against the left-hand-side of the equivalences which are highly overlapping when desugared and additionally contains repeating patterns which is not allowed. Repeating patterns is fixed by handling them in the same case, but representing highly overlapping cases in a non-overlapping way severely increases the number of cases. This is due in part because the formula type contains 17 constructors. The many cases results in an internal representation too large to be processed by Isabelle in reasonable time. Attempting to process the function using a laptop with with a four-core 1.60GHz CPU, resulted in Isabelle running out of memory. One of the sources for the combinatorial explosion can be attributed to two specific overlapping cases in particular, the first matching the left-hand-side of ME6 and the second matching the left-hand-size of ME8.[3] ME6 matches the left operand of Since to a conjunct while ME8 does the same to the right operand. The non-overlapping representation of these two cases requires 17^4 cases. Adding to this, that many of the temporal operators seen in the main equivalences are not constructors, but abbreviations of larger formulas, it is clear that something must be done to limit the combinatorial explosion of cases.

To solve this, two additional datatypes were defined, tformula and rformula along with projection and embedding functions, allowing conversion between the three types. The definition of tformula contains equivalent constructors for all constructors in formula and additionally has constructors for $\diamond$, $\blacklozenge$, $\Box$ and $\blacksquare$. rformula contains equivalent constructors for all constructors in tformula and additionally has constructors for $\mathcal{R}$ and $\mathcal{T}$. Embedding a formula into its corresponding rformula, is then done by embedding the formula into the temporary form of tformula and embedding this formula into an rformula. The reason a formula is not directly embedded as an rformula, is that the patterns in the embedding function would face the exact same problem of combinatorial explosion

[3] Using ME7 and ME9 is equally problematic

that one wants to avoid.

As a consequence of embedding formulas, shiftI has also been defined on the level of rformulas as shiftI_r. It was attempted to implement this function by composing the shiftI function with the embed and project functions, but it made some proofs more challenging.
Another consequence of embedding formulas is that satisfiability also must be defined at the level of rformulas. This is however simply done by composing embed and project with the sat predicate. The rformula level sat predicate is called `rsat`.

Embedding formulas significantly reduces the number of cases, but it does not address the fact that the non-overlapping representation of the left-hand-side P6 and P8 as patterns, adds 17^4 cases. This problem is solved by reordering sub-formulas, as described next.

3.3 Reordering Subformulas

For most of the main equivalences, the left-hand-side is a conjunction and as conjunction is commutative, the equivalence would remain sound after commuting the conjuncts. This will more generally be referred to as reordering the sub-formulas, the reason for which will be seen shortly. Motivated by the goal of maximizing completeness, while avoiding to duplicate the existing conjunction patterns in reordered form, the rewrite function takes an additional argument besides the formula to be rewritten. This additional argument has the type `argpos` containing the two constructors `Same` and `Swapped`. This serves as a flag indicating whether the input formula should be rewritten as is or if the sub-formulas should be reordered first. This serves two purposes. The first is to increase completeness by allowing a formula that is a conjunction which didn't match any rule to be reordered before trying again. The second purpose was to allow the left-hand-sides of ME6 and ME8 to be matched by the same case pattern, letting the flag indicate which rule to trigger. This solves the problem with combinatorial explosion of cases. Figure 13 shows how the rewrite function is defined for the case of ME6 and ME8. It can be seen that first ME6 is attempted to be applied where α is propagated and if that fails, γ is attempted to be propagated. If that fails, ME8 is attempted to be applied in a similar fashion. The last line `fix_r (RSince l I R) t`, reorders the sub-formulas if t is swapped, otherwise it returns the formula as is.

4 Correctness of the Rewrite Function

Earlier, implementation details of rewrite function were adressed. In this section, the main challenges in proving its correctness will be shown. The function is defined recursively, but it can not be inferred by Isabelle automatically that

```
(*6 first, then 8*)
| "rewrite (RSince (RAnd α γ) I β) t =
            (let l = rewrite (RAnd α γ) Same;
                 r =      if t=Same ∧ excl_zero I ∧ prop_cond α β
                          then rewrite (RAnd (RDiamondW I α) β) Same
                     else if t=Same ∧ excl_zero I ∧ prop_cond γ β
                          then rewrite (RAnd (RDiamondW I γ) β) Same
                     else if t=Swapped ∧ finite_int I ∧ prop_cond α β
                          then rewrite (RAnd (RDiamondB (init_int I) α) β) Same
                     else if t=Swapped ∧ finite_int I ∧ prop_cond γ β
                          then rewrite (RAnd (RDiamondB (init_int I) γ) β) Same
                     else rewrite β Same
             in fix_r (RSince l I r) t)"
```

Figure 13: An excerpt of the rewrite function applying ME6 and ME8.

the arguments to the recursive calls are smaller than the input, with respect to a measure that will be introduced shortly. Termination therefore has to be proved manually. Another challenge is that the rewrites differ from the equivalences from Section 2, by also containing recursive calls. It will be shown how to reduce these rewrites to the original equivalences by using substitution contexts.

4.1 Termination

When defining functions in Isabelle, a termination proof is often not necessary as it is proved behind the scenes automatically. In the case where Isabelle can not prove termination automatically, the function definition must be accompanied by a termination proof. Termination is proved in Isabelle by showing for all cases, that the size of the arguments to a recursive call is strictly less than the size of the input, with respect to some size measure. Isabelle provides a generic size function, overloaded for all types. This size function is however not used in the termination proof because it lacks two properties needed to prove termination. The size function should be invariant to the shifting of a formula. To see why, consider the function case implementing ME4 seen in Figure 14. In this excerpt, recursion is guaranteed to only be applied to an argument of smaller size if we can ensure that α has the same size as the shifted α, since the Exists constructor has been excluded from the recursion.

```
(*4*) | "rewrite (RAnd α (RExists β)) t =
       (if prop_cond α β
        then RExists (rewrite (RAnd (shiftI_r 0 α) β) Same)
```

Figure 14: Excerpt of rewrite function highlighting why a size function invariant to the shifting of a formula, is needed to prove termination

The reason the generic size function is not shift invariant, is because the size of a natural number is the number itself, making the size of a formula increase as variables are incremented.

The second property that is needed of the size function, is that it is *non-zero*, i.e. the smallest size a formula can have is 1. The reason why can be seen in Figure 15. Here it can be seen that recursion is applied to $\alpha \wedge \beta$ and $\alpha \wedge \gamma$, both of which should be smaller than $\alpha \wedge (\beta \vee \gamma)$. This is however only the case if the size of β and γ is guaranteed to be greater than zero.

```
(*1*)  "rewrite (RAnd α (ROr β γ)) t =
       (if prop_cond α β ∨ prop_cond α γ
       then ROr (rewrite (RAnd α β) Same) (rewrite (RAnd α γ) Same)
```

Figure 15: Excerpt of rewrite function highlighting why a size function that never returns a value lower than 1, is needed to prove termination

A size function called `my_size` has been implemented that satisfies these properties of being shift invariant and non-zero. To ensure non-zeroness, pattern matching against each constructor in rformula, returns 1 plus the size of any subformulas. To ensure shift invariance, the size function ignores all constructor arguments that are not formulas.

Using `my_size` as a measure on the formula argument of the rewrite function is however not enough to prove termination. Consider the cases where a formula has not matched any rule and its sub-formulas are reordered before attempting rewriting again. These cases can be seen in Figure 16.

```
| "rewrite (RSince l I r) Same = rewrite (RSince r I l) Swapped"
| "rewrite (RUntil l I r) Same = rewrite (RUntil r I l) Swapped"
| "rewrite (RAnd l r) Same = rewrite (RAnd r l) Swapped"
```

Figure 16: An excerpt of the rewrite function, reordering sub-formulas before reattempting a rewrite.

The final touch that is needed, is to define a measure on the flag of type argpos, assigning `Same` to 1 and `Swapped` to 0. Termination can then be showed by using as a size measure, the lexicographical ordering of the size of the formula and the size of the flag.

4.2 Substitution Contexts

In Section 2, the equivalences that the rewrite function is based on were proved, but because rewriting is done recurisvely throughout the sub-formulas, more than the equivalences alone is needed to prove correcntess of the rewrite function. This can be seen in Figure 17, which is the rewrite rule implementing ME4.

```
inductive f_con where
f_con_1_t: "f_con (λf1. Formula.Exists f1)"|
f_con_2_t: "f_con (λf1. Formula.Neg f1)" |
f_con_3_t: "f_con (λf1. Formula.Until TT I f1)"|
f_con_4_t: "f_con (λf1. Formula.Since TT I f1)" |
f_con_5_t: "f_con (λf1. Formula.Next I f1)"|
f_con_6_t: "f_con (λf1. Formula.Prev I f1)"
```

Figure 18: Restriction predicate on a substitution context

```
lemma sub_1:
"f_con P ⟹ (⋀v i. Formula.sat σ V v i (project α) =
                   Formula.sat σ V v i (project β)) ⟹
 Formula.sat σ V v i (P (project α)) = Formula.sat σ V v i (P (project β))"
```

Figure 19: Restricted substitution lemma

```
| "rewrite (RAnd α (RExists β)) t =
 (if prop_cond α β
  then RExists (rewrite (RAnd (shiftI_r 0 α) β) Same)
  else let α' = rewrite α Same; β' = rewrite β Same in RAnd α' (RExists β'))"
```

Figure 17: Applying ME4 as a rewrite rule

If the propagate condition is satisfied, it must be shown that the rewritten formula is equivalent to the right-hand-side of ME4. This could be achieved by proving a general substitution lemma that allows equivalent sub-formulas to be substituted within a larger formula. It could be stated as $(\forall \sigma'\ V'\ v'\ i'.\ sat\ \sigma'\ V'\ v'\ i'\ \alpha = sat\ \sigma'\ V'\ v'\ i'\ \beta) \implies sat\ \sigma\ V\ v\ i\ (P\ \alpha) = sat\ \sigma\ V\ v\ i\ (P\ \beta)$. This statement is however not true because P is an arbitrary function, in particular it could be a function that returned false for some formulas and true for others. The problem with the lemma is thus the unrestrictedness of P. Allowing P to be an arbitrary function is also a much stronger lemma than is needed. For the example given, all that's required is that the sub-formula of the Exists constructor can be substituted. P will be called a substitution-context, and it is constrained, by defining the predicate $f\_con :: formula \to formula$ (short for formula context) defining substitution-contexts by cases. Figure 18 shows all the cases that where necessary in the correctness proof of the rewrite function. The substitution lemma can be seen in Figure 19, which is proved by cases over f_con derivations. There is one derivation per rule.

Implications of embedding As the rewrite function uses the embedded syntax, an embedded variant of the substitution lemma sub_1 had to be proved (called $rsub\_1$). This is done by defining the predicate $f\_conr :: rformula \to rformula$, such that $f\_conr\ P$ constrains P to a set of rformula substitution contexts by cases. Substitution contexts on the level of rformulas are re-

```
lemma rsub_1:
"f_conr P ⟹ (⋀v i. rsat σ V v i α =
                    rsat σ V v i β) ⟹
              rsat σ V v i (P α) = rsat σ V v i (P β)"
```

Figure 20: Restricted substitution lemma on the level of rformulas.

lated to substitution contexts on the level of formulas by a translation relation $trans :: (rformula \to rformula) \to (formula \to formula)$. To prove $rsub\_1$, three intermediate lemmas must be shown about this $trans$ relation. To relate $trans$ to the predicates f_conr and f_con it must be shown that the first argument of a $trans$ relation indeed must satisfy the f_conr predicate and likewise the second argument must satisfy the f_con predicate. The next lemma that must be shown is that $trans\ conr\ con \implies project\ (conr\ r) = con\ (project\ r)$ i.e. after instantiation and projection both contexts are equivalent. Finally it must be shown that all rformula contexts $conr$ that satisfy $f\_conr\ conr$, have a translation to a formula context con defined in the $trans$ relation. These three lemmas along with sub_1 is enough to prove $rsub\_1$ whose definition can be seen in Figure 20.

Using the substitution contexts Returning to the rewrite rule that applies ME4, to use the substitution lemma to reduce the recursive expression, an equivalence is needed between `rewrite (RAnd (shiftI_r 0 α) β) Same` and `fix_r (RAnd (shiftI_r 0 α) β) Same`. Isabelle functions defined with the "function" keyword whose termination has been proved, provides an induction scheme called *computation induction*, where induction follows the computation. The exact equivalence that is needed is therefore available as an assumption when proving this case.

Other substitution contexts Two additional kinds of substitution contexts are used. For convenience, a substitution lemma for contexts expecting two input formulas, is also given. By necessity a separate substitution lemma is given for the aggregation context. A separate aggregation context is necessary, because the sub-formula that is aggregated over, must be more than just equivalent in terms of satisfiability to its replacement, they must also share the same free variables. Equivalence in terms of satisfiability does not imply that formulas share the same variables.
The proof for the aggregation substitution lemma is exactly the same as we have seen, but the aggregation substitution lemma not only assumes an equivalence between the substituted and replaced formula, it also assumes that their free variables coincide. The equivalence the substitution is based on is of the form $sat\ \sigma\ V\ v\ i\ (rewrite\ \alpha) = sat\ \sigma\ V\ v\ i\ \alpha$.[4] To then show that the free variables coincide, a separate lemma was proved which showed $fv\ (rewrite\ \alpha) = fv\ \alpha$.

[4]The argpos argument of the rewrite function is omitted for simplicity.

5 Conclusion

In this report, the value in formally verifying a procedure used in runtime monitoring tools such as MonPoly was shown. The core monitoring algorithm of MonPoly, called *VeriMon*, has been formally verified but other procedures used in MonPoly remain unverified such as formula rewriting. In this project the motivation behind the rewriting of formulas, namely to tackle the problem of domain independence, was described and a formally verified rewrite function was implemented in Isabelle. The unverified rewriting procedure currently implemented in MonPoly is based on several equivalences which in this project were shown to be unsound. A source of the unsoundness was due to an incorrect lower-bound of the interval I in $\Diamond_I$ and $\blacklozenge_I$. The solution to this problem was described as well. The other source of unsoundness was moving a sub-formula under an existential quantifier without ensuring capture-avoidance. The challenges in proving equivalences when the De Bruijn indices of variables must be shifted in a capture-avoiding way was described. It was also shown how strictness of temporal operators has been enforced without the presence of strict temporal operators in the syntax of formulas.

Implementation details about the rewrite function were seen and they included considerations on when to apply rewrite rules. The greatest implementation challenge was however representing the pattern-match cases in an Isabelle-friendly way. This meant that overlapping patterns were implemented sparingly to avoid a combinatorial explosion in the number of cases of Isabelle's internal representation of the rewrite function.

Finally relevant aspects of the correctness of the rewrite function itself were shown. This included how to prove termination by defining a size measure that satisfied the properties of decreasing for all recursive calls. It also included the use of substitution contexts to simplify expressions by removing recursive calls to the rewrite function.

As future work it would be interesting to prove some lemmas about range restriction. The purpose of the rewrite function is to propagate range restriction and since the functions most important property of preserving satisfiability has been shown, it would be interesting to show other properties related to range restriction.
The equivalences that have been proved in this project were originally defined in a section of Müller's thesis where he defines a larger procedure in which rewriting is just one step. Another step that happens earlier in this procedure involves pushing negations in formulas inwards and unverified rewrite rules for accomplishing this are given. Future work could also include the verification of these rewrite rules.
The auxiliary equivalences are called auxiliary because they were not used directly in the rewrite function, but they could have been added. This would also

be a meaningful addition as it would allow more formulas to be rewritten to safe formulas.
Finally in Section 3.1 it was mentioned that an improvement for the propagate condition of ME1 had been discussed with the advisors but not implemented. Implementing this improvement would be a useful as well as a simple addition because it has nearly no effect on the correctness proof of the rewrite function.

References

[1] D. A. Basin, F. Klaedtke, S. Müller, and E. Zalinescu, "Monitoring metric first-order temporal properties," *J. ACM*, vol. 62, no. 2, pp. 15:1–15:45, 2015.

[2] D. A. Basin, F. Klaedtke, and E. Zalinescu, "The monpoly monitoring tool," in *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, September 15, 2017, Seattle, WA, USA* (G. Reger and K. Havelund, eds.), vol. 3 of *Kalpa Publications in Computing*, pp. 19–28, EasyChair, 2017.

[3] D. A. Basin, T. Dardinier, L. Heimes, S. Krstic, M. Raszyk, J. Schneider, and D. Traytel, "A formally verified, optimized monitor for metric first-order dynamic logic," in *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I* (N. Peltier and V. Sofronie-Stokkermans, eds.), vol. 12166 of *Lecture Notes in Computer Science*, pp. 432–453, Springer, 2020.

[4] A. V. Gelder and R. W. Topor, "Safety and translation of relational calculus queries," *ACM Trans. Database Syst.*, vol. 16, no. 2, pp. 235–278, 1991.

[5] S. Müller, *Theory and applications of runtime monitoring metric first-order temporal logic*. PhD thesis, ETH Zurich, Zürich, 2009.

[6] A. Krauss, "Defining recursive functions in isabelle/hol," 2007.