ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Safe Evaluation of MFOTL Dual Temporal Operators

Bachelor's Thesis

Nico Hauser

August 09, 2021

Supervisor: Prof. Dr. David Basin
Advisor: Martin Raszyk

Department of Computer Science, ETH Zürich

Abstract

Runtime verification is the task of checking whether a given system satisfies its specification at runtime. This is usually accomplished using a runtime verification monitor: A program verifying the behaviour of a system. The monitor takes as input the actions taken by the target system and verifies whether they satisfy the specification at all times. The Institute of Information Security at ETHZ developed a formally verified monitor called *VeriMon*. VeriMon uses formulas in metric first-order temporal logic (MFOTL) as a way of formally specifying the behaviour of a target system and outputs all assignments to the free variables that satisfy a formula at given points in time. Since VeriMon uses finite tables to represent its output and reports all satisfying assignments, formulas with infinitely many satisfying assignments such as the negation of a formula with finitely many satisfying assignments cannot be monitored. It is possible to derive syntactic conditions (also called safety conditions) that guarantee that a given formula can be monitored by exclusively using finite tables in order to represent the intermediate results. If a given formula satisfies the safety conditions, the formula and the evaluation of said formula are called *safe*.

The MFOTL operator *Since* requires the satisfaction of a formula at all time points since another formula was satisfied. *Trigger* is the dual operator of Since and expresses its negation. The operators *Until* and *Release* are the symmetric counterparts to Since and Trigger reach into the future instead of the past. Because Trigger and Release are defined in terms of negations of formulas, the evaluation according to their definition often results in an infinite amount of satisfying assignments to the free variables.

This Bachelor's thesis presents MFOTL formulas semantically equivalent to the MFOTL operators *Trigger* and *Release* that reduce these operators to the more standard MFOTL operators Since and Until while relaxing the safety conditions compared to those implied by the definition of the dual operators. Moreover VeriMon is extended to support the evaluation of Trigger and Release under the relaxed safety conditions. The most straight-forward approach is to translate the operators to the semantically equivalent formulas within the monitoring algorithm. This effectively reduces the task of monitoring to the cases of Since and Until. An alternative approach pursued is the development of a specialized monitoring algorithm for Trigger. This specialized algorithm achieves a better performance in our empirical evaluation than the evaluation of the translated formulas. The semantic equivalence of the translated formulas and the correctness of the specialized monitoring algorithm are formally proven using the Isabelle/HOL proof assistant.

Acknowledgements

I would like to thank my advisor Martin Raszyk for his continuous assistance throughout the last six months. I am really grateful for his valuable inputs and aid in completing this thesis and thus would like to express my sincere gratitude.

Contents

Chapter 1

Introduction

Many security-critical systems run on complex software stacks for which it is becoming increasingly more difficult to cover every possible execution with tests. Often one cannot tolerate bugs in these security-critical systems and hence one would like to have stronger guarantees than tests give. An alternative to tests is to formally verify the program, i.e. to rigorously prove its correctness formulated as a mathematical statement. This can result in strong guarantees, but the process of formally verifying a program is very time-consuming. Yet another approach is to monitor the behaviour of the system during its execution at runtime. The latter approach is called runtime verification.

1.1 Formal Static Verification

In this section, approaches to formal verification of systems are presented. As already mentioned, formal verification is another way of getting guarantees for systems. In contrast to testing, formal verification is able to give much stronger guarantees such as the correctness for all possible inputs. Especially for security-critical applications, guarantees over all inputs are much more desirable as the program's properties are no longer dependent on the test coverage, i.e. how many possible executions of the system are covered by tests.

1.1.1 Theorem Provers

In mathematics, proofs consist of statements that logically follow from each other. Mostly one starts with some assumptions and then shows that these assumptions imply some conclusion. Usually, the steps between these statements are kept rather short so that readers can follow the thought process and hence verify whether the proof is correct.

Theorem provers are a way to automate this procedure: Instead of having humans check the proofs, the proofs are verified automatically. This is much quicker and less error-prone since a computer can be instrumented to base the verification only on a small set of axioms and lemmas derived from these axioms. Of course a human still has to make sure that the defined axioms are sound but this is a non-recurring task and thus scales much better than manually verifying every new line of a proof.

Isabelle [22] and Coq [3] are examples of general-purpose theorem provers based on the paradigm of functional programming. A comparison of the two is made by Yushkovskiy [26]. In addition to the ability to prove formalized mathematical statements, Isabelle and Coq both support code generation [1, 19]. This means that after formally defining data types and functions, one can directly export them to executable code with guaranteed partial correctness. Partial correctness in contrast to total correctness does not guarantee termination. If in addition properties about the exported functions were proven, the exported code is guaranteed to satisfy them. We refer to Nipkow and Klein that introduce Isabelle and HOL in a practical way [21].

1.1.2 Verifying Imperative Code

As mentioned in Section 1.1.1, theorem provers work very well for functional programming. For imperative programming, other approaches such as Hoare triples [16] exist.

Hoare triples have the form of $P\{C\}Q$ where P is called the *precondition*, Q the *postcondition* and C is the program these assertions are about. The meaning of such a Hoare triple is that if the program C is executed when the precondition P is satisfied, then either the program does not terminate or the postcondition Q is established afterwards.

Examples for pieces of software making use of these approaches are Viper [20] or Dafny [18] which are both based on Boogie [17]. A major difference to theorem provers is that the code is not generated, one instead directly proves properties about the hand-written code. Viper and Dafny which were first presented in 2015 [20] and 2010 [18], respectively, are relatively new compared to Isabelle which was first presented in 1986 [22] and Coq which was first released in 1989 [4] but the concepts Viper and Dafny make use of have been around for much longer as well: Hoare triples, for example, were introduced in 1969 [16].

1.2 Runtime Verification

Runtime verification is an alternative approach to verifying the behaviour of a system: Instead of verifying properties about a system ahead of it being

run, runtime verification tackles this problem during or after the system's execution. As in formal verification (Section 1.1), one must first describe the desired behaviour in a formal specification. An additional program called the monitor verifies whether the target system behaves as described by the specification. Runtime verification can operate in two different settings: online and offline monitoring.

Online monitoring In online monitoring the case where the target system is run side-by-side a monitor is considered. All information about the behaviour of the target system relevant for the specification is passed to the monitor. The monitor then verifies whether the system satisfies the specification at all times and reports potential violations while the system is executing.

Offline monitoring In contrast to online monitoring, offline monitoring is performed asynchronously. The information relevant for the specification is for example written to a log file. Later this log file can be used as the input to the monitor which will then check whether the program behaved as described in the specification and report violations if there were any.

There is a big difference between formal verification and runtime verification of software: In the former the program is effectively prevented from having a behaviour different from its specification while in the latter instances where the program does not behave according to the specification are just detected. Still, runtime verification has one major advantage over formal verification and that is productivity: Most software written today is not formally verified as writing formally verified software is much more time consuming. It requires the programmer to rigorously prove properties about the program she or he writes. Rewriting large existing programs to make use of formal verification seems impractical for many applications whereas runtime verification can be added incrementally with minor changes to existing codebases. Depending on the use case this might strike a good balance between development time and effectiveness: If bugs need not be prevented at all cost but their detection suffices, runtime verification might be a good choice. Violations of the specification are reported which enables the developers to take action and fix the detected flaws. For a more in-depth introduction to runtime verification we refer to the book by Bartocci et al. [10].

1.2.1 Traces

A question that remains is how the information about the programs behaviour is represented. A standard approach is to have a list of sets of *events* paired with a *timestamp*. An *event* contains information about the program's state at the respective point in time. As time just passes in one direction, the

timestamps are monotonically increasing. The indices in this event list are called *time points* [12] whereas the whole list is called a *trace*. One can also describe a trace as a *log* containing the relevant information about the program's behaviour.

Events that appear to occur at the same time can be grouped together in a *database* [11]. One way to represent the information about the programs behaviour is to use first-order predicates which means databases are sets of first-order predicates. Note that even though simultaneous events can be grouped in databases, multiple trace entries are still allowed to have the same timestamp.

As an example we present a login system using the two predicates `LoggedIn(u)` and `LogsIn(u)` where the former is satisfied at a given time point if and only if the user u is logged in and the latter if and only if the user performs the login at the respective time point. An example trace is depicted in Table 1.1.

time point	timestamp	database
0	5	{`LogsIn("Alice")`}
1	6	{`LoggedIn("Alice")`}
2	6	{`LoggedIn("Alice")`}
3	7	{`LoggedIn("Alice")`}
4	7	{`LoggedIn("Alice"), LogsIn("Bob")`}
5	8	{`LoggedIn("Alice")`}

Table 1.1: An example trace

1.2.2 Specifications

In addition to having a way to represent the behaviour of the program, a way to formulate the specification is required. The choice should not be natural language as words usually have multiple meanings and would thus result in ambiguous specifications. Mathematical notation is as precise as it gets and in fact different types of logic have been used for runtime verification specifications: Linear temporal logic (LTL) [23], metric temporal logic (MTL) [25], metric first-order temporal logic (MFOTL) [13], metric first-order dynamic logic (MFODL) [11] or metric interval temporal logic (MITL) [14]. Alur and Henzinger [9] compare the expressiveness, syntax and semantics of some of these LTL variants. For a general introduction to logic we refer to the book by Enderton [15].

The various logics differ in whether they allow predicates with data and quantifiers (first-order logic) and how time is modelled. In linear temporal logic there are operators imposing conditions on the past and future but

their range is not restricted. In metric temporal logic the range of temporal operators can be restricted using intervals over discrete timestamps whereas metric interval temporal logic allows intervals over continuous timestamps.

Going back to the example from section 1.2.1, it is now possible to give a specification like

$$\texttt{LoggedIn(u)}\ S_{[0,\infty)}\ \texttt{LogsIn(u)} \tag{1.1}$$

This specification can be read as follows: Since a given user u logged in (`LogsIn(u)`), he is logged in (`LoggedIn(u)`). The operator S is called *Since* and the subscript $[0, \infty)$ means its range is not restricted. The syntax and semantics are formally introduced in Section 2.2. Informally a Since operator requires the left-hand side to hold since the next time point after the right-hand side held in the past up until the current time point.

Given this informal semantics and the example trace in Table 1.1, one can deduce that formula 1.1 is satisfied at all time points for the assignment `u = "Alice"` to the free variable. At time point 4, the formula is also satisfied for the assignment `u = "Bob"` but at time point 5, formula 1.1 is violated for `u = "Bob"` as `LoggedIn("Bob")` did not hold in time point 5 after `LogsIn("Bob")` was part of the database at time point 4.

1.3 Monitorable MFOTL Formulas

D. Basin et al. [12] describe two approaches for monitoring MFOTL formulas. One approach builds on finite automata and the other on finite tables. Since the latter approach makes use of finite tables, it is often possible to use the operations from relational algebra to manipulate the finite tables efficiently. Unfortunately not all MFOTL formulas have equivalent descriptions in relational algebra and hence the class of MFOTL formulas has to be restricted if one wants to represent the results using finite tables containing the set of all satisfying assignments to the free variables. It is possible to syntactically restrict formulas to a subset where this property is guaranteed. We call this subset *safe* formulas and a given formula *safe* or *unsafe* depending on whether it is part of said subset. The evaluation of safe formulas using relational algebra (finite tables) is closely related to the concept of RANF described by Abiteboul et al. [8, Section 5.4].

A simple example for an unsafe formula is the negation of an equality: $\neg(x = 3)$ for the free variable $x \in \mathbb{N}$. For any $x \in \mathbb{N}\backslash\{3\}$, the formula is satisfied and hence the set of all satisfying assignments to the free variables cannot directly be represented by a finite table.

The *safe evaluation* of a formula denotes a procedure to obtain its satisfying assignments by only using finite tables to represent the intermediate results of certain subformulas. Moreover it is in general not possible to monitor a formula that requires some conditions to hold for an infinite amount of time because the monitoring algorithm does not have access to an infinite trace of an observed system's execution. Formulas that only impose conditions on a finite number of future time points are called *future bounded*. If a formula is *safe* and *future bounded* it is called *monitorable*.

VeriMon is a formally verified runtime monitor developed by the Institute of Information Security at ETHZ that operates on the subset of monitorable MFOTL formulas just described. [24].

1.4 Trigger and Release

The semantics of the MFOTL operator Until are analogous to the operator Since that was already introduced informally in 1.2.2: Until requires the left-hand side to be satisfied from the current time point up until the last time point before the right hand side is satisfied. The symbol denoting Until is U.

Trigger (T) and Release (R) are the dual operators of Since and Until, respectively. Informally this means they express their negation. As an example for Trigger, one can think of an application with the same predicates `LogsIn`, `LoggedIn` as defined in Section 1.2.1 and the additional predicate `LogsOut` defined symmetrically to `LogsIn`. The specification might include something like "After logging in, the user will remain logged in". This can then be expressed as `LogsIn(u)` $T_{[0,*)}$ `LoggedIn(u)`, i.e. the action `LogsIn` "triggers" the state `LoggedIn`.

For Release, the same predicates can be used but now the specification is "The user remains logged in as long as she does not log out". This can be expressed by formula `LogsOut(u)` $R_{[0,*)}$ `LoggedIn(u)`, i.e. the action `LogsOut` "releases" the state `LoggedIn`.

1.5 Contributions

This thesis is concerned with the safe evaluation of the MFOTL dual operators Trigger and Release. By their definition they are inherently unsafe and hence it is not straightforward how to monitor them. First a translation of the dual operators into semantically equivalent formulas only using the standard operators Since and Until is presented. This would already allow their evaluation in VeriMon under less strict safety conditions. However the evaluation of these translated formulas is not as performant as it could be and hence a specialized, more performant algorithm for the evaluation of

Trigger is developed along with a proof of its semantic correctness in Isabelle / HOL. The performance of the specialized algorithm is then compared to the performance of the translated formulas in various settings. Last but not least, the asymptotic behaviour of the running time of the translated formulas and the specialized algorithm are assessed as well.

Chapter 2

Preliminaries

2.1 Since and Until

Since and *Until* are two binary MFOTL operators that allow the specification of a temporal relationship between multiple time points in a trace. Their semantics are defined formally in definition (2.3), here just an intuition for their meaning is presented. The operator Since requires a given condition φ to hold since another condition ψ held.

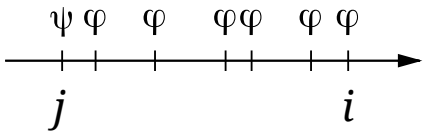

Figure 2.1: Example of a since

Slightly more formally, a Since is satisfied at a time point i in the trace, if there exists a $j < i$ in the past where ψ was satisfied and if at all time points strictly greater than j up to (and including) i, the formula φ is satisfied. This scenario is depicted in Figure 2.1.

Analogously, Until requires a condition φ to hold until another condition ψ is satisfied. Figure 2.2 shows this scenario.

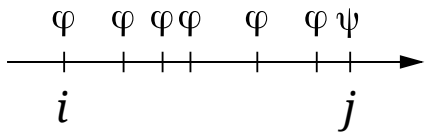

Figure 2.2: Example of an until

The difference is the temporal direction they impose their restrictions on: Since specifies conditions on the past whereas Until on the future. As mentioned before, MFOTL allows the restriction of the range of these temporal operators. Informally this just means that the timestamp associated with the time point j since when or up until when φ has to be satisfied must lie within some range of timestamps. This range is specified using an interval $I = [a, b)$ where a denotes the minimal difference in time and b is an upper bound for said time difference.

2.2 Metric First-Order Temporal Logic

This section formally introduces the syntax and semantics of MFOTL. The notation, syntax and semantics used in this thesis are the same as the ones described by Basin et al. [12, Chapter 2]. Hence the remainder of this section very closely follows chapter 2 of the paper "Monitoring Metric First-order Temporal Properties" [12]. The concepts defined here use the same names, symbols and notation as introduced by Basin et al. [12].

Let $\mathbb{I}$ be the set of nonempty intervals over the natural numbers, $\mathbb{N} = \{0, 1, ...\}$. Any element in $\mathbb{I}$ can be expressed as a pair $[a, b)$ where $a \in \mathbb{N}$, $b \in \mathbb{N} \cup \{\infty\}$ and $a < b$.

Next, *atomic predicates* are defined. An atomic predicate is characterized by a unique symbol and an arity (number of arguments). Hence the set of all atomic predicates can be described with a pair (R, ι) where R is the set of atomic predicate symbols and ι a function mapping from the set of atomic predicate symbols R to the natural numbers, corresponding to the arity of the respective atomic predicate. Moreover let C be a finite set of constants disjoint from R. The tuple (C, R, ι) is then called a *signature* [12]. Finally let V be a global countably infinite set of variables where V is always disjoint from C and R.

Definition 2.1 MFOTL formulas over the signature $S = (C, R, \iota)$ are recursively defined as follows:

1. For any two symbols from the set of constants and variables $t_1, t_2 \in C \cup V$, $t_1 \approx t_2$ is a formula.

2. For any predicate symbol $r \in R$ and arguments $t_1, ..., t_{\iota(r)} \in V \cup C$, the predicate $r(t_1, ..., t_{\iota(r)})$ is a formula.
3. For any formula φ, $(\neg\varphi)$ is a formula.
4. For any formulas φ and ψ, $(\varphi \vee \psi)$ is a formula.
5. For any variable $v \in V$ and formula φ, $(\exists x.\varphi)$ is a formula.
6. For any interval $I \in \mathbb{I}$ and formula φ, $(\bullet_I \varphi)$ and $(\bigcirc_I \varphi)$ are formulas.
7. For any interval $I \in \mathbb{I}$ and formulas φ and ψ, $(\varphi\ S_I\ \psi)$ and $(\varphi\ U_I\ \psi)$ are formulas.

The operator $\bullet$ is called "Previous" whereas $\bigcirc$ is called "Next".

Moreover, the following standard abbreviations are used:

$$\begin{aligned} \varphi \wedge \psi &:= \neg\,((\neg\varphi) \vee (\neg\psi)) \\ \varphi \rightarrow \psi &:= (\neg\varphi) \vee \psi \\ \forall x.\, \varphi &:= \neg\,(\exists x.\, (\neg\varphi)) \end{aligned}$$

In addition, the temporal operators $\blacklozenge$ "Once" and $\Diamond$ "Eventually" are defined as follows:

$$\begin{aligned} \blacklozenge_I \varphi &:= \mathit{True}\ S_I\ \varphi \\ \Diamond_I \varphi &:= \mathit{True}\ U_I\ \varphi \end{aligned}$$

where $I \in \mathbb{I}$ and *True* corresponds to some valid formula without any free variables such as $\exists x.\, x \approx x$ [12]. In the remainder of this thesis, the abbreviation *True* will be used with the same semantics, hence we additionally define *True* to be syntactic sugar for $\exists x.\, x \approx x$ as well as *False* for $\exists x.\, \neg\,(x \approx x)$.

Next we define the operator Trigger and Release formally.

Definition 2.2 The operators Trigger and Release are defined as the dual operators of S and U, respectively:

$$\begin{aligned} \varphi\ T_I\ \psi &:= \neg\,((\neg\,\varphi)\ S_I\ (\neg\,\psi)) \\ \varphi\ R_I\ \psi &:= \neg\,((\neg\,\varphi)\ U_I\ (\neg\,\psi)) \end{aligned}$$

The temporal operators $\blacksquare$ "Historically" and $\Box$ "Always" are defined as special cases of Trigger and Release:

$$\begin{aligned} \blacksquare_I \varphi &:= \mathit{False}\ T_I\ \varphi \\ \Box_I \varphi &:= \mathit{False}\ R_I\ \varphi \end{aligned}$$

In the following, the set of free variables of a formula φ will be denoted by $free(\varphi)$. For the sake of notational simplicity, the order of the free variables in φ is fixed by having a vector of free variables $\bar{x} = (x_1, ..., x_n)$ with $free(\varphi) = \{x_1, ..., x_n\}$ associated with the formula. Moreover if for some formula φ, the set $free(\varphi)$ is empty, the formula is called *closed*.

In order to omit certain parentheses, standard operator binding strengths are used [12]. The operator $\neg$ binds stronger than $\wedge$ which in turn binds stronger than $\vee$. The operator $\vee$ binds stronger than the existential quantifier $\exists$ which still binds stronger than all temporal operators.

After defining the syntax of MFOTL formulas, their semantics can be defined. First the notion of a *structure* $\mathcal{D}$ over a signature $S = (C, R, \iota)$ is introduced [12]. Such a structure consists of a domain $|\mathcal{D}| \neq \emptyset$ and interpretations $c^{\mathcal{D}} \in |\mathcal{D}|$ and $r^{\mathcal{D}} \subseteq |\mathcal{D}|^{\iota(r)}$ for every $c \in C$ and $r \in R$. Note that the bars around $\mathcal{D}$ do *not* denote the cardinality of $\mathcal{D}$ ($\mathcal{D}$ is not even a set) but rather the domain of the structure $\mathcal{D}$.

On top of a structure, a *temporal structure* [12] can be built. That is a tuple $(\bar{\mathcal{D}}, \bar{\tau})$ where $\bar{\mathcal{D}}$ stands for a sequence of structures $\mathcal{D}_0, \mathcal{D}_1, ...$ and $\bar{\tau}$ for a sequence of natural numbers $\tau_0, \tau_1, ...$. A temporal structure must satisfy the following conditions:

1. The sequence $\bar{\tau}$ of timestamps is monotonically increasing and for every $t \in \mathbb{N}$, there exists a time point $i \in \mathbb{N}$ so that $\tau_i > t$, i.e. the time-stamps eventually increase.
2. The domain does not change, i.e. $|\mathcal{D}_i| = |\mathcal{D}_j|$ holds for all time points $i, j \in \mathbb{N}$.
3. The interpretation of constants does not change, i.e. $c^{\mathcal{D}_i} = c^{\mathcal{D}_j}$ holds for all time points $i, j \in \mathbb{N}$.

Since the domain and the interpretation of the constants does not change, they are denoted with $|\bar{\mathcal{D}}|$ and $c^{\bar{\mathcal{D}}}$, respectively.

Last but not least, some variable assignments for the variables V are required in order to define the semantics. Let v be a function mapping from the set of variables V to the domain $|\bar{\mathcal{D}}|$. Such a function is called a *valuation* [12] and maps every variable to a value from the domain.

For any $x, y \in V$ and $d \in |\bar{\mathcal{D}}|$, let

$$v[y \mapsto d](x) = \begin{cases} d & \text{if } x = y \\ v(x) & \text{otherwise} \end{cases}$$

be the function obtained by updating where the value y is mapped to in the function v. Analogously, for any $n \in \mathbb{N}$, pairwise distinct variable vector $\bar{y} = (y_1, ..., y_n)$ and $\bar{d} = (d_1, ..., d_n) \in |\bar{\mathcal{D}}|^n$, let

$$v[\bar{y} \mapsto \bar{d}](x) = \begin{cases} d_i & \text{if } x = y_i \text{ for some } i \in \{1..n\} \\ v(x) & \text{otherwise} \end{cases}$$

be the function, where all variables in the variable vector $\bar{y}$ are mapped to the values according to $\bar{d}$ and the other mappings remain unchanged.

As done in [12], we will also abuse notation and allow the application of the valuation v to constants $c \in C$ where v just maps them to their interpretation, i.e. $v(c) = c^{\bar{\mathcal{D}}}$.

Now everything is introduced so that the semantics of MFOTL formulas can be defined:

Definition 2.3 Let $(\bar{\mathcal{D}}, \bar{\tau})$ be a temporal structure over the signature $S = (C, R, \iota)$ with $\bar{\mathcal{D}} = \mathcal{D}_0, \mathcal{D}_1, ...$ and $\bar{\tau} = \tau_0, \tau_1, ...$ Moreover let α be a formula over the same signature S and let v be a valuation.

Then the formula α is satisfied at some index $i \in \mathbb{N}$ in the temporal structure if and only if $(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \alpha$. The relation $(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \alpha$ is recursively defined as follows:

$$\begin{aligned}
(\bar{\mathcal{D}}, \bar{\tau}, v, i) &\models t \approx t' && \Leftrightarrow \quad v(t) = v(t') \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) &\models r(t_1, \ldots, t_{\iota(r)}) && \Leftrightarrow \quad \big(v(t_1), \ldots, v(t_{\iota(r)})\big) \in r^{\mathcal{D}_i} \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) &\models (\neg \varphi) && \Leftrightarrow \quad (\bar{\mathcal{D}}, \bar{\tau}, v, i) \not\models \varphi \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) &\models (\varphi \vee \psi) && \Leftrightarrow \quad (\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \varphi \text{ or } (\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \psi \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) &\models (\exists x.\, \varphi) && \Leftrightarrow \quad (\bar{\mathcal{D}}, \bar{\tau}, v[x \mapsto d], i) \models \varphi, \text{ for some } d \in |\bar{\mathcal{D}}| \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) &\models (\bullet_I \varphi) && \Leftrightarrow \quad i > 0 \wedge\ \tau_i - \tau_{i-1} \in I \wedge (\bar{\mathcal{D}}, \bar{\tau}, v, i-1) \models \varphi \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) &\models (\bigcirc_I \varphi) && \Leftrightarrow \quad \tau_{i+1} - \tau_i \in I \text{ and } (\bar{\mathcal{D}}, \bar{\tau}, v, i+1) \models \varphi \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) &\models (\varphi\, S_I\, \psi) && \Leftrightarrow \quad \exists j \leq i.\ (\tau_i - \tau_j) \in I \wedge\ (\bar{\mathcal{D}}, \bar{\tau}, v, j) \models \psi \wedge \\
& && \quad\ (\bar{\mathcal{D}}, \bar{\tau}, v, k) \models \varphi \text{ for all } k \in \mathbb{N} \text{ with } j < k \leq i \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) &\models (\varphi\, U_I\, \psi) && \Leftrightarrow \quad \exists j \geq i.\ (\tau_j - \tau_i) \in I \wedge\ (\bar{\mathcal{D}}, \bar{\tau}, v, j) \models \psi \wedge \\
& && \quad\ (\bar{\mathcal{D}}, \bar{\tau}, v, k) \models \varphi \text{ for all } k \in \mathbb{N} \text{ with } i \leq k < j
\end{aligned}$$

2.3 Tables

In a given temporal structure $(\bar{\mathcal{D}}, \bar{\tau})$, for any formula φ with free variable vector $\bar{x} = (x_1, ..., x_n)$ and for any time point $i \in \mathbb{N}$ there exists a possibly empty or infinite set of satisfying *tuples* defined by

$$\varphi^{(\bar{\mathcal{D}},\bar{\tau},i)} := \left\{ \bar{d} \in |\bar{\mathcal{D}}|^n \;\middle|\; (\bar{\mathcal{D}}, \bar{\tau}, v[\bar{x} \mapsto \bar{d}], i) \models \varphi, \text{ for some valuation } v \right\} \text{[12]}$$

This definition is motivated by the fact that the satisfaction of a formula φ under a variable assignment only depends on the mappings of the free variables $\bar{x}$. Hence, one can represent it by a tuple over $\bar{x}$.

These satisfying tuples can be represented in a *table* where the columns correspond to the free variables $\bar{x}$ and the rows to the different assignments to the free variables. Hence a single element of the set $\varphi^{(\bar{\mathcal{D}},\bar{\tau},i)}$ corresponds to a single row. Example tables for the formulas $\varphi = r(x, y, z)$ and $\varphi = r(x, y, x)$ are shown in Table 2.1 and Table 2.2, respectively. The atomic predicate r corresponds to the following relation: $r = \{(1,1,1), (1,3,4), (4,3,4), (4,3,5), (5,1,5)\}$.

x	y	z
1	1	1
1	3	4
4	3	4
4	3	5
5	1	5

Table 2.1: An example table for the formula $\varphi = r(x, y, z)$

x	y
1	1
4	3
5	1

Table 2.2: An example table for the formula $\varphi = r(x, y, x)$

2.4 Monitorability

Definition 2.4 Let `safe_formula` be a unary predicate on formulas. For any formula φ, `safe_formula`(φ) holds if and only if φ satisfies the syntactic conditions defined in Figure 2.3. A formula φ satisfying these conditions, i.e. if `safe_formula`(φ) holds, is called *safe*.

The predicate `safe_formula` describes syntactic restrictions guaranteeing that a given formula can be monitored by only using finite tables to represent the intermediate results. Moreover if a given formula satisfies the predicate

`safe_formula` then the set of satisfying assignments to the free variables of said formula can also be represented using a finite table.

Lemma 2.5 For all formulas φ, the following statement holds.

$$\varphi \text{ is } \mathit{safe} \rightarrow \forall\, \bar{\mathcal{D}}\ \bar{\tau}\ i.\ \varphi^{(\bar{\mathcal{D}},\bar{\tau},i)} \text{ is finite}$$

The proof for this lemma follows by induction over the recursive definition of `safe_formula`.

The safety conditions relevant for this thesis are listed in Figure 2.3. These were already part of VeriMon before this thesis was started.

$$
\begin{aligned}
&\texttt{safe\_formula}\,(\neg\varphi) && \Leftrightarrow && \mathit{free}(\varphi) = \emptyset \wedge \texttt{safe\_formula}(\varphi)\\
&\texttt{safe\_formula}\,(\varphi \vee \psi) && \Leftrightarrow && \mathit{free}(\varphi) = \mathit{free}(\psi) \wedge\\
& && && \texttt{safe\_formula}\,(\psi) \wedge \texttt{safe\_formula}\,(\varphi)\\
&\texttt{safe\_formula}\,(\varphi \wedge \psi) && \Leftrightarrow && \texttt{safe\_formula}(\varphi) \wedge (\\
& && && \quad \texttt{safe\_assignment}\,(\mathit{free}(\varphi), \psi)\ \vee\\
& && && \quad \texttt{safe\_formula}\,(\psi)\ \vee\\
& && && \quad \mathit{free}(\psi) \subseteq \mathit{free}(\varphi) \wedge (\\
& && && \quad\quad \texttt{is\_constraint}(\psi) \vee\\
& && && \quad\quad \exists\psi'.\psi = \neg\psi' \wedge \texttt{safe\_formula}(\psi')\\
& && && \quad)\\
& && &&)\\
&\texttt{safe\_formula}\,(\varphi\ S_I\ \psi) && \Leftrightarrow && \texttt{safe\_formula}(\psi) \wedge \mathit{free}(\varphi) \subseteq \mathit{free}(\psi)\ \wedge (\\
& && && \quad \texttt{safe\_formula}(\varphi) \vee\\
& && && \quad \exists\varphi'.\varphi = \neg\varphi' \wedge \texttt{safe\_formula}(\varphi')\\
& && &&)\\
&\texttt{safe\_formula}\,(\varphi\ U_I\ \psi) && \Leftrightarrow && \texttt{safe\_formula}(\psi) \wedge \mathit{free}(\varphi) \subseteq \mathit{free}(\psi)\ \wedge (\\
& && && \quad \texttt{safe\_formula}(\varphi) \vee\\
& && && \quad \exists\varphi'.\varphi = \neg\varphi' \wedge \texttt{safe\_formula}(\varphi')\\
& && &&)
\end{aligned}
$$

Figure 2.3: The equations of `safe_formula` relevant for this thesis. Definitions with adapted notation taken from VeriMon [6]

The details of `safe_assignment` and `is_constraint` are omitted as they are not relevant for this thesis. In a nutshell, these predicate hold on non-recursive formulas of the form of an equality or the negation of an equality.

The concept of future boundedness was already introduced in Section 1.3 as the property of a formula only imposing conditions on a finite amount of time points in the future. The only operator that reaches into the future for more than just one time point is the Until operator. Thus the definition can be made more formal:

Definition 2.6 A formula φ is called *bounded*, if all occurrences of Until operators in φ are over bounded intervals [12].

Both, boundedness and safety are requirements on a formula so that it can be monitored. Hence we define monitorability as follows:

Definition 2.7 A formula φ is called *monitorable* if and only if it is *safe* and *bounded*.

2.5 Progress

Since MFOTL includes temporal operators that can impose conditions on the future, it is not always possible to compute $\varphi^{(\bar{\mathcal{D}},\bar{\tau},i)}$ directly after having seen a finite prefix of the trace up until the time-point i. As an example, the table $\varphi^{(\bar{\mathcal{D}},\bar{\tau},i)}$ for the formula $\varphi = \bigcirc_{[0,*)} \psi$ for some atomic predicate ψ can only be outputted after reading $(\mathcal{D}_{i+1}, \tau_{i+1})$ which might not be available yet. In other words, for a given formula and time point, the monitor can only make a certain amount of *progress*.

Definition 2.8 The *progress* $prog^{\mathcal{A}}(\varphi, i)$ of a monitoring algorithm $\mathcal{A}$ on a formula φ at a time point i is defined as the number of time points j for which the value $\varphi^{(\bar{\mathcal{D}},\bar{\tau},j)}$ can be computed after reading the first i time-points.

2.6 Formalization in Isabelle / HOL

In Isabelle / HOL, the introduced concepts are formalized using various functions and data types. As they closely follow the just given definitions only the relevant differences are mentioned here. Even though the intervals introduced in section 2.2 could be modelled by a simple pair $[a, b)$ where $a \in \mathbb{N}, b \in \mathbb{N} \cup \{\infty\}$, the intervals are modelled more generally in VeriMon to allow for potential future extensions of intervals beyond a pair of natural numbers. In the formalization the intervals are modelled as tuples `(memL, memR, bounded)` where the functions

$$\begin{aligned}\texttt{memL} &: \mathbb{N} \mapsto \{\texttt{True}, \texttt{False}\}\\ \texttt{memR} &: \mathbb{N} \mapsto \{\texttt{True}, \texttt{False}\}\\ \texttt{bounded} &\in \{\texttt{True}, \texttt{False}\}\end{aligned}$$

satisfy the following equations:

$$\exists m \in \mathbb{N}.\, \texttt{memL}(m) \wedge \texttt{memR}(m) \tag{2.1}$$
$$\forall\, l, m \in \mathbb{N}.\, \texttt{memL}(l) \rightarrow l \leq m \rightarrow \texttt{memL}(m) \tag{2.2}$$
$$\forall\, m, r \in \mathbb{N}.\, \texttt{memR}(r) \rightarrow m \leq r \rightarrow \texttt{memR}(m) \tag{2.3}$$
$$\texttt{bounded} \leftrightarrow \exists m. \neg \texttt{memR}(m) \tag{2.4}$$

Informally, `memL` checks whether the input is at least as large as the lower bound whereas `memR` checks whether the input is at most of the size of the upper bound. Equations 2.2 and 2.3 characterize the transitivity of these relations. Equation 2.1 guarantees that intervals are non-empty whereas Equation 2.4 defines the last component of the tuple to be `True` if and only if the interval is bounded.

An interval $[a, b+1)$ can then be represented by the tuple

$$\begin{cases} (\lambda i.\, i \geq a, \lambda i.\, i \leq b, \texttt{False}) & \text{if } b = \infty \\ (\lambda i.\, i \geq a, \lambda i.\, i \leq b, \texttt{True}) & \text{if } b \neq \infty \end{cases}$$

where the notation $\lambda i.\, f(i)$ for a function $f : X \mapsto Y$, some X, Y and an argument $i \in X$ denotes the function taking an argument i from X and returning $f(i) \in Y$.

Chapter 3

Translating Dual Operators

As already mentioned in the Introduction in Section 1.5, both Trigger and Release are inherently unsafe. With the semantics of MFOTL formulas being defined in section 2.2 and having seen the relevant equations for `safe_formula` in Figure 2.3, the reason immediately becomes apparent: Trigger and Release have a top-level negation and negated formulas are in general only safe if they do not contain any free variables and are safe by themselves (see Figure 2.3). However, this restriction is stronger than it could be.

An approach to deal with the inherent unsafety was described by Basin et al. ([12], Section 4). The idea is to require that zero is part of the interval, in addition to further syntactic conditions resembling those of Since and Until. This is already better but if possible it would be beneficial to allow intervals not including zero as well.

The conditions described in this thesis are derived from the approach suggested by M. Raszyk: Expressing the two operators with the already supported operators Since and Until. The safety conditions can then be derived by evaluating `safe_formula` for the rewritten formula.

Now the structure of this chapter is outlined. First the semantics of Trigger and Release are derived on the basis of their definition and the semantics of MFOTL. Next, the translated formulas along with their intuition are presented. On the basis of these rewrite rules, the safety conditions for Trigger and Release are then derived.

3.1 Semantics

3.1.1 Trigger

We start off by looking at Trigger. By definition 2.2, the following holds

$$\varphi \ T_I \ \psi = \neg \left((\neg \varphi) \ S_I \ (\neg \psi) \right)$$

whereas by the semantics 2.3 it holds that

$$\begin{aligned}
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models (\neg \varphi) \quad &\Leftrightarrow \quad (\bar{\mathcal{D}}, \bar{\tau}, v, i) \not\models \varphi \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models (\varphi \, S_I \, \psi) \quad &\Leftrightarrow \quad \exists j \leq i.\ (\tau_i - \tau_j) \in I \wedge\ (\bar{\mathcal{D}}, \bar{\tau}, v, j) \models \psi \wedge \\
&\qquad (\bar{\mathcal{D}}, \bar{\tau}, v, k) \models \varphi, \text{ for all } k \in \mathbb{N} \text{ with } j < k \leq i
\end{aligned}$$

By plugging the definition into the semantics, one obtains

$$\begin{aligned}
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models (\varphi \, T_I \, \psi) \quad &\Leftrightarrow \quad (\bar{\mathcal{D}}, \bar{\tau}, v, i) \not\models ((\neg\, \varphi)\ S_I\ (\neg\, \psi)) \\
&\Leftrightarrow \quad \neg(\exists j \leq i.\ (\tau_i - \tau_j) \in I \wedge \\
&\qquad (\bar{\mathcal{D}}, \bar{\tau}, v, j) \models \psi \wedge \\
&\qquad (\bar{\mathcal{D}}, \bar{\tau}, v, k) \models \varphi \text{ for all } k \in \mathbb{N} \\
&\qquad \text{with } j < k \leq i) \qquad (3.1) \\
&\Leftrightarrow \quad \forall j \leq i.\ (\tau_i - \tau_j) \in I \rightarrow \\
&\qquad (\bar{\mathcal{D}}, \bar{\tau}, v, j) \models \psi \vee \\
&\qquad (\bar{\mathcal{D}}, \bar{\tau}, v, k) \models \varphi \text{ for some } k \in \mathbb{N} \\
&\qquad \text{with } j < k \leq i
\end{aligned}$$

The semantics can be described as follows: For all time points j before the current one (i) where the difference of timestamps is within the interval I, the following condition has to hold: Either ψ holds at j or there exists a strictly greater time point $k > j$, less than or equal to the current one ($k \leq i$), where φ holds. The intuition for the name Trigger comes from the following: Either ψ holds in the whole interval or there exists a time point within the interval where ψ does not hold. In the latter case for the Trigger operator to still be satisfied, there has has to exist a later time point where φ holds. After (and including) the last time point (in increasing time point order) where φ was satisfied, ψ must hold at all time points within the interval for the Trigger semantics to be satisfied. Hence one can think of this last time point where φ holds as triggering the afterwards continuous satisfaction of ψ within the interval.

3.1.2 Release

Now an analogous derivation is performed for Release using its definition (2.2) and the semantics of MFOTL (2.3):

$$
\begin{aligned}
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models (\varphi \, R_I \, \psi) \quad &\Leftrightarrow \quad (\bar{\mathcal{D}}, \bar{\tau}, v, i) \not\models ((\neg \, \varphi) \; U_I \; (\neg \, \psi)) \\
&\Leftrightarrow \quad \neg(\exists j \geq i. \; (\tau_j - \tau_i) \in I \wedge \\
&\qquad (\bar{\mathcal{D}}, \bar{\tau}, v, j) \models \psi \wedge \\
&\qquad (\bar{\mathcal{D}}, \bar{\tau}, v, k) \models \varphi \text{ for all } k \in \mathbb{N} \\
&\qquad \text{with } i \leq k < j) \qquad (3.2) \\
&\Leftrightarrow \quad \forall j \geq i. \; (\tau_j - \tau_i) \in I \rightarrow \\
&\qquad (\bar{\mathcal{D}}, \bar{\tau}, v, j) \models \psi \vee \\
&\qquad (\bar{\mathcal{D}}, \bar{\tau}, v, k) \models \varphi \text{ for some } k \in \mathbb{N} \\
&\qquad \text{with } i \leq k < j
\end{aligned}
$$

The semantics can be interpreted analogously to the one for Trigger: All time points within the interval must satisfy ψ up until and including the first occurrence of a time point where φ is satisfied and throughout the entire interval if there is none.

3.2 Rewriting

According to the semantics of Trigger and Release, they are both satisfied if there exists no time point within the interval. Hence in that case there is no hope to obtain a safe formula that is semantically equivalent (unless the formula is closed, i.e., without any free variables) and we restrict ourselves to the case where there is a time point within the interval. This means finding formulas equivalent to $(\blacklozenge_I \mathit{True}) \wedge (\varphi \; T_I \; \psi)$ and $(\lozenge_I \mathit{True}) \wedge (\varphi \; R_I \; \psi)$ for formulas φ and ψ and an interval I. If the interval I includes zero, then the formulas $\blacklozenge_I \mathit{True}$ and $\lozenge_I \mathit{True}$ are trivially satisfied and could be dropped since the time point for which the formula is evaluated is always included in the interval. An alternative approach would be to replace $\blacklozenge_I \mathit{True}$ and $\lozenge_I \mathit{True}$ by a safe formula α satisfying $(\mathit{free}(\varphi) \cup \mathit{free}(\psi)) \subseteq \mathit{free}(\alpha)$. This ensures that in case the interval I does not contain any time points, the set of satisfying assignments to the free variables is already restricted to be finite by the safe formula α.

Before presenting formulas equivalent to Trigger and Release, alternative formulas for Historically and Eventually are presented as they will be used as building blocks when translating the operators Trigger and Release. When looking at their definitions (see Figure 3.1) in terms of Once and Eventually rather than Trigger and Release, it becomes clear that we are facing the same issue as with Trigger and Release: There is a top-level negation which imposes very strong safety conditions if the definition was used directly.

$$\begin{aligned} \blacksquare_I \varphi &:= \neg(\blacklozenge_I(\neg\varphi)) \\ \square_I \varphi &:= \neg(\diamondsuit_I(\neg\varphi)) \end{aligned}$$

Figure 3.1: Alternative definitions of Historically and Always

3.2.1 Historically

We begin with the case where the interval contains zero. Then it is possible to distinguish the cases whether there exists a time point outside the interval $[0, b)$. If there exists one, the formula $\varphi\ S_{[b,\infty)}\ \left(\bigcirc_{[0,\infty)}\varphi\right)$ captures the semantics of $\blacksquare_I \varphi$ as $0 \in I$ guarantees the existence of a time point within the interval. If there is no time point outside the interval, then φ has to hold since the first entry in the temporal structure. The first entry in the temporal structure can be targeted with $\neg\left(\bullet_{[0,\infty)}\ True\right)$ as it is the only time point without a predecessor.

Definition 3.1 Let $first$ be the MFOTL formula $\neg\left(\bullet_{[0,\infty)}\ True\right)$.

Lemma 3.2 $first$ is only satisfied at the first time point in a temporal structure. More formally:

$$(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models first \Leftrightarrow i = 0$$

The proof of this directly follows from the semantics in Definition 2.3.

Historically for an interval of the form $[0, b)$ can be expressed as

If $b \neq \infty$, then

$$\blacksquare_{[0,b)}\ \varphi = \underbrace{\left(\varphi\ S_{[b,\infty)}\ (\bigcirc_{[0,\infty]}\varphi)\right)}_{\text{There exists a time point outside } [0,b)} \vee \underbrace{\left(\varphi\ S_{[0,\infty)}\ (\varphi \wedge first)\right)}_{\text{All earlier time points are in } [0,b)}$$

If $b = \infty$, then

$$\blacksquare_{[0,b)}\ \varphi = \left(\varphi\ S_{[0,\infty)}\ (\varphi \wedge first)\right)$$

If zero is not part of the interval, one has to be more cautious as the same issue as in Trigger and Release resurfaces again: If there exists no time point within the interval, Historically is trivially satisfied and hence cannot be safe. Thus, analogously to Trigger and Release, we seek a formula semantically equivalent to $\blacklozenge_{[a,b)}\ \varphi \wedge \blacksquare_{[a,b)}\ \varphi$ for $a > 0$. (Note that in the case of Historically it does not matter whether we use $\blacklozenge_{[a,b)}\ \varphi$ or $\blacklozenge_{[a,b)}\ True$ as in conjunction with Historically the two are equivalent)

If the interval is unbounded, i.e. $b = \infty$, then an analogous approach as in the case $[0, \infty)$ can be used because φ must hold since the first time point. The only difference is that the last time point in the interval $[a, \infty)$ should be targeted and $\varphi\ S_{[0,\infty)}\ (\varphi \wedge first)$ has to be shifted by combining a ◆ (Once) and a ● (Previous) operator.

$$\blacklozenge_{[a,\infty)}\ \varphi \wedge \blacksquare_{[a,\infty)}\ \varphi = (\blacklozenge_{[a,\infty)}\ \varphi) \wedge \left(\blacklozenge_{[0,a)}\ \left(\bullet_{[0,\infty)}\ \left(\varphi\ S_{[0,\infty)}\ (\varphi \wedge first)\right)\right)\right)$$

The last remaining case, the one of a bounded interval $[a, b)$ with $a > 0$ and $b \neq \infty$ is much more tricky. Although the use of a negation in general is dangerous if the resulting formula should be safe, M. Raszyk has come up with the following trick:

$$\blacklozenge_{[a,b)}\ \varphi \wedge \blacksquare_{[a,b)}\ \varphi = \left(\blacklozenge_{[a,b)}\ \varphi\right) \wedge \left(\neg\left(\blacklozenge_{[a,b)}\ \left(\left(\blacklozenge_{[0,b-a)}\ \varphi\right) \vee \left(\diamondsuit_{[0,b-a)}\ \varphi\right)\right) \wedge \neg\varphi\right)\right)$$

First it is important to note that this formula is indeed safe. According to the equations in Figure 2.3 and Lemma 2.5, a conjunction with a negated right hand side is safe if the negated subformula is safe itself and the free variables of the negated subformula are included among the free variables of the left-hand side of the conjunction which is the case here.

Next an informal argument for the semantic equivalence is given. Apart from the left hand side of the inner conjunction, $\left(\left(\blacklozenge_{[0,b-a)}\ \varphi\right) \vee \left(\diamondsuit_{[0,b-a)}\ \varphi\right)\right)$, the formula is exactly the definition of Historically in terms of Once (see Figure 3.1). Hence if $\left(\left(\blacklozenge_{[0,b-a)}\ \varphi\right) \vee \left(\diamondsuit_{[0,b-a)}\ \varphi\right)\right)$ is always satisfied within the interval, then the two formulas are semantically equivalent.

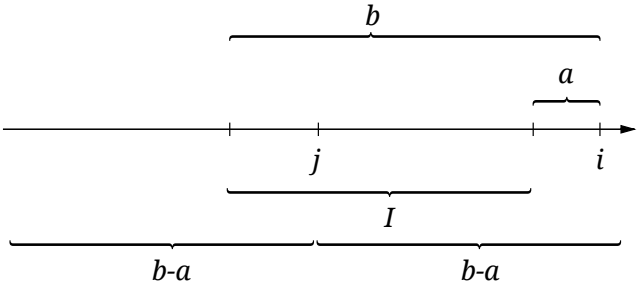

Figure 3.2: Example for the translated formula for Historically in the case where the interval is bounded and $a > 0$ is given

Let i be the current time point where the formula is evaluated at. It is given that the time point j where the subformula $\left(\left(\blacklozenge_{[0,b-a)}\ \varphi\right) \vee \left(\diamondsuit_{[0,b-a)}\ \varphi\right)\right)$ is

evaluated at, is located inside the interval $I = [a, b)$ as it is the argument to a Once with said interval I. Moreover there exists a time point k inside the interval where φ is satisfied. As both time points lie within the interval, the one satisfying φ must be within the range of the length of interval $(b - a)$ for any time point j inside the interval. An example of this scenario is depicted in Figure 3.2.

Now this rewrite formula would be correct but because of the actual formalization of intervals (described in section 2.6), the interval $[0, b - a)$ cannot be efficiently computed from the interval $[a, b)$. One can notice that the idea still works, if the larger interval $[0, b)$ is used instead, i.e. $\left(\left(\blacklozenge_{[0,b)}\ \varphi\right) \vee \left(\diamondsuit_{[0,b)}\ \varphi\right)\right)$. Given $[a, b)$, this interval can be constructed in the formalization by simply setting the function `memL` to the constant function returning `True`. As it is not clear how the interval $[0, b - a)$ can be computed efficiently, we use this less restrictive formula instead.

With these four rewrite formulas, an alternative Historically operator $\blacksquare'_I\ \varphi$ that for some formula φ and interval I is semantically equivalent to $\blacklozenge_I\ \varphi \wedge \blacksquare_I\ \varphi$ can be defined. It is safe exactly when the subformula φ is which is the best one could hope for without rewriting the formula φ itself.

Definition 3.3 The unary temporal operator $\blacksquare'_I\ \varphi$ for the interval $[a, b)$ is defined as follows:

If $a = 0 \wedge b \neq \infty$, then

$$\blacksquare'_{[0,b)}\ \varphi = \left(\varphi\ S_{[b,\infty)}\ (\bigcirc_{[0,\infty]}\ \varphi)\right) \vee \left(\varphi\ S_{[0,\infty)}\ (\varphi \wedge first)\right)$$

If $a = 0 \wedge b = \infty$, then

$$\blacksquare'_{[0,b)}\ \varphi = \left(\varphi\ S_{[0,\infty)}\ (\varphi \wedge first)\right)$$

If $a > 0 \wedge b = \infty$, then

$$\blacksquare'_{[a,\infty)}\ \varphi = \left(\blacklozenge_{[a,\infty)}\ \varphi\right) \wedge \left(\blacklozenge_{[0,a)}\ \left(\bullet_{[0,\infty)}\ \left(\varphi\ S_{[0,\infty)}\ (\varphi \wedge first)\right)\right)\right)$$

If $a > 0 \wedge b \neq \infty$, then

$$\blacksquare'_{[a,b)}\ \varphi = \left(\blacklozenge_{[a,b)}\ \varphi\right) \wedge \left(\neg\left(\blacklozenge_{[a,b)}\ \left(\left(\blacklozenge_{[0,b)}\ \varphi\right) \vee \left(\diamondsuit_{[0,b)}\ \varphi\right)\right) \wedge \neg\varphi\right)\right)$$

The formalization in Isabelle / HOL [6] proves that $\blacksquare'_{[a,b)}\ \varphi$ is semantically equivalent to $(\blacklozenge_I\ \varphi) \wedge \left(\blacksquare_{[a,b)}\ \varphi\right)$:

Lemma 3.4 For any interval $I \in \mathbb{I}$ and any formula φ, the following equivalence holds for any $\bar{\mathcal{D}}, \bar{\tau}, v, i$:

$$(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models (\blacklozenge_I \varphi) \wedge (\blacksquare_I \varphi) \Leftrightarrow (\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \blacksquare'_I \varphi$$

Moreover, the formalization also includes a proof for the following lemma:

Lemma 3.5 The formula $\blacksquare'_{[a,b)}\ \varphi$ is part of the subset of safe formulas exactly when φ is.

$$\blacksquare'_{[a,b)}\ \varphi \text{ is } \textit{safe} \Leftrightarrow \varphi \text{ is } \textit{safe}$$

3.2.2 Always

Since formulas with an unbounded interval into the future cannot be monitored, we restrict ourselves to the case of bounded intervals. For $a > 0$ the task is tackled analogous to Historically but for $a = 0$ a different idea has to be used as it is not clear how to target the "last" time point. One can make use of the fact that a temporal structure guarantees that the sequence of timestamps will always grow beyond any bound. Assuming the interval is not empty, there either is a timestamp within the interval $[b, 2 \cdot b - 1)$ meaning $\left(\varphi\, U_{[b,2\cdot b-1)} \left(\bullet_{[0,\infty)}\, \varphi\right)\right)$ must hold or there is none in there. If there are no timestamps within the interval $[b, 2 \cdot b - 1)$, then φ must be satisfied up until the time point where the next time point is at least b time units apart from the last time point within the interval $[0, b)$. The reason being that $b - 1$ is the greatest possible timestamp difference to the current timestamp that is within the interval $[a, b)$. Moreover if $[b, 2 \cdot b - 1)$ is empty, the next possible timestamp has at least $2 \cdot b - 1$ units of time difference to the current one. Hence the difference between the last time point within the interval $[0, b)$ and the first time point within $[b, 2 \cdot b - 1)$ is at least $2 \cdot b - 1 - (b - 1) = b$ units of time. The two scenarios are depicted in Figures 3.3 and 3.4.

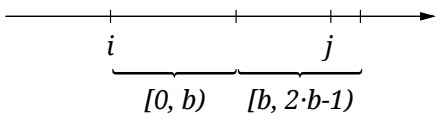

Figure 3.3: Example of the case when the interval $[b, 2 \cdot b - 1)$ contains a time point j.

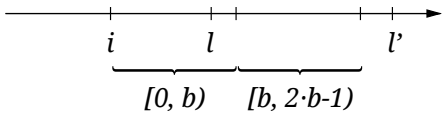

Figure 3.4: Example of the case where the interval $[b, 2 \cdot b - 1]$ is empty. Time points l and l' represent the last entry in the interval $[0, b)$ and the first entry with a timestamp difference at last $2 \cdot b - 1$, respectively. Their distance is at least b units of time.

The latter case can be captured by $\Big(\varphi \ U_{[0,b)} \ \Big(\varphi \wedge \Big(\bigcirc_{[b,\infty)} \mathit{True}\Big)\Big)\Big)$.

Analogous to definition 3.3, an alternative Always operator $\Box'$ is defined which is semantically equivalent to $\Box$ in conjunction with $\Diamond$ and *safe* if its argument is.

Definition 3.6 The unary temporal operator $\Box'_I \, \varphi$ for the bounded interval $[a, b), b \neq \infty$ is defined as follows:

If $a = 0$, then

$$\Box'_{[0,b)} \ \varphi = \Big(\varphi \, U_{[b,2\cdot b-1)} \ \Big(\bullet_{[0,\infty)} \, \varphi\Big)\Big) \vee \Big(\varphi \ U_{[0,b)} \ \Big(\varphi \wedge \Big(\bigcirc_{[b,\infty)} \mathit{True}\Big)\Big)\Big)$$

If $a > 0$, then

$$\Box'_{[a,b)} \ \varphi = \Big(\Diamond_{[a,b)} \, \varphi\Big) \wedge \Big(\neg \Big(\Diamond_{[a,b)} \, \Big(\Big(\blacklozenge_{[0,b)} \, \varphi\Big) \vee \Big(\Diamond_{[0,b)} \, \varphi\Big)\Big) \wedge \neg\varphi\Big)\Big)$$

It is proven in the formalization in Isabelle / HOL [6] that $\Box'_I \ \varphi$ and $(\Diamond_I \, \varphi) \wedge (\Box_I \ \varphi)$ are semantically equivalent:

Lemma 3.7 For any interval $I \in \mathbb{I}$ and any formula φ, the following equivalence holds for any $\bar{\mathcal{D}}, \bar{\tau}, v, i$:

$$(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models (\Diamond_I \, \varphi) \wedge (\Box_I \, \varphi) \Leftrightarrow (\bar{\mathcal{D}}, \bar{\tau}, v, i) \models (\Box'_I \, \varphi)$$

As for Historically, the formalization includes a proof about the safety:

Lemma 3.8 $\Box'_{[a,b)} \ \varphi$ is part of the subset of safe formulas exactly when φ is.

$$\Box'_{[a,b)} \ \varphi \text{ is } \mathit{safe} \Leftrightarrow \varphi \text{ is } \mathit{safe}$$

3.2.3 Trigger

We begin by discussing the different cases of satisfactions for a Trigger operator $\varphi\ T_I\ \psi$ for some formulas φ and ψ and an interval $I \in \mathbb{I}$ at some time point i.

It is possible that there exists a time point within $[0, a)$ where φ holds. This immediately satisfies the semantics of Trigger (see Definition 3.1). This condition directly corresponds to a once: $\blacklozenge_{[0,a)}\ \varphi$.

Moreover if there is no time point within $[0, a)$ where φ holds, there either is a time point within $I = [a, b)$ where φ is satisfied or there is none in $[0, b)$. Assuming there is a time point inside the interval $I = [a, b)$ where φ holds, let j be the latest such time point. Then ψ only has to hold at j and all later time points up until the end of the interval. This can also be expressed with a Since: $\psi\ S_{[0,\infty)}\ (\psi \wedge \varphi)$. But this condition is stronger than the one imposed by Trigger as the semantics of Trigger only quantify over the time points within the interval. Hence the time point where this just derived Since formula should hold, must be shifted. This can be accomplished by combining a Once with a Previous operator: Let the interval I be equal to $[a, b)$. Then $\blacklozenge_{[0,a)} \left(\bullet_{[0,\infty)}(\psi\ S_{[0,\infty)}\ (\psi \wedge \varphi))\right)$ imposes the condition $\psi\ S_{[0,\infty)}\ (\psi \wedge \varphi)$ on the last entry within the interval I.

Finally, if there is no time point within the interval $[0, b)$, the last remaining way for the semantics of Trigger to be satisfied is if the right hand side ψ holds at all time points within the interval $I = [a, b)$. This part is semantically equivalent to a Historically $\blacksquare_I\ \psi$.

Now of course this description is not even close to a formal proof but it should give the reader an intuition where the translated formula comes from. Next we define the translation operator $\varphi\ T'_I\ \psi$ that is semantically equivalent to $(\blacklozenge_I\ True) \wedge (\varphi\ T_I\ \psi)$ and allows a relaxation of the safety conditions.

Definition 3.9 The binary MFOTL operator T' for the interval $[a, b)$ is defined as follows:

If $a = 0$, then

$$\varphi\ T'_{[0,b)}\ \psi = \left(\blacksquare'_{[0,b)}\ \psi\right) \vee \left(\psi\ S_{[0,\infty)}\ (\psi \wedge \varphi)\right)$$

If $a > 0$, then

$$\varphi\ T'_{[a,b)}\ \psi = \left(\blacklozenge_{[a,b)}\ True\right) \wedge \left(\left(\blacksquare'_{[a,b)}\ \psi\right) \vee \left(\blacklozenge_{[0,a)}\ \varphi\right) \vee \left(\blacklozenge_{[0,a)} \left(\bullet_{[0,\infty)} \left(\psi\ S_{[0,\infty)}\ (\psi \wedge \varphi)\right)\right)\right)$$

Note the usage of $\blacksquare'_I$ instead of $\blacksquare_I$.

The operator $\varphi\ T'_I\ \psi$ can be proven to be semantically equivalent to $\varphi\ T_I\ \psi$ conjoined with Once:

Lemma 3.10 For any interval $I \in \mathbb{I}$ and any formulas φ and ψ, the following equivalence holds for any $\bar{\mathcal{D}}, \bar{\tau}, v, i$:

$$(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models (\blacklozenge_I \mathit{True}) \wedge (\varphi\ T_I\ \psi) \Leftrightarrow (\bar{\mathcal{D}}, \bar{\tau}, v, i) \models (\varphi\ T'_I\ \psi)$$

This lemma is proven in the formalization in Isabelle / HOL [6].

3.2.4 Release

An analogous intuition applies to Release as well since the semantics are symmetric to the ones for Trigger. Please note that the case of an unbounded interval is ignored as it is not possible to monitor it anyway.

A formula $\varphi\ R_{[a,b)}\ \psi$ is satisfied if either ψ is satisfied in the whole interval $[a, b)$, $\psi\ U_{[0,\infty)}\ (\psi \wedge \varphi)$ holds at the smallest time point inside the interval or there exists a time point within $[0, a)$ where φ is satisfied. Hence the operator R' is analogously defined to T'.

Definition 3.11 The binary MFOTL operator R'_I for the bounded interval $I = [a, b)$ is defined as follows:

If $a = 0$, then

$$\varphi\ R'_{[0,b)}\ \psi = \left(\Box'_{[0,b)}\ \psi\right) \vee \left(\psi\ U_{[0,\infty)}\ (\psi \wedge \varphi)\right)$$

If $a > 0$, then

$$\varphi\ R'_{[a,b)}\ \psi = \left(\Box'_{[0,b)}\ \psi\right) \vee \left(\Diamond_{[0,a)}\ \varphi\right) \vee \left(\Diamond_{[0,a)}\ \left(\bigcirc_{[0,\infty)}\ \left(\psi\ U_{[0,\infty)}\ (\psi \wedge \varphi)\right)\right)\right)$$

Again, note the usage of $\Box'_I$ instead of $\Box_I$.

And once again, the following lemma is proven in the formalization in Isabelle / HOL [6]:

Lemma 3.12 For any *bounded* interval $I \in \mathbb{I}$ and any formulas φ and ψ, the following equivalence holds for any $\bar{\mathcal{D}}, \bar{\tau}, v, i$:

$$(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models (\Diamond_I \mathit{True}) \wedge (\varphi\ R_I\ \psi) \Leftrightarrow (\bar{\mathcal{D}}, \bar{\tau}, v, i) \models (\varphi\ R'_I\ \psi)$$

3.3 Deriving safety conditions

Based on the definitions of the operators T' and R' it is now possible to derive the conditions under which their evaluation is safe by reducing them to the cases of `safe_formula` given in Figure 2.3. As the two operators are defined symmetrically with respect to Since and Until and the safety conditions for Since and Until are the same, the safety conditions derived for both Trigger and Release are the same as well. The conditions are abstracted by the following predicate on two formulas and an interval:

Definition 3.13 Let `safe_dual` be the predicate defined by

$$\texttt{safe\_dual}(\varphi, I, \psi) = \begin{cases} \texttt{safe\_formula}(\psi) \wedge & \text{if } 0 \in I \\ free(\varphi) \subseteq free(\psi) \wedge (& \\ \quad \texttt{safe\_formula}(\varphi) \vee & \\ \quad (\exists \varphi'.\ \varphi = \neg\varphi' \wedge \texttt{safe\_formula}\,(\varphi')) & \\ ) & \\ \texttt{safe\_formula}(\varphi) \wedge & \text{otherwise} \\ \texttt{safe\_formula}(\psi) \wedge & \\ free(\varphi) = free(\psi) & \end{cases}$$

Using this definition, the safety conditions for the operators T' and R' are given by the following lemma.

Lemma 3.14 The evaluation of the operators T' and R' is safe, given `safe_dual` is satisfied with the following arguments:

$$\forall \varphi\ I\ \psi.\ \texttt{safe\_dual}(\varphi, I, \psi) \rightarrow \texttt{safe\_formula}(\varphi\ T'_I\ \psi)$$
$$\forall \varphi\ I\ \psi.\ \texttt{safe\_dual}(\varphi, I, \psi) \rightarrow \texttt{safe\_formula}(\varphi\ R'_I\ \psi)$$

Note that these are implications, not equivalences. For an equivalence `safe_dual` would have to allow the additional cases $\texttt{safe\_assignment}(free(\psi), \varphi)$ and $\texttt{is\_constraint}(\psi)$. For simplicity these cases are omitted and can be added in future work.

The proof of this lemma follows from definition 3.13 and the cases of `safe_formula` given in Figure 2.3. It is also been verified using Isabelle / HOL.

Now we informally argue why these safety conditions seem reasonable. As already discussed, the two operators are satisfied if the right-hand side ψ

holds at all time points within the interval. Moreover if zero is not part of the interval $I = [a, b)$, one can see in the definitions of T' (Definition 3.9) and R' (Definition 3.11) that the satisfaction of the left-hand side φ in the interval $[0, a)$ suffices for the respective operator to be satisfied. Thus in the case of $0 \notin I$ it is for both subformulas possible to cause the operator to be satisfied independent of the other subformula. As a result, the set of free variables of the two subformulas must be the same. Assuming it was not, at least one subformula $\alpha \in \{\varphi, \psi\}$ would have a free variable x that is not part of the set of the free variables of the other subformula $\beta \in \{\varphi, \psi\} \backslash \{\alpha\}$. In case β then causes the operator to be satisfied, any value can be assigned to x and since x is not an element of $free(\beta)$, the operator would still be satisfied. This would of course lead to an infinite table and hence a contradiction to Lemma 3.14. Hence both subformulas must have the same set of free variables and because the definition of T'_I does not apply rewriting to the subformulas recursively they both must be safe as well.

If $0 \in I$, the safety conditions can be relaxed as the satisfaction of the left-hand side φ alone cannot cause the operator to be satisfied and hence it suffices for the set of free variables to be a subset and it is also possible to allow a negated formula on the left-hand side.

3.4 Formalization

This section describes the formalization in Isabelle and hence the properties of the exported code using the translated formulas.

In the previous sections semantically equivalent formulas to Trigger and Release were derived which resulted in looser safety conditions than the predicate `safe_formula` would have allowed based on their definition. By translating the operators to the semantically equivalent formulas, they can now be monitored by VeriMon under the just derived safety conditions without many additional changes.

The idea originally was to extend the definition of the predicate `safe_formula` to include the cases $(\blacklozenge_I \mathit{True}) \wedge (\varphi\ T_I\ \psi)$ for formulas φ, ψ and interval I given the safety conditions described in section (3.3) are satisfied.

Detecting such a formula would require deep pattern-matching, in particular because *True* is not part of the defined `formula` data type. Hence we decided that the evaluation of Trigger and Release operators that do not occur in a conjunction with a safe formula containing all the free variables of the Trigger or Release operators are only allowed if zero is part of their interval. The evaluation of Trigger and Release over intervals excluding zero are then allowed under a conjunction where the degenerate case of the interval not containing a time point can be handled as well.

In particular, the definition of `safe_formula` for conjunctions is extended to

$$
\begin{aligned}
\texttt{safe\_formula}\,(\varphi \wedge \psi) \quad \Leftrightarrow \quad & \texttt{safe\_formula}(\varphi) \wedge (\\
& \quad \texttt{safe\_assignment}\,(\mathit{free}(\varphi), \psi) \vee \\
& \quad \texttt{safe\_formula}\,(\psi) \vee \\
& \quad \mathit{free}(\psi) \subseteq \mathit{free}(\varphi) \wedge (\\
& \quad\quad \texttt{is\_constraint}(\psi) \vee \\
& \quad\quad \exists \psi'.\psi = \neg\psi' \wedge \texttt{safe\_formula}(\psi') \vee \\
& \quad\quad \underbrace{\exists \varphi'\, I\, \psi'.\psi = \varphi'\, T_I\, \psi' \wedge \texttt{safe\_dual}(\varphi', I, \psi')}_{\text{This part is new, compare to Figure 2.3}} \vee \\
& \quad\quad \underbrace{\exists \varphi'\, I\, \psi'.\psi = \varphi'\, R_I\, \psi' \wedge \texttt{safe\_dual}(\varphi', I, \psi')}_{\text{This part is new, compare to Figure 2.3}} \\
& \quad) \\
&)
\end{aligned}
$$

whereas the definition of `safe_formula` is extended by the following two cases:

$$
\begin{aligned}
\texttt{safe\_formula}\,(\varphi\, T_I\, \psi) \quad &\Leftrightarrow \quad 0 \in I \wedge \texttt{safe\_dual}(\varphi, I, \psi) \\
\texttt{safe\_formula}\,(\varphi\, R_I\, \psi) \quad &\Leftrightarrow \quad 0 \in I \wedge \texttt{safe\_dual}(\varphi, I, \psi)
\end{aligned}
$$

Note that the actual definition in the formalization is slightly different for the case of Release. The reason being that no specialized algorithm is developed for Release and all occurrences of the operator are translated to the semantically equivalent formulas. Thus the safety conditions for Release are defined recursively on the translated formulas which allows for easier (inductive) proofs in Isabelle / HOL.

So far we have only shown formulas semantically equivalent to $(\blacklozenge_I \mathit{True}) \wedge (\varphi\, T_I\, \psi)$ and $(\Diamond_I \mathit{True}) \wedge (\varphi\, R_I\, \psi)$ for some formulas φ, ψ and an interval I, not formulas equivalent to $\alpha \wedge (\varphi\, T_I\, \psi)$ and $\alpha \wedge (\varphi\, R_I\, \psi)$ for some formula α.

Fortunately, the operators T' and R' can be easily adapted to this case:

Lemma 3.15 For any $\bar{\mathcal{D}}, \bar{\tau}, v, i$, the conjunctions of the dual operators Trigger and Release with a formula α are semantically equivalent to the following disjunctions:

$$
\begin{aligned}
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \alpha \wedge (\varphi\, T_I\, \psi) &\Leftrightarrow (\bar{\mathcal{D}}, \bar{\tau}, v, i) \models (\alpha \wedge \neg\,(\blacklozenge_I \mathit{True})) \vee \left(\alpha \wedge \left(\varphi\, T'_I\, \psi\right)\right) \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \alpha \wedge (\varphi\, R_I\, \psi) &\Leftrightarrow (\bar{\mathcal{D}}, \bar{\tau}, v, i) \models (\alpha \wedge \neg\,(\Diamond_I \mathit{True})) \vee \left(\alpha \wedge \left(\varphi\, R'_I\, \psi\right)\right)
\end{aligned}
$$

Moreover the two translated formulas are safe given the original conjunctions were. The proof of this lemma directly follows from a case distinction on whether the interval contains a time point or not.

These definitions then allow the extension of the monitoring algorithm. Instead of using a custom data-structure to monitor the operator, the original monitoring algorithm is invoked on the translated formulas.

Chapter 4

Direct Evaluation of Trigger

Instead of translating the operators to semantically equivalent formulas one can design a specialized algorithm in order to evaluate them as it was done for the other operators that are supported by VeriMon [11, 24].

Due to the time constraints given for this thesis, no specialized algorithm is developed for Release, just one for Trigger. The core ideas are heavily inspired by the semantically equivalent formulas derived in Chapter 3 (see Definition 3.9) and the specialized algorithm designed for the evaluation of Since [11].

The interface of the specialized monitoring algorithm for Trigger is described by the *locale* (an interface in Isabelle / HOL) `mtaux`. It consists of three functions `init_mtaux`, `update_mtaux` and `result_mtaux` analogous to the interface of `msaux`, the locale for evaluating the operator Since [11]. The types of the three functions are listed below.

$$\begin{aligned} \texttt{init\_mtaux} &:: args \Rightarrow mtaux \\ \texttt{update\_mtaux} &:: args \Rightarrow ts \Rightarrow table \Rightarrow table \Rightarrow mtaux \Rightarrow mtaux \\ \texttt{result\_mtaux} &:: args \Rightarrow mtaux \Rightarrow nat\,set \times table \end{aligned}$$

In contrast to `mtaux`, the interface of `msaux` is more refined. Namely the function performing the update of the data-structure is split into multiple stages.

The parameter *args* contains static information independent of the trace required for the initialization and evaluation of the data-structure such as the interval I or the set of free variables of the left- and the right-hand side of $\varphi\, T_I\, \psi$ for some formulas φ and ψ. The details are omitted.

The type *ts* stands for timestamp and is an abbreviation for a natural number, $nat = \mathbb{N}$. The type *table* corresponds to a finite table containing all satisfying assignments to the free variables (*tuples*) as described in Section

2.3. The function `init_mtaux` initializes the data-structure whereas the function `update_mtaux` updates the data-structure given a new entry of the trace where *ts* is the timestamp and the two tables correspond to the two safe subformulas of the Trigger operators. If the left-hand side is a negation then the finite table for the left-hand side corresponds to the subformula of the negation. Finally the result of the function `result_mtaux` encodes the set of tuples satisfying the Trigger operator.

The major difference to the interface of `msaux` is that `result_mtaux` not only outputs the table of satisfying assignments to the free variables but also the set of free variables this table is over. By having the set of free variables the table is over dynamic, it is possible to return the table for the closed formula *True*. Hence, in case the interval does not contain any time points, this table can be returned which conveys the information that the operator is trivially satisfied.

4.1 Correctness

The way the correctness of the algorithm is shown follows the approach used to prove the correctness of `msaux`, the locale describing the evaluation of Since: An invariant is defined that is initially established by `init_mtaux` and is preserved by the execution of `update_mtaux`. Moreover it is shown that the invariant guarantees that the function `result_mtaux` outputs the results described by the semantics of Trigger.

In the development of the optimized data-structure for Since, these properties were formulated with respect to an existing data-structure [11, 24]. In contrast to Since, there was no existing data-structure for the evaluation of Trigger and hence these properties were formulated with respect to a non-executable specification. More specifically, the invariant for the evaluation of a Trigger operator $\varphi\ T_I\ \psi$ for some formulas φ, ψ and an interval $I = [a, b)$ was written with respect to a trivial data-structure: A list that would simply store all input tables together with their associated timestamps as long as the timestamp's difference to the current one is less than b. Because this trivial data-structure stores all data within the interval $[0, b)$, the satisfying assignments to the free variables for $\varphi\ T_I\ \psi$ can directly be described using the derived semantics of Trigger (see Equation 3.1). Note that this description does not have to be executable as this trivial data-structure is only used to prove the correctness of the algorithm.

Let the predicate `valid_mtaux` with the signature

$$\texttt{valid\_mtaux} :: \mathit{args} \Rightarrow \mathit{ts} \Rightarrow \mathit{mtaux} \Rightarrow (\mathit{ts} \times \mathit{table} \times \mathit{table})\ \mathit{list} \Rightarrow \mathit{bool}$$

be the invariant the locale `mtaux` should satisfy with respect to an instance of the trivial data-structure with type $\mathit{ts} \times \mathit{table} \times \mathit{table}$, as just described. The

first table corresponds to the set of satisfying assignments for φ (or the set of satisfying assignments to φ' if $\varphi = \neg\varphi'$ for some φ') whereas the second to the ones for ψ. The timestamp (ts) argument corresponds to the respective timestamp.

Assumptions in the locale `mtaux` guarantee `init_mtaux` establishes this invariant and that it is preserved after calling `update_mtaux` with the new timestamp `nt`, the set of tuples `l` satisfying φ (or the subformula of φ' if $\varphi = \neg\varphi'$ for some formula φ') and the set of tuples `r` satisfying ψ at the new time point:

```
  valid_init_mtaux : safe_args args ⇒
                     valid_mtaux args 0 (init_mtaux args) []
valid_update_mtaux : nt ≥ t ⇒
                     valid_mtaux args t mtaux trivial_list ⇒
                     valid_mtaux
                       args
                       nt
                       (update_mtaux args nt l r mtaux)
                       ((filter f_I,nt trivial_list) @ [(nt, l, r)])
```

where `safe_args` checks whether the safety conditions described in Section 3.3 are satisfied. The function `safe_args` does not exist in the formalization, it is used as a shorthand in this thesis to abstract the underlying safety conditions imposed on the arguments.

Some additional preconditions about the wellformedness of the tables are omitted as they do not contribute to the understanding of the assumptions.

The function $f_{I,\mathtt{nt}}$ filters the trivial data-structure to remove all entries with timestamps differences with respect to the new timestamp `nt` that are no longer in the interval $[0, b)$. Moreover the at-symbol `@` denotes list concatenation meaning `((filter` $f_{I,\mathtt{nt}}$ `trivial_list) @ [(nt, l, r)])` corresponds to `trivial_list` being filtered and then the additional element `(nt, l, r)` being appended.

The last assumption of the locale guarantees the validity of the results returned:

```
valid_mtaux args t mtaux trivial_list ⇒
result_mtaux args mtaux = trigger_results args t trivial_list
```

where `trigger_results` corresponds to a function describing the results of the Trigger operator analogous to the semantics expressed in Equation 3.1.

4.2 Underlying Data-Structure and Invariant

In the state for an evaluation of $\varphi\ T_I\ \psi$ for some formulas φ, ψ and an interval $I = [a, b)$, the list of tables is split into two lists sorted by the timestamps: `data_prev` containing the tables with timestamps that are not in the interval yet and `data_in` containing the tables with timestamps that are within the interval analogous to the state for Since [11]. What is different compared to the lists in the optimized state for Since, is that the list `data_prev` now contains the tables for both subformulas. The list `data_in` still contains just the tables of the right-hand side ψ.

In addition, the state contains a mapping `tuple_in_once` that maps tuples satisfying φ to the newest timestamp for which this tuple occurs in a table in `data_prev`. This is analogous to the mapping `data_in` used in the optimized state for Since with the difference that the mapping `tuple_in_once` is used for the interval $[0, a)$, not $I = [a, b)$. When looking at definition 3.9 for the general case, one can see that the set of keys of this mapping directly corresponds to the tuples satisfying the subformula $\left(\blacklozenge_{[0,a)}\ \varphi\right)$. Figure 4.1 shows an example how the contents of this mapping might change over time. In the example shown, the Interval is $[3, 6)$ meaning at every time point the mapping `tuple_in_once` contains the tuples satisfying the atomic predicate P inside the interval $[0, 3)$.

tp	ts	database	`data_prev`	`tuple_in_once`
0	1	{P(a), Q(c)}	[{(1, {a}, {c})}]	{a ↦ 1 }
1	2	{P(a), Q(a), Q(b)}	[{(1, {a}, {c})}, {(2, {a}, {a, b})}]	{a ↦ 2}
2	3	{P(c), Q(a)}	[{(2, {a}, {a, b})}, {(3, {c}, {a})}]	{a ↦ 2, c ↦ 3}
3	5	{Q(a), Q(c)}	[{(3, {c}, {a})}, {(5, {}, {a, c})}]	{c ↦ 3}
4	6	{Q(a), Q(d)}	[{(5, {}, {a, c})}, {(6, {}, {a, d})}]	{}

Figure 4.1: An example of the contents of `tuple_in_once` for the formula $P(x)\ T_{[3,6)}\ Q(x)$ for the unary atomic predicates P and Q and tuples a, b, c, d. The keys of `tuple_in_once` satisfy the formula $\blacklozenge_{[0,3)}\ P(x)$. The abbreviation ts stands for timestamp and tp for time point

At time points 0, 1 and 2 the mapping is updated for the tuples a and c as the atomic predicate P is satisfied (element of the database) for said tuples in the respective time points. At time points 3 and 4 the tuples a and c are respectively removed from the mapping `tuple_in_once` because the timestamp they are mapped to is at least 3 (upper bound of the interval) units of time away in the past. If 0 is in the interval, then `data_prev` will always be empty and hence the mapping `tuple_in_once` will always be empty.

Moreover, the state also includes a mapping `tuple_since_hist` and a set `hist_sat` which is designed to compute the tuples satisfying the subformula $\left(\blacksquare'_{[a,b)}\ \psi\right)$ (see Definition 3.9). Similar to the mapping `tuple_since` used in the optimized state for evaluating a Since operator [11], `tuple_since_hist` maps tuples to the smallest time point in `data_in` since when all entries with larger or equal time points in `data_in` include said tuple in the table corresponding to ψ (i.e. a given tuple satisfies ψ since that time point). The set `hist_sat` simply contains all tuples for which the time point assigned by mapping `tuple_since_hist` is at most the time point of the oldest entry in `data_in`, i.e. ψ is satisfied for all time points inside the interval. Figure 4.2 depicts an example of how the contents of `tuple_since_hist` and `hist_sat` evolve over time for the shown example trace.

The depicted example (Figure 4.2) shows how contents of the mapping `tuple_since_hist` and the set `hist_sat` change during the evaluation of the formula $P(x)\ T_{[1,3)}\ Q(x)$. Initially both of them are empty. At time point 1 with timestamp 4, the table from time point 0 with timestamp 3 now is in `data_in` because the difference between the current timestamp (4) and the timestamp for time point 0 (3) is part of the interval $[1, 3)$. The mapping `tuple_since_hist` is updated to include the tuple b and points to the time point 0 as all tables in `data_in` after (and including) time point zero up to the newest one in the interval (also time point zero in this case) include the tuple b. In addition `hist_sat` is updated because the time point `tuple_since_hist` maps b to is at most equal to the oldest time point within the interval. At time point 2, the entry of time point 1 with timestamp 4 enters `data_in` as well and `tuple_since_hist` is updated for a to point to time point 1. The reason being the same as before: After (and including) time point 1 up to the newest one in `data_in` (1), the tuple a is included in all tables in `data_in`. The same holds for the tuple b but because we want the mapping to map to the earliest time point where this condition is satisfied, its value for b is not updated. The set `hist_sat` remains unchanged as all tuples inside of it are part of the new table in `data_in` and there is no new tuple in `tuple_since_hist` that maps to a time point that is at most equal to the time point associated with the oldest entry in `data_in`. The changes for the next few time points follow the same reasoning. Note that the interval $[1, 3)$ does not include 3 and thus at time point 3 with timestamp 6, the difference to the entry with timestamp

tp	ts	database	oldest tp within the interval	newest tp within the interval	data_prev	data_in	tuple_since_hist	hist_sat
0	3	{P(a), Q(b)}	$\bot$	$\bot$	[{(3, {a}, {b})}]	[]	{}	{}
1	4	{Q(a), Q(b)}	0	0	[{(4, {}, {a, b})}]	[{(3, {b})}]	{b ↦ 0}	{b}
2	5	{Q(a), Q(b)}	0	1	[{(5, {}, {a, b})}]	[{(4, {a, b})}, {(3, {b})}]	{b ↦ 0, a ↦ 1}	{b}
3	6	{Q(a)}	1	2	[{(6, {}, {a})}]	[{(5, {a, b})}, {(4, {a, b})}]	{b ↦ 0, a ↦ 1}	{a, b}
4	7	{}	2	3	[{(7, {}, {})}]	[{(6, {a})}, {(5, {a, b})}]	{a ↦ 1}	{a}

Figure 4.2: An example of the contents of `tuple_since_sat` and `hist_sat` for the formula $P(x)\ T_{[1,3)}\ Q(x)$ with the unary atomic predicates P and Q and tuples a, b. The abbreviation ts stands for timestamp and tp for time point. Note that `tuple_since_sat` points to time points, not timestamps like `tupe_in_once`.

3 is not within the interval anymore and is thus removed from `data_in`.

The last not covered subformula of the disjunction in the general case of definition 3.9 is $\blacklozenge_{[0,a)} \left(\bullet_{[0,\infty)} \left(\psi \ S_{[0,\infty)} \ (\psi \wedge \varphi) \right) \right)$. As described in section 3.2.1, the goal of this subformula is to require the formula $\left(\psi \ S_{[0,\infty)} \ (\psi \wedge \varphi) \right)$ to hold at the newest time point inside the interval. To compute the tuples satisfying this condition, the state additionally includes the set `since_sat` which simply contains all tuples satisfying this subformula. This is enough because the interval is $[0, \infty)$, otherwise a more complex data-structure like `msaux` has to be used [11]. Figure 4.3 shows an example of how this set evolves over time for a given trace. At time point 1 where the entry $(3, \{a\}, \{a\})$ is moved to `data_in`, the tuple a is added to `since_sat` because the tuple is in the intersection of the two sets. At time points 2 and 3, the tuple stays in `since_sat` because a is at both time points in the set associated with the right-hand side of the Trigger operator of the entries moved to `data_in`. Moreover at time point 3 the tuple b is part of both sets contained in the entry moved to `data_in` resulting in b being added to `since_sat` as well. At time point 4, the tuple a is removed because it is not element of the set corresponding to the right-hand side of the Trigger operator that is contained in the entry moved from `data_prev` to `data_in`.

There are some other properties stored in the `mtaux` state for keeping track of the time points and optimizing the performance. Most of these details are omitted in this thesis but some remarks are made in Chapter 5. For the sake of performance `data_prev` and `data_in` are implemented as queues but in this thesis we refer to them as lists. The initialization of the data-structure is straight-forward: All sets, mappings and lists are initialized empty.

tp	ts	database	`data_prev`	`data_in`	`since_sat`
0	3	{P(a), Q(a)}	[{(3, {a}, {a})}]	[]	{}
1	4	{P(b), Q(a)}	[{(4, {b}, {a})}]	[{(3, {a})}]	{a}
2	5	{P(b), Q(a), Q(b)}	[{(5, {b}, {a, b})}]	[{(4, {a})}, {(3, {a})}]	{a}
3	6	{Q(b)}	[{(6, {}, {b})}]	[{(5, {a, b})}, {(4, {a})}]	{a, b}
4	7	{}	[{(7, {}, {})}]	[{(6, {b})}, {(5, {a, b})}]	{b}

Figure 4.3: An example of the contents of `since_sat` for the formula $P(x) \ T_{[1,3)} \ Q(x)$ with the unary atomic predicates P and Q and tuples a, b. The abbreviation ts stands for timestamp and tp for time point.

The invariant `valid_mtaux` describes the just given properties of the different parts of the data-structure formally and additionally establishes a connection to the trivial data-structure. The formal details are omitted.

4.3 Obtaining Results

From the description of the state given in Section 4.2, it follows that the set of tuples satisfying the Trigger operator $\varphi\, T_I\, \psi$ can be obtained by computing the union of the keys of `tuple_in_once`, the set `hist_sat` and the set `since_sat`. Given the safety conditions described in Definition 3.13 under which this locale is used, the set of free variables of the Trigger operator is always exactly the set of free variables of ψ. In case there are no time points within the interval, it is possible to simply return the table representing *True* (denoted by $\{()\}$) as described in the beginning of Chapter 4. Hence the function `result_mtaux` is defined as follows.

```
if (is_empty data_in) then
  ({}, {()})
else
  (free(ψ), Mapping.keys tuple_in_once ∪ hist_sat ∪ since_sat)
```

where $\{()\}$ stands for the table representing *True*.

4.4 Updating State

Most of the work is done during updates to the data-structure. All mappings and sets have to be updated in an invariant-preserving way. The update functionality can be split into two parts: One for updating the data-structure with respect to a new largest timestamp `nt` and one for adding the new entry `(nt, l, r)`, in particular the tables `l` and `r` to the data-structure.

First the former part is described as the second part then makes use of similar concepts. As a first step, the list `data_prev` is partitioned into two lists: one denoted by `data_prev'` consisting of the elements with a timestamp difference with respect to the new timestamp `nt` that is still smaller than the lower bound of the interval (i.e. less than of a) and the rest denoted by `move`. The list `move` now contains exactly all entries where the timestamp differences to the second largest timestamp is less than a but the difference to the new timestamp `nt` is not. Moreover the mapping `tuple_in_once` is updated by removing all tuples mapping to timestamps that belong to the just removed entries just like `tuple_in` in `msaux`. In fact, the code for this part is shared between `msaux` and `mtaux`. The list `data_prev'` is exactly the one replacing `data_prev` in the updated state.

Next, the mapping `tuple_since_hist` and the sets `hist_sat` and `since_sat` are updated. The data relevant for this update are the entries of the list `move`. The list is iterated in the order the time points arrived in the temporal structure and for every entry `(t, l, r)` in the list, the following updates are performed:

1. The keys of the mapping `tuple_since_hist` are filtered to only include tuples that are inside the table `r`, i.e. the table containing all tuples satisfying ψ. As mentioned before, `tuple_since_hist` points to the smallest time point since when ψ was continuously satisfied. Hence if ψ is not satisfied at the newest time point inside the interval, the mapping should no longer point to a time point.
2. The mapping `tuple_since_hist` is updated for all tuples ($r \setminus$ `Mapping.keys tuple_since_hist`) to point to `t`. This makes sure that if some tuple satisfies ψ at the newest point inside the interval, the mapping `tuple_since_hist` keeps track of this. If the mapping already contained an entry for a given tuple in `r`, the mapping must not change for said tuple as the chain of time points where ψ is satisfied already started earlier for said tuple.
3. The set of tuples satisfying Historically inside the interval, i.e. the tuples contained in the set `hist_sat`, must of course satisfy ψ at the time point corresponding to `(t, l, r)` as well. Hence the set `hist_sat` is intersected with r: `hist_sat` $\cap\, r$. Note that this only filters out tuples not satisfying Historically anymore, new tuples satisfying Historically are added later.
4. Analogously to the set `hist_sat`, `since_sat` is also intersected with `r`. In addition, all tuples being part of both tables, `l` and `r`, are added to the set as they now satisfy the subformula $\Big(\psi \;\; S_{[0,\infty)} \;\; (\psi \wedge \varphi)\Big)$. The whole update is expressed by $(\texttt{hist\_sat} \cap r) \cup (l \bowtie r)$ where $\bowtie$ denotes a join of the two tables. If the left-hand side φ is a negation of some formula φ', l contains the tuples satisfying φ' and an anti-join is performed. An anti-join between l and r is well defined because the safety conditions given in Definition 3.13 guarantee the set of free variables of φ to be a subset of the ones of ψ.

The resulting mapping and sets are called `tuple_since_hist'`, `hist_sat'` and `since_sat'`, respectively.

Next, the list `move` is now filtered to only contain entries that are within the interval. This is necessary as it is possible that after some entry was added to `data_prev`, no new time points occur for a long time. The time difference to the next timestamp then might already be larger then the upper bound of the interval and thus these entries will never enter `data_in`. The resulting

list is called `move'`. The steps performed before must operate on `move` rather than `move'` because the following scenario is possible: Assume there is a tuple a in `since_sat` meaning the tuple satisfies $\left(\psi\ S_{[0,\infty)}\ (\psi \wedge \varphi)\right)$ at the newest time point that is inside interval. When updating the timestamp there might be an entry for time point i that is no longer in `data_prev` and has a timestamp old enough to directly skip the interval (and thus `data_in` and `move'`). Assuming tuple a does not satisfy ψ at said time point i, using `move'` instead of `move` will lead to an issue: In that case the set `since_sat` is never intersected with the entry skipping `data_in` and thus after the update the tuple a is still within the set `since_sat` even though it might no longer satisfy the formula $\left(\psi\ S_{[0,\infty)}\ (\psi \wedge \varphi)\right)$ at the newest time point inside the interval. Hence the invariant would be violated. Using `move` instead of `move'` for the updates described before is one way to circumvent this issue.

Afterwards, the list `data_in` is partitioned into two additional lists: one denoted by `data_in'` containing the elements with a timestamp whose difference to `nt` is still inside the interval I and the rest denoted by `drop`. The list `drop` consists exactly of the entries previously being inside the interval that now have to be dropped because the difference to the new timestamp `nt` is no longer within the interval I.

After partitioning `data_in`, the list `move'` can be appended to `data_in'`. The order of operations could also be reversed here: One could first append `move` to `data_in` and then partition the resulting list but as this will result in an additional append operation for entries that are immediately dropped again, the former order was preferred. The resulting list `data_in''` will replace `data_in` in the updated state.

Finally, the set `hist_sat'` is updated one last time. As it is possible that Historically previously was not satisfied but now all time points where the right-hand side was not satisfied are no longer in the interval, it is required that these tuples are added. This set of tuples exactly corresponds to the set of tuples for which `tuple_since_hist` now points to a time point less or equal than the smallest time point inside `data_in`. Because this set of tuples is a subset of all tables inside `data_in`, it suffices to iterate over the table of the oldest entry in `data_in`, lookup their corresponding value in `tuple_since_hist` and check whether this time point is at most the oldest time point whose timestamp is still in the interval.

In the special case where the list `data_in''` is empty, i.e. there are no time points within the interval, the mapping `tuple_since_hist'` as well as the set `hist_sat'` are simply emptied.

Let `tuple_since_hist''` and `hist_sat''` denote the mapping and set where (if necessary) the updates described in the last two paragraphs were applied.

This is all that is needed to update the data-structure with respect to a new timestamp `nt` in an invariant-preserving way. The new state consists of `data_prev'`, `data_in''`, `tuple_in_once'`, `tuple_since_hist''`, `hist_sat''` and `since_sat'`.

Adding the two tables `l` and `r` with timestamp `nt` from the entry to the state is now straight-forward: First it is checked whether zero is part of the interval I. If it is, the new entry is appended to `data_in` and the mapping `tuple_since_hist` as well as the two sets `hist_sat` and `since_sat` are updated analogous to the way it is done for the entries moved from `data_prev` to `data_in`. If zero is not part of the interval, the new entry is appended to `data_prev` and the tuples in `r` are mapped to `nt` in the mapping `tuple_in_once`.

In the formalization, this whole procedure is proven to satisfy the described assumption on the update in the locale `mtaux`.

4.5 Progress

The described algorithm computes the set of satisfying assignments to the free variables of $\varphi\ T_I\ \psi$ for some formulas φ, ψ and an interval I at all time points where the set of satisfying assignments to the free variables of φ and ψ can be computed. Thus using this algorithm, the progress of $\varphi\ T_I\ \psi$ is equal to the minimum progress of φ and ψ which is the same amount of progress the Since operator achieves given the same two subformulas.

The translated formulas derived in Chapter 3 are not always capable of making the same amount of progress, especially as the case of a general interval for Historically includes an Eventually operator reaching into the future. In the formalization it is proven that the specialized algorithm makes at least as much progress as the translated formulas. The following lemma states this more formally:

Lemma 4.1 Let $\mathcal{M}$ denote the monitoring algorithm VeriMon that evaluates Trigger operators using the specialized algorithm described in this chapter. Then the following inequality holds:

$$\forall \varphi\ \psi\ I\ i. prog^{\mathcal{M}}(\varphi\ T_I'\ \psi, i) \leq prog^{\mathcal{M}}(\varphi\ T_I\ \psi, i)$$

This lemma follows by plugging in the progress values for all operators and simplifying the resulting expression.

4.6 Optimizations

Compared to the described algorithm, the formalization additionally includes optimizations to make the exported code run faster. Some of them are

highlighted in this section.

A simple optimization is to empty the set `since_sat'` in addition to the mapping `tuple_since_hist'` and the set `hist_sat'` if the list `data_in` is empty. This is not required because `since_sat'` is responsible for keeping track of the tuples satisfying the formula $\left(\psi\ S_{[0,\infty)}\ (\psi \wedge \varphi)\right)$ which is over the unbounded interval $[0, \infty)$. Nevertheless it is possible because of the following reason: Assume there is a tuple a that previously was element of `since_sat'` but was removed from it just because `data_in` is empty. The next time an entry is added to `data_in` there are two possibilities: Either a is element of the table containing the tuples satisfying the right hand side ψ or not. If it is, then the tuple a also satisfies Historically and is thus part of the results anyway. If a is not part of this table, then it is removed from the set `since_sat'` anyway because of the update steps previously described. For additional entries this property holds inductively.

Another optimization is to include an additional set in the state that is exactly equal to `Mapping.keys tuple_in_once`. All updates performed on the mapping are analogously performed on the set. This increases the time required for an update by a constant factor but removes the necessity to compute the mapping's keys in `result_mtaux`. Otherwise the function `Mapping.keys` would iterate over the whole mapping `tuple_in_once` in order to create the set and `tuple_in_once` contains *every* tuple satisfying the left-hand side in the interval $[0, a)$ where a is the lower bound of the interval Trigger is evaluated over.

In contrast to these implemented optimizations there are also places where optimizations are possible but not yet implemented. One missed opportunity for optimization is for updating `hist_sat'`. Recall that it suffices to look at the tuples in the oldest entry in the list `data_in` in order to find the new tuples satisfying Historically. This iteration over the table would not be required in cases where the oldest entry in `data_in` does not change. This could improve the runtime quite a bit if the tables are large and there are many time points with the same timestamp.

Moreover it should be noted that for simplicity the operators Trigger (and Release) are currently not supported in multiway-joins unless the interval contains zero. There are planned extensions to VeriMon that will reduce the required effort to support `mtaux` in multiway-joins significantly.

Chapter 5

Performance Evaluation

In order to assess the performance of the specialized algorithm its running time is compared to the running time of the translated formulas. The parameters l and n determining the interval length and the number of tuples in the result are used to generate the inputs to the monitoring algorithm. With respect to these two parameters, the asymptotic behaviours of the specialized algorithm and the translated formulas are analyzed. First some implementation details related to the experiments are mentioned and then the different experiments are described. Afterwards the results of the experiments are presented and discussed.

5.1 Implementation

Some unverified parts of VeriMon are extended in order to support the evaluation of the operators Trigger and Release. Namely the lexer and parser are extended to include the two new operators. Moreover a new flag is added to the unverified part that makes the monitor translate the Trigger operator to the formulas derived in Chapter 3. This allows to compare the running time of the specialized algorithm to the translated formulas. The translation is done just before conjunctions are rewritten to multiway conjunctions and is formally verified. Currently only top-level Trigger operators and Trigger operators under a top-level conjunction are translated but it is relatively straightforward to extend the translation to proceed recursively. Furthermore M. Raszyk noted that in the development version of Isabelle / HOL new code equations for faster set operations are in place and hence we used the development version in order to extract the verified code.

Moreover we decided not to measure the running time of the whole monitoring algorithm but only the time spent in the function `meval` responsible for evaluating the formula. This way reading the input files, setting up the monitor and writing the results to the console is excluded from the measured

time. In order to accomplish this, the generated OCaml code is manually modified. The statement `let t = Unix.gettimeofday ()` for some new variable `t` is added before and after the part of the program we want to measure. The function `Unix.gettimeofday` returns the current time in unix time with "resolution better than 1 second" [7].

5.2 Description of the Experiments

The performance evaluation is performed for different settings resulting in multiple experiments. The varied parameters include the interval, the number of free variables of the left-hand side of the formula and the pattern of the input trace. The left-hand side is fixed to either be an atomic predicate (the simplest type of formula dependent on the trace) with either one or two free variables and possibly negated ($A(x), \neg A(x), A(x,y), \neg A(x,y)$ for the free variables x and y) or the closed formula *False* whereas the right-hand side is always fixed to the binary predicate $B(x,y)$ with the free variables x and y. For the interval the following cases are considered: $[0,\infty)$, $[0,b)$, $[a,\infty)$ and $[a,b)$ for some natural numbers a, b. The values of a and b are set so that $b-a$ is proportional to the parameter l. Note that not all combinations are allowed by the safety conditions defined in Definition 3.13. For example if the interval does not include zero the set of free variables of the left-hand side must be the same as the set of free variables of the right-hand side and a negated left-hand side is not allowed. Thus only the left-hand side $A(x,y)$ is used if zero is not in the interval. Moreover for intervals not including zero, the updated safety conditions presented in Section 3.3 require the evaluation under a conjunction. Thus instead of evaluating top-level Trigger operators, the operators are evaluated under the conjunction with the atomic predicate $C(x,y)$ with the same set of free variables. The atomic predicate C is never included in the input trace. This way, the algorithm does not have to output any results and the experiments are able to finish earlier. Note that the evaluation of the operator Trigger is not affected by this. Furthermore the join with an empty table for $C(x,y)$ is performed in constant time, independent of the size of the other table. Hence the computation of the join does not distort the performance evaluation.

In addition to a formula, the monitoring algorithm also requires a trace as input. The length of the trace is also set dependent on l in a way that guarantees that the whole interval fits in the generated trace. Inspired by the derived formulas semantically equivalent to Trigger and Release (Definitions 3.9 and 3.11), the traces are generated in a way that guarantees the satisfaction of one of the three disjuncts in the translated formula by a certain amount of tuples. The number of distinct tuples is with high probability at least equal to n because we uniformly draw n random integer assignments to the free variables x and y from the interval $[0, 10^9]$ where $n \leq 800$ holds for

all experiments performed and thus $n^2 \ll |[0, 10^9]| = 10^9 + 1$. In addition, at every time point, n additional random assignments to the free variables of the left- and right-hand side are added to make the trace less uniform. These random tuples can also increase the total number tuples in the result but as they are completely random it is unlikely that any of the disjuncts in Definition 3.9 besides Once is satisfied by chance. But in the case of the Once disjunct, the result is actually much bigger: All time points before the interval contain n additional random assignments making the expected number of tuples satisfying the Once disjunct to be in the order of $\Omega(l \cdot n)$. Since the interval length is proportional to l, there are $\Omega(l)$ such time points.

For Trigger the three patterns Historically, Since and Once are generated and for Release analogously the three patterns Always, Until and Eventually. These cases correspond to the three subformulas in case of a bounded interval not including zero in Definitions 3.9 and 3.11.

5.3 Results and Discussion

The experiments are executed on a MacBook Pro (Retina, 15-inch, Early 2013) running macOS Catalina (Version 10.15.7, build 19H1217) on a 2.7 GHz Quad-Core Intel Core i7 CPU with access to 16 GB 1600 MHz DDR3 RAM. All of the experiments are repeated ten times in order to reduce the impact of the execution environment. The input files for the experiments, the scripts to evaluate them and the resulting CSV files can be found on GitHub [2, 5].

First the running times of the monitor evaluating the translated formulas for Trigger are compared to the running times of the specialized algorithm. For this comparison all safe combinations of left-hand sides, intervals, and trace patterns are evaluated using the translated formulas and also using the specialized algorithm. The results of the experiments in the different settings are shown in Table 5.1. The evaluated formulas are listed in the column *Formula* whereas the column *Pattern* lists the underlying pattern used in order to generate the trace. The columns *Translated Formula* and *Specialized Algorithm* show the mean running time of VeriMon over the ten executions in seconds when monitoring the respective formula on a trace with the pattern listed in the same row. The standard deviations can be computed from the raw data linked above using a provided script but they are omitted in the table as they are negligible.

As noted in the caption of Table 5.1 it is important that neither l nor n stay the same across all experiments. Thus the running times of the different experiments are not comparable. But for every individual experiment, i.e. a row, the running time using the translated formulas can be compared to the running time of the specialized algorithm as they both originate from the exact same input trace. The last column `mtaux` shows the portion of the

Formula	Trace Pattern	Translated Formula	Specialized Algorithm	`mtaux`
$False\ T_{[0,\infty)}\ B(x,y)$	historically	2.91	1.95	0.87
	since	1.95	1.24	0.45
$\neg A(x)\ T_{[0,\infty)}\ B(x,y)$	historically	6.22	2.85	1.07
	since	3.89	1.94	0.63
$A(x)\ T_{[0,\infty)}\ B(x,y)$	historically	6.81	2.86	1.03
	since	5.36	2.05	0.72
$\neg A(x,y)\ T_{[0,\infty)}\ B(x,y)$	historically	6.27	2.93	1.09
	since	4.05	2.05	0.65
$A(x,y)\ T_{[0,\infty)}\ B(x,y)$	historically	6.44	2.88	1.02
	since	5.33	2.11	0.72
$False\ T_{[0,b)}\ B(x,y)$	historically	5.54	1.9	0.78
	since	3.91	1.24	0.47
$\neg A(x)\ T_{[0,b)}\ B(x,y)$	historically	8.6	2.87	1.03
	since	5.73	1.99	0.65
$A(x)\ T_{[0,b)}\ B(x,y)$	historically	9.12	2.7	0.9
	since	7.68	2.07	0.73
$\neg A(x,y)\ T_{[0,b)}\ B(x,y)$	historically	8.33	2.83	0.99
	since	5.63	2.01	0.65
$A(x,y)\ T_{[0,b)}\ B(x,y)$	historically	9.23	2.76	0.92
	since	7.83	2.14	0.74
$A(x,y)\ T_{[a,\infty)}\ B(x,y)$	historically	22.8	1.09	0.62
	since	22.81	1.05	0.63
	once	22.54	0.99	0.62
$A(x,y)\ T_{[a,b)}\ B(x,y)$	historically	77.14	0.38	0.23
	since	75.68	0.34	0.21
	once	73.93	0.32	0.2

Table 5.1: Mean running times of VeriMon in seconds when monitoring the formula $\varphi\ T_I\ B(x,y)$ with different left-hand sides on traces with different patterns. Note that neither the length of the trace nor the number of tuples in the output is the same across all experiments / rows.

running time using the specialized algorithm that is spent in the functions interpreting the locale `mtaux`, i.e. in the evaluation of the Trigger operator. In particular this excludes the evaluation of the atomic predicates.

Table 5.1 shows that the specialized algorithm is able to compute the set of satisfying assignments to the free variables more efficiently than the translated formulas. The difference is much more pronounced for the experiments where zero is not part of the interval (last six rows) and especially large if we have a bounded interval not including zero (last three rows). This is not an unexpected result as the translated formulas for the various types of intervals are different. We remark that the translated formula for Historically for bounded intervals not including zero in Definition 3.3 includes a subformula that is just added in order to make the formula safe but is most-likely the reason its evaluation takes much longer. Note that the translated formula in this case collects a lot of tuples in order to (over-)approximate the result. Even though this over-approximation is then refined to obtain the actual result, the intermediate tables still have to be computed. The complexity is amplified by the data distribution chosen for the experiments: The additional random entries added to the time points lead to a large amount of tuples satisfying Once and thus further increase the size of this over-approximation. A further reason is mentioned later when discussing the asymptotic behaviour.

Next the results of the experiments assessing the specialized algorithm's asymptotic behaviour are presented. In order to measure the asymptotic behaviour with respect to the parameters l and n the values $2l, 4l, 8l, 2n, 4n, 8n$ and their combinations are analyzed. They represent an increase of the parameters l and n by the respective factors. All numbers are rounded to show two decimal places. The column 'time `meval`' contains the mean running times of the function `meval` in seconds whereas the column 'time `mtaux`' contains the mean running time (in seconds) that is spent in the locale `mtaux`. The columns 'ratio' show the factor the mean running times 'time `meval`' and 'time `mtaux`' have increased relative to the previous row. As before we omit the standard deviations as they are negligible (The provided scripts can calculate them). Since the parameters are doubled between consecutive rows, we expect the 'ratio' columns to contain values close to 2 if the running time increases linearly in the respective parameter, values close to 4 if it increases quadratically, etc. All tables resulting from the asymptotic experiments can be found on GitHub [2] and only a representative selection is presented in this thesis. Note that the values of the parameters l and n are not always the same for the baseline and they are show in the captions of the tables. In the case of the specialized algorithm, the results are very homogenous and just the two sample Tables 5.2 and 5.3 are listed.

parameters	time `meval`	ratio	time `mtaux`	ratio
l	0.08		0.03	
$2l$	0.17	2.09	0.07	2.27
$4l$	0.35	2.09	0.15	2.04
$8l$	0.72	2.05	0.31	2.08
n	0.08		0.03	
$2n$	0.19	2.35	0.06	1.76
$4n$	0.37	1.96	0.12	2.17
$8n$	0.73	1.97	0.30	2.43
n, l	0.08		0.03	
$2n, 2l$	0.40	4.95	0.12	3.54
$4n, 4l$	1.57	3.95	0.51	4.47
$8n, 8l$	6.05	3.85	2.49	4.84

Table 5.2:

$n = 100$, $l = 100$
Formula: $False\ T_{[0,b)}\ B(x, y)$
Pattern: Historically

parameters	time `meval`	ratio	time `mtaux`	ratio
l	0.16		0.09	
$2l$	0.35	2.25	0.20	2.31
$4l$	0.85	2.40	0.50	2.53
$8l$	1.99	2.35	1.24	2.46
n	0.16		0.09	
$2n$	0.37	2.37	0.20	2.31
$4n$	0.84	2.25	0.54	2.73
$8n$	1.92	2.28	1.36	2.50
n, l	0.16		0.09	
$2n, 2l$	0.88	5.57	0.49	5.73
$4n, 4l$	4.41	5.04	3.15	6.38
$8n, 8l$	22.65	5.14	18.01	5.72

Table 5.3:

$n = 100$, $l = 100$
Formula: $A(x, y)\ T_{[a,\infty)}\ B(x, y)$
Pattern: Since

In both tables, the columns 'ratio' for increasing l and n, respectively, contain values around 2 meaning the running time increases by about a factor two when doubling the respective parameter. The same is the case for all other asymptotic experiments conducted with the specialized algorithm. Hence the experiments suggest a linear asymptotic behaviour in both l and n when keeping the other parameter fixed. In contrast, when the parameters l and n are increased together, the running times increases superlinearly. The values in the columns 'ratio' are in these cases around the value 4 which indicates a quadratic asymptotic behaviour. These results suggest that the specialized algorithm's runtime complexity with respect to l and n is not in $\mathcal{O}(n + l)$ but rather in $\Omega(n \cdot l)$. Due to the set operations used, there are definitely additional logarithmic terms in the expression but no detailed analysis of the asymptotic complexity is performed. The just described trends are consistent over all conducted experiments. It was to expect that the specialized algorithm is linear in $n \cdot l$ as this corresponds to the expected input size. The trace contains at every time point in expectation $\Theta(n)$ tuples and the described algorithm processes the whole input at some point. This is comparable to the implementations for the operators Since and Until which are also both linear in the input size.

One can also notice that the fraction $\frac{\text{time mtaux}}{\text{time}}$, i.e. the portion of the running time that is spent in the locale *mtaux* is larger if the interval does not include zero. A possible explanation is that the result is a larger set due to the Once disjunct in the translated formula. Due to the data distribution chosen for the experiments, there can be many distinct tuples appearing before the interval that all have to be collected. If the interval contains zero, the Once disjunct is not present in the semantically equivalent formula and hence less operations have to be performed.

The asymptotic behaviour of the translated formulas is in some cases similar to that of the specialized algorithm, in others the translated formulas yield worse performance. In order to assess the asymptotic behaviour of the translated formulas the same method was used as for the specialized algorithm. Note that the input traces are not the same as the ones used for the specialized algorithm because the absolute running times in some cases were substantially larger and we wanted the experiments to terminate in a reasonable amount of time. Hence the absolute running times of the two sets of experiments cannot be compared but it is of course still possible to assess the asymptotic behaviour. The meaning of the columns is the same as before. For intervals including zero, the values in column 'ratio' are close to 2 and thus the results suggest a linear runtime complexity in both l and n. Analogously to the specialized algorithm, the rows where both l and n are increased at the same time suggest, that the running time is a term in $\Omega(n \cdot l)$ rather than $\mathcal{O}(n + l)$. The two sample Tables 5.4 and 5.5 for intervals including zero are shown in this thesis.

parameters	time `meval`	ratio
l	0.24	
$2l$	0.50	2.04
$4l$	1.03	2.08
$8l$	2.07	2.01
n	0.24	
$2n$	0.59	2.42
$4n$	1.32	2.25
$8n$	2.85	2.16
n, l	0.24	
$2n, 2l$	1.20	4.95
$4n, 4l$	5.40	4.49
$8n, 8l$	23.10	4.28

Table 5.4:

$n = 100$, $l = 100$
Formula: $False\ T_{[0,b)}\ B(x, y)$
Pattern: Historically

parameters	time `meval`	ratio
l	0.22	
$2l$	0.47	2.10
$4l$	0.98	2.10
$8l$	2.02	2.06
n	0.22	
$2n$	0.53	2.36
$4n$	1.17	2.23
$8n$	2.64	2.25
n, l	0.22	
$2n, 2l$	1.07	4.82
$4n, 4l$	4.80	4.47
$8n, 8l$	21.25	4.43

Table 5.5:

$n = 100$, $l = 100$
Formula: $A(x, y)\ T_{[0,\infty)}\ B(x, y)$
Pattern: Since

Again it was to expect that the running time is linear in $l \cdot n$ (i.e. the running time increases quadratically when doubling both) for the same reason as in the case of the specialized algorithm: The evaluation of the operator Since is linear in the input size and the trace contains $\Omega(l \cdot n)$ tuples in expectation.

The results obtained when evaluating the translated formulas on intervals including zero differ quite a bit from what can be observed when doubling the parameter l for intervals excluding zero. Then the values in the column 'ratio' are significantly larger than 2 and seem to indicate more of a quadratic increase than a linear one. Moreover the running time definitely seems to grow faster than quadratic when both l and n are doubled. This seems reasonable as now the evaluation is not even linear in l anymore. In contrast to these superlinear asymptotic behaviours, the numbers still make a linear asymptotic behaviour in n seem plausible. The sample Tables 5.6 and 5.7 for intervals excluding 0 are listed in this thesis. The worse asymptotic behaviour in l is not unexpected. The implementation of the operator Since is linear in its output size which in the case of an interval excluding zero is expected to be in the order of $\Omega(n \cdot l)$ for at least $\Omega(l)$ time points. The reason for this is the effect of the chosen data distribution on the Once disjunct in the semantically equivalent formula (see Definition 3.9). The interval size is proportional to l and all tuples satisfying the right-hand side before the interval must be outputted by the Since operator at every time point. Computing an output of the size $\Omega(n \cdot l)$ at $\Omega(l)$ time points results in an asymptotic running time

complexity in $\Omega(n \cdot l^2)$ which can be observed by looking at the column 'ratio' in Table 5.6 which contains values around 4 indicating a quadratic increase in running time. In Table 5.7 the values in the column 'ratio' also indicate a superlinear increase reaching the described quadratic increase in the last row. For intervals including zero, this is not observed as the size of the result in this case is in expectation in the order $\mathcal{O}(n)$ because the Once disjunct is not part of the translated formulas. On the other hand, in the specialized algorithm the set of tuples satisfying the Once disjunct is not computed from scratch at every time point due to the optimized way the keys of a mapping are computed (see Section 4.6) and hence does not show a quadratic increase in running time. It is important to note that if the output was printed to a console or written to a file and that time was measured as well, it is to expect that the specialized algorithm will show a similar asymptotic behaviour.

parameters	time `meval`	ratio
l	0.89	
$2l$	3.80	4.26
$4l$	17.27	4.54
$8l$	73.00	4.23
n	0.89	
$2n$	1.97	2.20
$4n$	4.37	2.22
$8n$	9.66	2.21
n, l	0.89	
$2n, 2l$	8.56	9.58
$4n, 4l$	81.49	9.52
$8n, 8l$	762.28	9.35

Table 5.6:

$n = 40$, $l = 40$
Formula: $A(x, y)\ T_{[a,b)}\ B(x, y)$
Pattern: Once

parameters	time `meval`	ratio
l	0.07	
$2l$	0.19	2.72
$4l$	0.60	3.10
$8l$	2.49	4.15
n	0.07	
$2n$	0.15	2.18
$4n$	0.41	2.65
$8n$	0.97	2.37
n, l	0.07	
$2n, 2l$	0.47	6.59
$4n, 4l$	4.06	8.70
$8n, 8l$	35.28	8.68

Table 5.7:

$n = 40$, $l = 40$
Formula: $A(x, y)\ T_{[a,\infty)}\ B(x, y)$
Pattern: Since

Similar observations can be made when looking at the running times of the translated formulas for Release. The results of two experiments are shown in Tables 5.8 and 5.9. The remaining Tables can once again be found on GitHub [2]. The difference in whether the interval includes zero or not is once again clearly visible in the asymptotic behaviour in l because the evaluation of the Until operator is linear in the output size as well and the same reasoning about the result size of the Once disjunct symmetrically applies to the Eventually disjunct in the translated formulas for Release.

parameters	time `meval`	ratio
l	0.39	
$2l$	0.81	2.08
$4l$	1.75	2.15
$8l$	3.62	2.06
n	0.39	
$2n$	0.94	2.40
$4n$	2.13	2.26
$8n$	4.72	2.22
n, l	0.39	
$2n, 2l$	1.96	5.00
$4n, 4l$	8.78	4.49
$8n, 8l$	39.79	4.53

Table 5.8:

$n = 100$, $l = 100$

Formula: $A(x)\ R_{[0,b)}\ B(x,y)$

Pattern: Until

parameters	time `meval`	ratio
l	1.09	
$2l$	4.67	4.30
$4l$	20.39	4.37
$8l$	89.66	4.40
n	1.09	
$2n$	2.40	2.21
$4n$	5.22	2.18
$8n$	11.57	2.22
n, l	1.09	
$2n, 2l$	10.27	9.45
$4n, 4l$	97.52	9.49
$8n, 8l$	941.59	9.66

Table 5.9:

$n = 40$, $l = 40$

Formula: $A(x,y)\ R_{[a,b)}\ B(x,y)$

Pattern: Eventually

Chapter 6

Conclusion

The approach of deriving safe formulas that are semantically equivalent to the unsupported cases for the operators Trigger and Release gives good insight on their safety conditions. In the case of Trigger it was possible to remove the redundant and inefficient parts of the semantically equivalent formulas when turning their evaluation into a specialized monitoring algorithm. It seems likely that a similar approach could work for the operator Release as well but due to the time constraints given for this thesis, this hypothesis has not been tested. Moreover the semantically equivalent formulas turned out to be a good reference for assessing the specialized algorithm's performance. The results of the performance evaluation were clearly in favour of the specialized algorithm, especially in the cases of bounded intervals not including zero. The experiments for assessing the asymptotic behaviour with respect to the interval length (that is linked to the trace length) suggest that having the zero in the interval is the deciding factor for achieving linear runtime complexity with respect to the interval length in case of the translated formulas. A different result was obtained for the specialized algorithm: The results from the experiments assessing its asymptotic behaviour with respect to the interval length always suggest a linear runtime complexity, independent of whether zero is part of the interval. Based on the performed experiments, the asymptotic behaviour with respect to the parameter n determining the number of tuples in the result and the random noise in the trace seems to be linear for both the translated formulas and the specialized algorithm, no matter whether zero is in the interval or not. Moreover the translated formulas and the specialized algorithm both show a non-linear increase in their running time when the interval length and n are increased together.

The translated formulas for Trigger seem to perform pretty well for intervals including zero, especially as the asymptotic behaviour in the performed experiments seems to be linear when doubling just one of the parameters. Nevertheless, the specialized algorithm is able to undercut their running

time in all performed comparison experiments and also shows a linear runtime complexity when doubling just one of the parameters. Moreover the specialized algorithm is in some cases able to make more *progress* (see Definition 2.8) than the translated formulas meaning it can output some results earlier. For these reasons the formalization in VeriMon uses the specialized algorithm in order to evaluate the operator Trigger rather than rewriting it to the respective semantically equivalent formula as it is done for Release. Based on the performed experiments, the presented formulas semantically equivalent to Release have similar asymptotic properties as the ones semantically equivalent to Trigger: The linear runtime complexity with respect to the interval size is only achieved in experiments evaluating Release over an interval that includes zero.

The current formalization could be further extended to evaluate formulas that are currently not safe. As mentioned in Section 3.3, the safety conditions can be relaxed to allow certain additional cases such as formulas satisfying `safe_assignment` or `is_constraint` occurring in temporal operators. Moreover the safety conditions derived in Section 3.3 require the left-hand side of the Trigger operator to have the same set of free variables as the right-hand side if zero is not in the interval. This property is directly derived from the safety conditions of the translated formula but it could be loosened if the left-hand side corresponds to a `False` and the Trigger operator thus to a Historically. This is an operator of interest and thus a very likely future extension to the specialized algorithm described in this thesis. Only a small set of changes should be required, the major difficulty will probably be the detection of the subformula `False` which currently is not part of the formula data type and hence would require deep pattern matching as in the case of *True* described in Section 3.4. An easier approach might be to include Historically in the formula data type. Moreover the performance of the specialized algorithm described can be further improved if for example the last time point in the interval does not change for a long period of time (see Section 4.6).

In conclusion the goal of being able to safely evaluate the operator Trigger under less strict safety conditions has been achieved, even though not exactly the way it was imagined first. Originally the idea was to abstract the algorithm for evaluating Since [11] in a locale such that this locale could be used in order to instantiate monitoring algorithms for the two operators Since and Trigger. During the work on this thesis we decided to limit the amount of shared functionality due to the differences between the two operators that were larger than initially anticipated.

Bibliography

[1] Code generation from isabelle/hol theories. https://isabelle.in.tum.de/dist/Isabelle2021/doc/codegen.pdf. Accessed: 09.07.2021.

[2] Copy of described monpoly version, list of all tables and an explanation how the obtained results can be verified. https://github.com/Tyratox/safe-evaluation-of-mfotl-dual-temporal-operators.

[3] Coq proof assistent. https://coq.inria.fr. Accessed: 01.07.2021.

[4] Early history of coq. https://coq.inria.fr/refman/history.html. Accessed: 01.07.2021.

[5] The experiments evaluated in the performance evaluation in this thesis. https://github.com/Tyratox/safe-evaluation-of-mfotl-dual-temporal-operators/releases/tag/1.0.

[6] The formalization of verimon. https://bitbucket.org/jshs/monpoly/.

[7] Ocaml documentation for `Unix.gettimeofday`. https://ocaml.org/api/Unix.html#VALgettimeofday. Accessed: 07.08.2021.

[8] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*. Addison-Wesley, 1996.

[9] Rajeev Alur and Thomas A. Henzinger. Logics and models of real time: A survey. *Real-Time: Theory in Practice Lecture Notes in Computer Science*, page 74–106, 1992.

[10] Ezio Bartocci and Falcone Yliès. *Lectures on Runtime Verification Introductory and Advanced Topics*. Springer International Publishing, 2018.

[11] David Basin, Thibault Dardinier, Lukas Heimes, Srđan Krstić, Martin Raszyk, Joshua Schneider, and Dmitriy Traytel. A formally verified, optimized monitor for metric first-order dynamic logic. *Automated Reasoning Lecture Notes in Computer Science*, page 432–453, 2020.

[12] David Basin, Felix Klaedtke, Samuel Müller, and Eugen Zălinescu. Monitoring metric first-order temporal properties. *Journal of the ACM*, 62(2):1–45, 2015.

[13] David Basin, Felix Klaedtke, and Eugen Zalinescu. The monpoly monitoring tool. In Giles Reger and Klaus Havelund, editors, *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools*, volume 3 of *Kalpa Publications in Computing*, pages 19–28. EasyChair, 2017.

[14] Adel Dokhanchi, Bardh Hoxha, and Georgios Fainekos. Metric interval temporal logic specification elicitation and debugging. *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, 2015.

[15] Herbert B. Enderton. *A mathematical introduction to logic*. Academic Press, 1997.

[16] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[17] K. Rustan M. Leino. This is boogie 2. June 2008.

[18] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[19] Pierre Letouzey. A new extraction for coq. *Lecture Notes in Computer Science Types for Proofs and Programs*, page 200–219, 2003.

[20] Peter J. Müller, Malte J. Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. *Lecture Notes in Computer Science Verification, Model Checking, and Abstract Interpretation*, page 41–62, 2015.

[21] Tobias Nipkow and Gerwin Klein. *Concrete semantics: with Isabelle /HOL*. Springer, 2015.

[22] Lawrence C. Paulson. Natural deduction as higher-order resolution. *The Journal of Logic Programming*, 3(3):237–258, 1986.

[23] Albert Rizaldi, Jonas Keinholz, Monika Huber, Jochen Feldle, Fabian Immler, Matthias Althoff, Eric Hilgendorf, and Tobias Nipkow. Formalising and monitoring traffic rules for autonomous vehicles in isabelle/hol. In Nadia Polikarpova and Steve Schneider, editors, *Integrated Formal Methods*, pages 50–66, Cham, 2017. Springer International Publishing.

[24] Joshua Schneider, David Basin, Srđan Krstić, and Dmitriy Traytel. A formally verified monitor for metric first-order temporal logic. *Runtime Verification Lecture Notes in Computer Science*, page 310–328, 2019.

[25] Prasanna Thati and Grigore Roşu. Monitoring algorithms for metric temporal logic specifications. *Electronic Notes in Theoretical Computer Science*, 113:145–162, 2005.

[26] Artem Yushkovskiy. Comparison of two theorem provers: Isabelle/hol and coq, 2018.