Mutation-based Validation of Temporal Logic Specifications with Guarantees

Eva Charlotte Mayer

Version: 1.0

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Wilhelm-Schickard-Institute for Computer Science
Department of Computer Engineering

Bachelor's thesis

Mutation-based Validation of Temporal Logic Specifications with Guarantees

Eva Charlotte Mayer

1. Reviewer	Prof. Dr. Thomas Kropf Department of Computer Engineering Eberhard Karls University Tübingen, Germany
2. Reviewer	Hendrik Röhm Corporate Research Renningen, Germany Robert Bosch GmbH
3. Reviewer	Thomas Heinz Corporate Research Renningen, Germany Robert Bosch GmbH
Supervisors	Prof. Dr. Thomas Kropf, Hendrik Röhm and Thomas Heinz

Eva Charlotte Mayer
Mutation-based Validation of Temporal Logic Specifications with Guarantees
Bachelor's thesis, Reviewers: Prof. Dr. Thomas Kropf, Hendrik Röhm and Thomas Heinz
Supervisors: Prof. Dr. Thomas Kropf, Hendrik Röhm and Thomas Heinz

Eberhard Karls University Tübingen, Germany
Department of Computer Engineering
Wilhelm-Schickard-Institute for Computer Science
Geschwister-Scholl-Platz
72074 Tübingen, Germany

Abstract

English: Trust in software relies on trust in the specification the software is based on. In this bachelor's thesis we aim to support engineers to gain trust in specifications written in Signal Temporal Logic (STL). We achieve this goal by introducing a systematic approach for the validation of STL specifications. The approach is realized in the tool STLInspector that computes validation tests which can be checked against informal textual requirements of a software. We discuss how validation tests are constructed by means of the coverage criteria property mutation, unique first cause and property inactive clause. The use of coverage criteria enables for certain guarantees about the correctness of the specification. The thesis further covers test generation algorithms that depend on SMT solving. Finally, we detail the implementation of STLInspector and apply the tool to several STL formulae to show its effectiveness in finding faulty specifications. By enabling validation of temporal logic specifications the bachelor's thesis contributes to improve the application of formal specifications in industry.

German: Vertrauen in eine Software basiert auf Vertrauen in die Spezifikation, auf der die Software aufbaut. Das Ziel dieser Bachelorarbeit ist es Vertrauen in Spezifikationen zu stärken, die in Signal-Temporaler Logik (STL) geschrieben werden. Dafür erläutern wir ein systematisches Vorgehen für die Validierung von STL Spezifikationen. Dieses Vorgehen wurde in dem Programm STLInspector umgesetzt. STLInspector generiert Validierungstests, die von einem Software-Architekten hinsichtlich informal formulierten Anforderungen an die Software bewertet werden. Die Arbeit zeigt, wie wir Validierungstests mit Hilfe der Testabdeckungsmethoden Property Mutation, Unique First Cause und Property Inactive Clause konstruieren. Diese Methoden ermöglichen bestimmte Garantien bezüglich der Korrektheit einer Spezifikation. Außerdem beschreiben wir Algorithmen zur Testgenerierung, die auf SMT-Solving beruhen. Schlussendlich wird die Implementierung von STLInspector dargestellt und das Programm auf mehrere STL Formeln angewendet, um zu zeigen, wie effektiv STLInspector fehlerhafte Formeln findet. Da diese Bachelorarbeit Validierung temporal-logischer Formeln ermöglicht, leistet sie einen Beitrag dazu, die Anwendung formaler Spezifikationen in der Industrie zu verbessern.

To Paul Mayer.

Acknowledgement

I am grateful for the support of various people during my bachelor's thesis and my undergraduate studies in general.

First of all, I would like to thank Prof. Dr. Thomas Kropf for making the bachelor's thesis at Bosch Corporate Research possible. Further, I am much obliged for the assistance given by my two supervisors Hendrik Röhm and Thomas Heinz. Working with you was always a pleasure and I have learned a great deal from both of you. In Renningen the departments AEE and AEX, especially Hassan Muhammad Qayyum and Fabian Meyer, made me feel welcome right from the start.

Special thanks go to my family, above all Susanne Klopries-Mayer and Benjamin Mayer, for supporting me not only during my bachelor's thesis but all my life. Thank you, Harald and Björn Brandenburg, for proofreading my thesis and for unintentionally influencing me to study Computer Science.

In Tübingen I am fortunate to call the Getenv-Crew my friends and to have benefited throughout my studies from your solidarity. Heidenei - without you my time in Tübingen would have not been the same! In particular, Anne-Kathrin Mahlke, Jonas Uli Benn, Robert Eisele and Christian Brauner always provided me with indispensable advice. Beyond Tübingen, I would like to extend my gratitude to my oldest friends, the Hannover girls.

Last but not least I would like to express my appreciation to the Stiftung der Deutschen Wirtschaft for their financial assistance during my studies.

Contents

1 Introduction

1.1 Motivation

With the rise of autonomous software in various fields that have great impact on human life, the issue of trusting software is becoming increasingly relevant. For instance, in the field of medicine we can find surgical robots or assistance systems for the elderly that humans rely on. Likewise, autonomous cars and automatic pilot avionics begin to be established as alternative ways of transportation. Confidence in the accuracy of these kinds of products is thus indispensable. It is achieved by verification and validation of the software.

The systems described above are examples of so-called cyber-physical systems (CPS). A CPS is the interconnection of embedded systems that is organized over networks, i.e. in CPS the physical parts of the system (mechanics, thermodynamics, sensors, etc.) interact via technologies such as the Internet closely with software components [1].

Designing a CPS, or any software for that matter, normally starts out with informal descriptions about how the system is supposed to behave. These descriptions are called requirements. Requirements written in natural language have the advantage of a greater understanding among various stakeholders. However, they are often ambiguous and can also not be used for automatic verification. Automated testing techniques, such as model checking, depend upon formalizations of requirements, namely specifications [2]. With formal specifications we also gain explicitness and conciseness in comparison to informal descriptions of requirements. Temporal logics provide a compact mathematical formalism for specifications of CPS since they have been proven valuable for expressing timed behaviour of systems [3]. In this exposition we consider only specifications written in Signal Temporal Logic (STL) [4].

As trust in the specification is the fundamental basis for trust in the implementation, verification of CPS is only possible after validating the temporal logic specification the system is based on [2]. Let us assume a system is implemented based on a specification of some requirements and tests of the finished implementation show that the system's behavior differs from what the requirements desired. Then we would call the system faulty. There are many sources for a fault in a system:

First, the implementation itself could contain a programming error (logic error) that results in unintended behavior of the system. If so the fault could be searched for by conventional debugging techniques. Secondly, the system could be free of logic errors but not implement the specification correctly. In this case automated testing techniques that use the specification as input could help in localizing the fault. And thirdly, the specification could not be the correct formalization of the requirements that were given. In other words, system properties characterized by the specification are different from properties the requirements described. Then the system would not behave in the way it was intended by the requirements since the specification is used as a foundation for the implementation.

In both the second and third scenario we need to be able to validate that the specification is the correct formalization of the requirements. In the second scenario demonstrating correctness of the specification is vital because the specification is used for automated testing and tests are meaningless if the specification does not conform with the requirements. In the third scenario the specification needs to be proven faultless due to the fact that a faulty specification results in a faulty implementation. Hence, our goal is to design a tool that enables engineers to give certain guarantees concerning the accuracy of their Signal Temporal Logic specification.

1.2 Goal

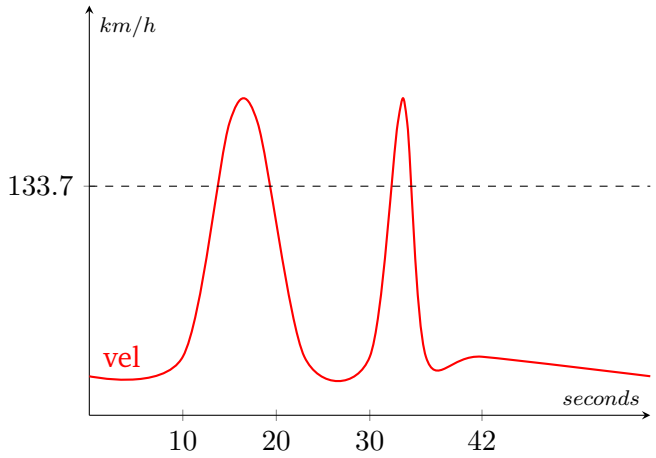

Fig. 1.1: Validation test

Formalizing requirements using STL is rather difficult due to the fact that STL formulae tend to be quite complex. Evidence for this statement is suggested by an investigation conducted by Dokhanchi et al. [5]. The investigation included an online survey where participants had to write STL specifications for several textual requirements. It was found that for many requirements the formalizations differed widely and most proposed formalizations did not fit the requirements given. Therefore the urge arises for a tool that provides the possibility to check whether an

STL formula that serves as a specification of given requirements conforms to these requirements. The main objectives of the tool are to deepen the understanding of a given STL formula and to support the engineer while using STL as a formalism. We achieve these objectives by letting the users of our tool validate specifications based on validation tests that are graphs as shown in Figure 1.1. This reduction of the complex STL formalism to a simple graphical format even enables other uses of the tool without the knowledge of STL to validate the specification.

The tool that was written as part of this Bachelor's thesis is called STLInspector. Given a specification of a system, STLInspector produces test cases in the form of an output graph of the system such as the one depicted in Figure 1.1. The tests either satisfy or do not satisfy the specification. During the validation the user has to decide whether the tests conform to the requirements that were specified or not. If this classification differs from the satisfaction value of the tests in regard to the specification, then we can deduce that the specification is not correct. The following example will illustrate the validation procedure:

Let us assume the requirement

At some point of time in the first 42 seconds the vehicle speed (vel) will rise above 133.7 km/h and from that point on stay above 133.7 km/h for at least 11 seconds.

is specified with the STL formula

$$\mathcal{F}_{[0,42]}((vel > 133.7) \Rightarrow (\mathcal{G}_{[0,11]}(vel > 133.7)))$$

Then a test such as the one in Figure 1.1 would show that the proposed specification is not correct. This is due to the fact that while the test satisfies the specification formula, it does not conform to the requirement since the velocity does not stay above 133.7 km/h long enough.

This thesis will cover the theoretical groundwork of STLInspector and overview its design and implementation. The second chapter introduces temporal logics, defines syntax and semantics of STL and exemplifies how to formalize requirements using the described logic. The third chapter states how STLInspector transforms a formula that is supposed to be tested into multiple new temporal logic formulae that create the basis for test generation. Furthermore, this chapter also explains the purpose of guarantees in the validation process of STL specifications. The fourth chapter deals with algorithms used to generate tests based on the new temporal logic formulae obtained as described in the third chapter. And finally, the fifth chapter covers the practical realization of all the theoretical aspects described beforehand and gives details on the implementation of STLInspector.

2 Temporal Logic

Statements such as "the engine is on" or "the velocity is higher than 133.7 km/h" can be described using propositional logic. Temporal logic extends propositional logic such that we can specify propositions over the course of time. Temporal operators to characterize circumstances that *eventually* happen or that are *always* the case are introduced. Thus, we can express conditions like "eventually, the velocity is higher than 133.7 km/h" or "the engine is always on" [6].

Requirements of software or hardware can be formalized using temporal logic. Therefore, temporal logic is often applied in formal verification [3]. There are multiple variations of temporal logic that are used for different scenarios. For example, to express unpredictable behavior of a system one might use a temporal logic that considers parallel time lines and therefore allows for branching whenever the system might behave in diverse ways [6]. However, in the following we will focus on temporal logics with only one single time line.

2.1 Signal Temporal Logic

Remark 1. *In the following $\mathbb{R}$ denotes the set of real numbers, $\mathbb{N}$ the set of natural numbers and $\mathbb{B}$ the set of the two Boolean values $\top$ (true) and $\bot$ (false). Furthermore, let χ^B be a finite set of Boolean-valued variables and χ^R a finite set of real-valued variables.*

In Signal Temporal Logic (STL) [4] time is continuous and the logic details pre- and postconditions of atomic propositions on an endless time line. Atomic propositions are statements that are either true or false. An atomic proposition can have one of the two forms:

Definition 1. *A Boolean atomic proposition p is simply a Boolean-valued variable $p \in \chi^B$.*

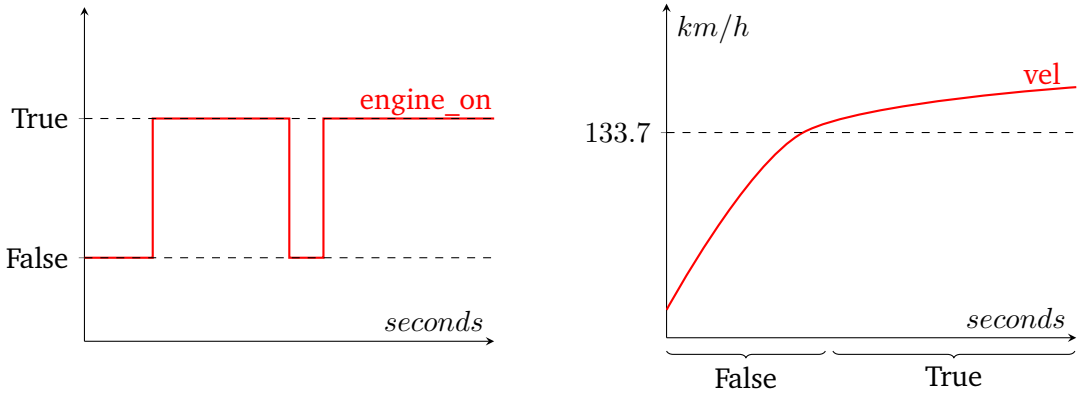

Fig. 2.1: Example of the Boolean atomic proposition $engine\_on$ (left) and the real atomic proposition $vel > 133.7$ (right)

Definition 2. *Let $x \in (\chi^R)^n$ be a vector of n variables, $c \in \mathbb{R}^n$ a vector of n real values, $b \in \mathbb{R}$ a real constant and $\sim \in \{\leq, <, \geq, >, \equiv, \not\equiv\}$ a relational operator where $\sim\colon \mathbb{R} \times \mathbb{R} \to \mathbb{B}$. With this we define a real atomic proposition as a formula of the form*

$$c^T \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \sim b$$

with T denoting vector transposition.

An example for a Boolean atomic proposition is the proposition "the engine is on" from above. Note that atomic propositions are interpreted over the course of time. Therefore, a visualization of the Boolean atomic proposition described by the variable $engine\_on$ could look like in Figure 2.1 on the left. The scenario in Figure 2.1 could be one where the car is turned on and off multiple times. Real atomic propositions are used to model statements such as "the velocity is higher than 133.7 km/h" which we could formalize as $vel > 133.7$. The graph on the right in Figure 2.1 depicts the scenario of an atomic proposition in this format over the course of time .

The examples in Figure 2.1 show that atomic propositions hold different values over the course of time. Assigning a value to an atomic proposition means that we valuate the variable(s) used in the atomic proposition. For this we need the following functions:

Definition 3. *We define a valuation function V that maps a variable to a value. There are two types of valuation functions:*

- *$V^B : \chi^B \to \mathbb{B}$, the Boolean valuation function that maps a Boolean variable to a Boolean value.*

- $V^R : \chi^R \to \mathbb{R}$, *the real valuation function that maps a real variable to a real value.*

We illustrate the use of valuation functions in the following examples: With $\chi^B = \{a, b\}$ possible valuations are $V^B(a) = \top$ and $V^B(b) = \bot$. With $\chi^B = \{x, y, z\}$ possible valuations are $V^R(x) = 133.7$, $V^R(y) = 42$ and $V^R(z) = 3.1415$.

Now that we can valuate the variables of atomic propositions, we want to review which values satisfy the proposition (i.e. the proposition evaluated to true). We are therefore in need of the definition of an evaluation function:

Definition 4. *Let AP denote the set of all atomic propositions. We define an evaluation function π with*

$$\pi : (\chi^B \to \mathbb{B}) \times (\chi^R \to \mathbb{R}) \times AP \longrightarrow \mathbb{B}$$

that, given a Boolean valuation function V^B and a real valuation function V^R, maps each atomic proposition p to a truth value:

$$\pi(V^B, V^R, p) = \begin{cases} V^B(p) & , p \text{ is a Boolean atomic proposition} \\ c^T \cdot \begin{pmatrix} V^R(x_1) \\ \vdots \\ V^R(x_n) \end{pmatrix} \sim b & , p \text{ is a real atomic proposition} \end{cases}$$

where $\begin{pmatrix} V^R(x_1) \\ \vdots \\ V^R(x_n) \end{pmatrix}$ *is a vector of reals and $\cdot$ denotes the scalar product.*

Atomic propositions can be valuated with different values over the course of time and therefore hold or not hold at distinct points of time. For example, at point of time t the atomic proposition p might be valuated using the valuation function v_t^B if p is Boolean or v_t^R if p is real. Assuming $\pi(V_t^B, V_t^R, p) = \top$, the proposition p holds at time point t. However, at point of time t' with $t' \neq t$ it might be the case that the new valuation functions used for this time point result in $\pi(V_{t'}^B, V_{t'}^R, p) = \bot$, i.e. the proposition p does not hold at time point t'. Consequently, we need a function that distinguishes which values are assigned to the propositions at which point of time. In other words, we need a function to determine which valuation functions are used at when. This function is what we call a *signal*.

Definition 5. *A signal s is a function that maps each point of time to a pair of valuation functions; one for Boolean-valued variables and one for real-valued variables.*

$$s : \mathbb{R}_0^+ \longrightarrow (\chi^B \to \mathbb{B}) \times (\chi^R \to \mathbb{R})$$

$$t \mapsto (V^B, V^R)$$

Remark 2. *If the atomic proposition p is fulfilled by the signal s and a point of time $t \in \mathbb{R}_0^+$, we write $(s,t) \models p$, with the interpretation*

$$(s,t) \models p \;\Leftrightarrow\; \pi(V_t^B, V_t^R, p) = \top$$

where $s(t) = (V_t^B, V_t^R)$.
In this case we say that the atomic proposition p holds at point of time t or that p is satisfied by the value that the respective valuation function assigns to p. A real atomic proposition can be satisfied by multiple real numbers while a Boolean atomic proposition only holds when they are assigned the value true.

With the help of atomic propositions, their evaluation function and signals we can now proceed with a formal definition of STL:

Definition 6. *The syntax of Signal Temporal Logic (STL) is defined by the following Backus-Naur form of a formula ϕ:*

$$\phi \;::=\; p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \,\mathcal{U}_{[a,b]}\, \phi_2$$

where p is an atomic proposition, $\neg$ and $\vee$ are the propositional operators for negation and disjunction that are presupposed to be known and $\mathcal{U}_{[a,b]}$ is a new symbol for the temporal operation Until bounded by the interval $[a,b]$. The interval bounds of Until are points of time $a, b \in \mathbb{R}_0$.

The STL formula $\phi_1 \,\mathcal{U}_{[a,b]}\, \phi_2$ states that the STL formula ϕ_1 holds at every point of time until somewhere in between a and b the STL formula ϕ_2 holds. An example for a signal that satisfies an Until formula is given in Figure 2.2.

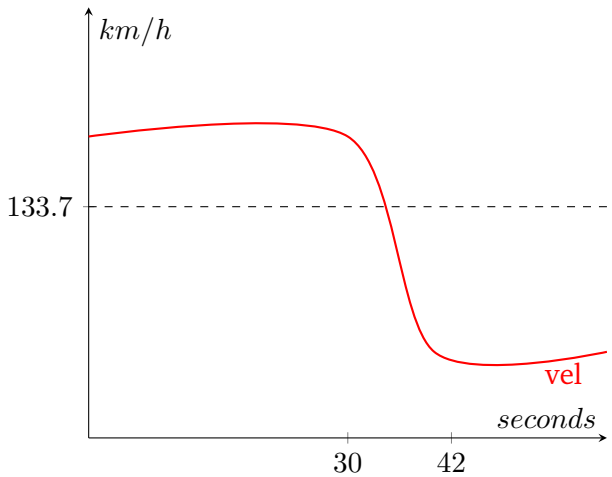

Fig. 2.2: Example signal for Until-formula $\phi = (vel > 133.7)\,\mathcal{U}_{[33,42]}\,(vel \leq 133.7)$

Remark 3. *In the following let $s \models \phi$ abbreviate $(s, 0) \models \phi$ for a STL formula ϕ. Furthermore, we define $s^i(t) := s(t + i)$ with $i \in \mathbb{R}_0$ which shifts the signal s in time. Note that we write $s^i := s^i(0)$.*

Using a signal s we can establish the STL semantics as follows:

Definition 7. *The STL semantics are defined by*

$s \models p$ *iff* $\pi_p(s(0))$ *is true.*

$s \models \neg\phi$ *iff not* $s \models \phi$.

$s \models \phi_1 \vee \phi_2$ *iff* $s \models \phi_1$ *or* $s \models \phi_2$.

$s \models \phi_1 \, \mathcal{U}_{[a,b]} \, \phi_2$ *iff* $\exists \, t_1, a \leq t_1 \leq b \, : \, s^{t_1} \models \phi_2$ *and* $\forall \, t_2, 0 \leq t_2 < t_1 \, : \, s^{t_2} \models \phi_1$

Based on the STL definition above we can further introduce the following operators:

Definition 8. *Given two STL formulae ϕ_1 and ϕ_2. Then $\phi_1 \wedge \phi_2 = \neg(\neg\phi_1 \vee \phi_2)$ is an STL formula that is called a conjunction of ϕ_1 and ϕ_2.*

Definition 9. *Given two STL formulae ϕ_1 and ϕ_2. Then $\phi_1 \rightarrow \phi_2 = \neg\phi_1 \vee \phi_2$ is an STL formula that is called a implication, i.e. ϕ_1 implies ϕ_2.*

Definition 10. *Given an STL formula ϕ. Then $\mathcal{N}_{[a]}(\phi) = \top \, \mathcal{U}_{[a,a]} \, \phi$ with $a \in \mathbb{R}$ is an STL formula. We call $\mathcal{N}_{[a]}$ the temporal Next operator with bound a. The Next operator declares that after a time steps the formula ϕ holds. (Note that for a signal s we have $s \models \mathcal{N}_{[a]}(\phi)$ if and only if $s^a \models \phi$.)*

Definition 11. *Given an STL formula ϕ. Then $\mathcal{F}_{[a,b]}(\phi) = \top \, \mathcal{U}_{[a,b]} \, \phi$ with $a, b \in \mathbb{R}$ is an STL formula. We call $\mathcal{F}_{[a,b]}$ the temporal Finally operator bound to the interval $[a, b]$. Finally states that at some point in time in the interval $[a, b]$ the formula ϕ holds.*

Definition 12. *Given an STL formula ϕ. Then $\mathcal{G}_{[a,b]}(\phi) = \neg\mathcal{F}_{[a,b]}(\neg\phi)$ with $a, b \in \mathbb{R}$ is an STL formula. We call $\mathcal{G}_{[a,b]}$ the temporal Globally operator bounded by the interval $[a, b]$. With the Globally operator we can affirm that ϕ holds at every single state in between a to b.*

Definition 13. *Given two STL formulae ϕ_1 and ϕ_2. Then $\phi_1 \mathcal{R}_{[a,b]} \phi_2 = \neg(\neg\phi_1 \mathcal{U}_{[a,b]} \neg\phi_2)$ with $a, b \in \mathbb{R}$ is an STL formula. We call $\mathcal{R}_{[a,b]}$ the temporal Release operator bound to the interval $[a, b]$. It states that ϕ_2 holds at every point in time up to a point in between a and b where both ϕ_1 and ϕ_2 hold, thereby ϕ_1 releases ϕ_2. It is also valid if ϕ_1 does not hold at all, hence ϕ_2 holds all the way until b (then ϕ_2 was released right in the beginning).*

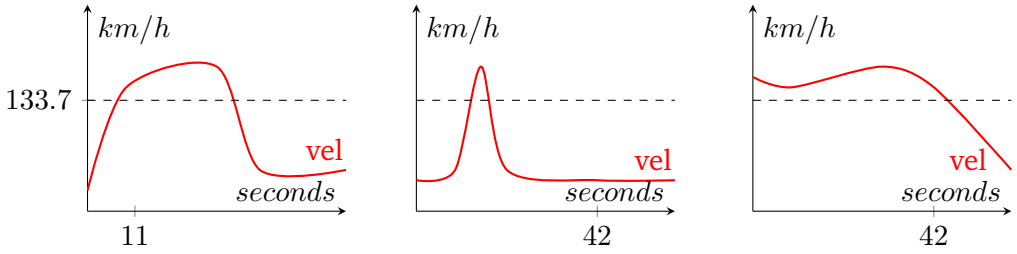

Fig. 2.3: Example signals for $\phi = \mathcal{N}_{[11]}(vel > 133.7)$ (right), $\phi = \mathcal{F}_{[0,42]}(vel > 133.7)$ (middle), $\phi = \mathcal{G}_{[0,42]}(vel > 133.7)$ (left)

For the three new temporal operators Next, Finally and Globally the graphs of the Figure 2.3 depicted example signals. One temporal logic formula may be satisfied by multiple signals. For example the signals for Next (on the left in Figure 2.3) and Globally (on the right in Figure 2.3) also satisfy the Finally formula defined in the caption of the figure. This is due to the fact that Finally only states that its operand has to be fulfilled at least once and this is the case in the graphs on the right and left of Figure 2.3 where the atomic proposition $vel > 133.7$ holds multiple seconds. Therefore a temporal logic formula can be seen as a description for a set of signals.

The STL operators have the following precedence in descending order (hence $\neg$ has the highest precedence): $\neg, \mathcal{N}, \mathcal{G}, \mathcal{F}, \mathcal{U}, \mathcal{R}, \wedge, \vee, \rightarrow$.

Example 1. *Using Signal Temporal Logic we can formalize the statement*

"It is always the case that if the car is moving its engine is on."

as follows: The car is moving means that its velocity is higher than 0, thus resulting in the real atomic proposition "$vel > 0$". The state of the engine can either be on or off, leaving us with a Boolean variable "engine_on" that is true if and only if the engine is on. Since the statement says that "It is always the case that...", we use the temporal Globally operator and can summarize the formalization as

$$\mathcal{G}_{[0,t]}(vel > 0 \rightarrow engine\_on)$$

where $t \in \mathbb{R}_0$ denotes the end of the time we consider for the statement.

We have now covered all aspects of Signal Temporal Logic that are necessary to understand the formalization of requirements and the groundwork for the test generation. The next chapter explains how meaningful tests can originate from so-called test predicates that are generated based on an original temporal logic formula by the means of coverage criteria. We introduce three different coverage criteria, where property mutation is the most important one.

3 Coverage Criteria

> *Program testing can be a very effective way to show the presence of bugs but is hopelessly inadequate for showing their absence.*
>
> **— Edsger Wybe Dijkstra**
> (1972)

From this so called Dijkstra's law we can infer that software can never be tested for every possible bug but that we always have to limit ourselves to a sample of test cases. This sampling should, however, be done wisely. Therefore, we formulate prerequisites our test suite should fulfill. A set of coherent prerequisites make up a coverage criterion. A coverage criterion allows us to claim that certain aspects of the system under test have been tested and gives us a way to measure the degree to which the system was tested with regard to these aspects [7]. For example, a given program could be tested with the goal to cover all lines of source code during the execution of a test suite. Or a coverage criterion for a GUI could be to test every graphical element it holds.

Coverage criteria often include rules for modeling test cases [7]. These rules provide a systematic way to exclude certain faults. When we let a test suite run on a system, we can measure the percentage of test cases that did not find an error. I.e. this percentage of all faults that fall under the used coverage criterion was certainly omitted by the system under test. We deduce our test cases for STLInspector using coverage criteria and can hence give guarantees in form of a such a percentage for the correctness of the temporal logic specifications.

Since we aim to test specifications in the form of temporal logic formulae, we will now give definitions for a *test* and a *test predicate* in our context:

Definition 14. *A signal T is called a test of the temporal logic formula ϕ if either $T \models \phi$ or $T \not\models \phi$. A test suite is a set of test cases $\{T_1, \ldots, T_n\}$ and its size is the number n of test cases it consists of. [8]*

Remark 4. *In the following we will use the phrases "test", "test case" and "test signal" as synonyms.*

The coverage criteria used are made up of rules that produce temporal logic formulae. Each of these formulae stand for one requirement in regard to our test objective. They make up the set of so-called test predicates.

Definition 15. *A coverage criteria C is a set of rules $\{R_1, \ldots, R_n\}$ where each rule is a mapping*

$$R_i : S \to S,\ \Phi \mapsto \Psi$$

with S as the set of all STL formulae.

Definition 16. *Given a coverage criterion C and a formula under test ϕ. We call ψ a test predicate of ϕ if $\psi = R_i(\phi)$ for some $R_i \in C$. A test predicate is therefore an STL formula produced by the rules of a coverage criterion. A test predicate ψ is satisfied by a test signal T if there exists $i \in \mathbb{N}_0$ such that $T^i \models \psi$. [8]*

A test suite satisfies a coverage criterion with regard to a given formula if for each test predicate r produced by the rules of the coverage criteria there is a test case T such that $T \models r$. A test suite satisfies a coverage criterion if each of its tests satisfy the criterion.

Now that we have established the definitions of test case, test predicate and coverage criteria in general, the testing process of STLInspector can be summarized as follows: We begin with the generation of test predicates derived from a temporal logic formula that represents a specification using our coverage criteria. We then model a test suite using these predicates such that the test suite satisfies the given coverage criterion. Finally, if the user correctly classifies all tests, we can guarantee that all faults described by the coverage criterion are omitted by the specification.

3.1 Property Mutation

In the case of property mutation the coverage criterion aims to differentiate between the original formula and formulae representing versions of the original formula that adapt small syntactic errors [7]. For example, given the formula

$$\phi = \mathcal{F}_{[0,42]}(vel > 133.7)$$

we could check whether we have used the correct relational expression by finding a test that distinguishes ϕ from the formula

$$\psi = \mathcal{F}_{[0,42]}(vel < 133.7)$$

A test that could be used is shown in Figure 3.1. It shows a scenario where the velocity is always greater than 133.7 km/h, which obviously satisfies the statement that the velocity has to be greater than 133.7 at least once in between 0 and 42.

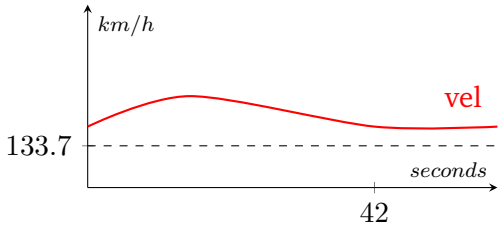

Fig. 3.1: Validation test to distinguish ϕ and its mutant ψ

Thus, the test satisfies ϕ. However, ψ is not satisfied by the test and since the two formulae therefore have different satisfaction values regarding the signal in Figure 3.1, the test can distinguish between the two formulae. As ψ holds $<$ instead of $>$ as in the original formula ϕ, we can - given the user validated the test to be conform to her requirement - guarantee that the user did not accidentally choose $<$ instead of $>$ in the relational expression. New formulae that are adaptations of a given formula under test, such as ψ is for ϕ, are called mutants.

Definition 17. *Given a formula ϕ and a version of it named ψ that differs from ϕ in a single syntactic change. Then we call ψ a mutant of ϕ. If a test T can distinguish between the two formulae, meaning that either $T \models \phi \wedge T \not\models \psi$ or $T \not\models \phi \wedge T \models \psi$ holds, we say that T kills ψ.*

Definition 18. *A mutation operator is a rule that is applied to a formula to create mutants [7]. Mutation operators are the rules that make up the coverage criterion of property mutation.*

The mutation operators mimic various typical STL formalization errors such as using the wrong relation like we have seen above, in- or decreasing the interval bounds of a formula or using an inaccurate temporal operator ($\mathcal{G}$ instead of $\mathcal{F}$). Since the test that STLInspector creates are based on mutants for a formula the user inserted and each test kills at least one mutant, we can hence guarantee that the formalization precludes the faults described by the mutation operators as long as the tests are correctly classified by the user.

Fraser and Wotawa introduced multiple mutation operators for Linear Temporal Logic (LTL) in [8]. In this thesis we extend these operators for STL and also introduce completely new mutation operators such as the Interval Replacement Operator. The next pages define these mutation operators that were used as coverage criteria for STLInspector. If the mutation operator does not change the interval of STL operators, the intervals are simply ignored in the definition of the mutation operator. All examples for the mutation operators are performed on the STL formula

$$\phi = a\mathcal{U}_{[0,1]}b \wedge \mathcal{F}_{[1,2]}(x > 3)$$

with the Boolean atomic propositions a, b and the real atomic proposition $x > 3$.

3.1.1 Operand Replacement Operator

For a given formula mutants are created for each atomic proposition by once replacing the proposition with every other proposition that is part of the formula. Notice that we do not replace an operand if the mutated expression would result in a constant (i.e. *x operator x*). Let AP denote the set of atomic propositions.

$$\begin{aligned} oro(p) &= \{a | a \in AP\} \\ oro(X \star Y) &= \{x \star Y | x \in oro(X)\} \cup \{X \star y | y \in oro(Y)\} && \star \in \{\vee, \wedge, \rightarrow, \mathcal{U}, \mathcal{R}\} \\ oro(\star\, X) &= \{\star\, x | x \in oro(X)\} && \star \in \{\neg, \mathcal{F}, \mathcal{G}, \mathcal{N}\} \end{aligned}$$

Example:

$$\begin{aligned} oro(\phi) = \{ & a\mathcal{U}_{[0,1]} a \wedge \mathcal{F}_{[1,2]}(x > 3), \\ & a\mathcal{U}_{[0,1]} b \wedge \mathcal{F}_{[1,2]} a, \\ & b\mathcal{U}_{[0,1]} b \wedge \mathcal{F}_{[1,2]}(x > 3), \\ & a\mathcal{U}_{[0,1]} b \wedge \mathcal{F}_{[1,2]} b, \\ & (x > 3)\mathcal{U}_{[0,1]} b \wedge \mathcal{F}_{[1,2]}(x > 3), \\ & a\mathcal{U}_{[0,1]}(x > 3) \wedge \mathcal{F}_{[1,2]}(x > 3)\} \end{aligned}$$

3.1.2 Logical Operator Replacement Operator

For a given formula mutants are created for each binary logical operator by once replacing the operator with every other existing binary logical operator ($\vee, \wedge$ and $\rightarrow$).

$$\begin{aligned} lro(p) &= \{p\} \\ lro(X \star_1 Y) &= \{X \star_2 Y | \star_2 \in \{\vee, \wedge, \rightarrow\} \setminus \{\star_1\}\} \\ &\quad \cup \{x \star_1 Y | x \in lro(X)\} \\ &\quad \cup \{X \star_1 y | y \in lro(Y)\} && \star_1 \in \{\vee, \wedge, \rightarrow\} \\ lro(X \star Y) &= \{x \star Y | x \in lro(X)\} \cup \{X \star y | y \in lro(Y)\} && \star \in \{\mathcal{U}, \mathcal{R}\} \\ lro(\star\, X) &= \{\star\, x | x \in lro(X)\} && \star \in \{\neg, \mathcal{F}, \mathcal{G}, \mathcal{N}\} \end{aligned}$$

Example:

$$\begin{aligned} lro(\phi) = \{ & a\mathcal{U}_{[0,1]} b \vee \mathcal{F}_{[1,2]}(x > 3), \\ & a\mathcal{U}_{[0,1]} b \rightarrow \mathcal{F}_{[1,2]} a\} \end{aligned}$$

3.1.3 Temporal Operator Replacement Operator

For a given formula mutants are created for each temporal operator by once replacing the operator with every other existing temporal operator that has the same number of operands. Thus Until can be replaced by Release and vice versa, and Next, Finally and Globally can replace each other. Notice that when replacing Finally or Globally through Next, the interval is reduced to only the higher limit. The other way around the index of the replaced Next-operator is de- and increased once to make up the lower and upper interval bound.

$$
\begin{aligned}
tro(p) &= \{p\} \\
tro(X \star_1 Y) &= \{X \star_2 Y | \star_2 \in \{\mathcal{U}, \mathcal{R}\} \setminus \{\star_1\}\} \\
&\quad \cup \{x \star_1 Y | x \in tro(X)\} \\
&\quad \cup \{X \star_1 y | y \in tro(Y)\} && \star_1 \in \{\mathcal{U}, \mathcal{R}\} \\
tro(\star_1 Y) &= \{\star_2 Y | \star_2 \in \{\mathcal{F}, \mathcal{G}\} \setminus \{\star_1\}\} \\
&\quad \cup \{\star_1 y | y \in tro(Y)\} && \star_1 \in \{\mathcal{F}, \mathcal{G}\} \\
tro((\star_1)_{[a,b]} Y) &= \{\mathcal{N}_{[b]}\} \\
&\quad \cup \{\star_1 y | y \in tro(Y)\} && \star_1 \in \{\mathcal{F}, \mathcal{G}\} \\
tro(\mathcal{N}_{[a]} Y) &= \{(\star_2)_{[a-1,a+1]} Y | \star_2 \in \{\mathcal{F}, \mathcal{G}\}\} \\
&\quad \cup \{\mathcal{N}_{[a]} y | y \in tro(Y)\} \\
tro(X \star Y) &= \{x \star Y | x \in tro(X)\} \cup \{X \star y | y \in tro(Y)\} && \star \in \{\vee, \wedge, \rightarrow\} \\
tro(\neg X) &= \{\neg x | x \in tro(X)\}
\end{aligned}
$$

Example:

$$
\begin{aligned}
tro(\phi) = \{ & a\mathcal{R}_{[0,1]} b \wedge \mathcal{F}_{[1,2]}(x > 3), \\
& a\mathcal{U}_{[0,1]} b \wedge \mathcal{N}_{[2]}(x > 3), \\
& a\mathcal{U}_{[0,1]} b \wedge \mathcal{G}_{[1,2]}(x > 3)\}
\end{aligned}
$$

3.1.4 Relational Operator Replacement Operator

This mutation operator can only be used for signal temporal atomic propositions, thus for atomic propositions p of the format $p = c^T \cdot x \star b$ with $\star \in \{\equiv, \neq, >, \geq, <, \leq\}$. For a given formula mutants are created for each one of these atomic proposition by once replacing the relational operator $\star$ with every other existing relational operator except its exact opposite. This is due to the fact that replacing a relational operator of an atomic proposition by its exact opposite would result in a semantically equivalent

mutant to the mutant that is produced by applying the Atomic Proposition Negation Operator (see definition below) on the atomic proposition.

$$
\begin{aligned}
rro(c^T \cdot x \equiv b) &= \{c^T \cdot x \star_2 b | \star_2 \in \{>, \geq, <, \leq\}\} \\
rro(c^T \cdot x \neq b) &= \{c^T \cdot x \star_2 b | \star_2 \in \{>, \geq, <, \leq\}\} \\
rro(c^T \cdot x > b) &= \{c^T \cdot x \star_2 b | \star_2 \in \{\equiv, \neq, \geq, <\}\} \\
rro(c^T \cdot x \leq b) &= \{c^T \cdot x \star_2 b | \star_2 \in \{\equiv, \neq, \geq, <\}\} \\
rro(c^T \cdot x \geq b) &= \{c^T \cdot x \star_2 b | \star_2 \in \{\equiv, \neq, >, \leq\}\} \\
rro(c^T \cdot x < b) &= \{c^T \cdot x \star_2 b | \star_2 \in \{\equiv, \neq, >, \leq\}\} \\
rro(X \star Y) &= \{x \star Y | x \in rro(X)\} \\
&\quad \cup \{X \star y | y \in rro(Y)\} && \star \in \{\vee, \wedge, \rightarrow, \mathcal{U}, \mathcal{R}\} \\
rro(\star X) &= \{\star x | x \in rro(X)\} && \star \in \{\neg, \mathcal{F}, \mathcal{G}, \mathcal{N}\}
\end{aligned}
$$

Example:

$$
\begin{aligned}
rro(\phi) = \{ & a\mathcal{U}_{[0,1]} b \wedge \mathcal{F}_{[1,2]}(x \equiv 3), \\
& a\mathcal{U}_{[0,1]} b \wedge \mathcal{F}_{[1,2]}(x \neq 3), \\
& a\mathcal{U}_{[0,1]} b \wedge \mathcal{F}_{[1,2]}(x \geq 3), \\
& a\mathcal{U}_{[0,1]} b \wedge \mathcal{F}_{[1,2]}(x < 3)\}
\end{aligned}
$$

3.1.5 Interval Replacement Operator

We produce mutants using this operator by increasing or decreasing the size of the interval for each signal temporal logic operator. Either the left or the right interval limit is increased/reduced by one, hence for each temporal logic operator a total of four mutants is created. For the Next-operator the index is simply pushed one time step forward/backward.

$$
\begin{aligned}
iro(p) &= \{p\} \\
iro(X \star_{[a,b]} Y) &= \{X \star_{[a-1,b]} Y, X \star_{[a+1,b]} Y, \\
&\quad X \star_{[a,b-1]} Y, X \star_{[a,b+1]} Y\} \\
&\quad \cup \{x \star_{[a,b]} Y | x \in iro(X)\} \\
&\quad \cup \{X \star_{[a,b]} y | y \in iro(Y)\} && \star \in \{\mathcal{U}, \mathcal{R}\}, a \geq 1 \\
iro(\star_{[a,b]} X) &= \{\star_{[a-1,b]} X, \star_{[a+1,b]} X, \star_{[a,b-1]} X, \star_{[a,b+1]} X\} \\
&\quad \cup \{\star_{[a,b]} x | x \in iro(X)\} && \star \in \{\mathcal{F}, \mathcal{G}\}, a \geq 1 \\
iro(X \star_{[0,b]} Y) &= \{X \star_{[0,b-1]} Y, X \star_{[1,b]} Y, X \star_{[0,b+1]} Y\} \\
&\quad \cup \{x \star_{[0,b]} Y | x \in iro(X)\} \\
&\quad \cup \{X \star_{[0,b]} y | y \in iro(Y)\} && \star \in \{\mathcal{U}, \mathcal{R}\}, b \geq 1
\end{aligned}
$$

$$
\begin{aligned}
iro(\star_{[0,b]}X) &= \{\star_{[0,b-1]}X, \star_{[1,b]}X, \star_{[0,b+1]}X\} && \\
&\cup \{\star_{[0,b]}x | x \in iro(X)\} && \star \in \{\mathcal{F}, \mathcal{G}\}, b \geq 1 \\
iro(X \star_{[a,b]} Y) &= \{X \star_{[a-1,b]} Y, X \star_{[a,a]} Y, X \star_{[a,b+1]} Y\} && \\
&\cup \{x \star_{[a,b]} Y | x \in iro(X)\} && \\
&\cup \{X \star_{[a,b]} y | y \in iro(Y)\} && \star \in \{\mathcal{U}, \mathcal{R}\}, b = a+1 \\
iro(\star_{[a,b]}X) &= \{\star_{[a-1,b]}X, \star_{[a,a]}X, \star_{[a,b+1]}X\} && \\
&\cup \{\star_{[a,b]}x | x \in iro(X)\} && \star \in \{\mathcal{F}, \mathcal{G}\}, b = a+1 \\
iro(\mathcal{N}_{[a]}X) &= \{\mathcal{N}_{[a-1]}X, \mathcal{N}_{[a+1]}X\} && \\
&\cup \{\mathcal{N}_{[a,b]}x | x \in iroX\} && a \geq 1 \\
iro(\mathcal{N}_{[0]}X) &= \{\mathcal{N}_{[1]}X\} \cup \{\mathcal{N}_{[a,b]}x | x \in iroX\} && \\
iro(X \star Y) &= \{x \star Y | x \in iro(X)\} && \\
&\cup \{X \star y | y \in iro(Y)\} && \star \in \{\vee, \wedge, \rightarrow\} \\
iro(\neg X) &= \{\neg x | x \in iro(X)\} &&
\end{aligned}
$$

Example:

$$
\begin{aligned}
iro(\phi) = \{&a\mathcal{U}_{[0,0]}b \wedge \mathcal{F}_{[1,2]}(x > 3), \\
&a\mathcal{U}_{[0,2]}b \wedge \mathcal{F}_{[1,2]}(x > 3), \\
&a\mathcal{U}_{[0,1]}b \wedge \mathcal{F}_{[0,2]}(x > 3), \\
&a\mathcal{U}_{[0,1]}b \wedge \mathcal{F}_{[2,2]}(x > 3), \\
&a\mathcal{U}_{[0,1]}b \wedge \mathcal{F}_{[1,3]}(x > 3)\}
\end{aligned}
$$

3.1.6 Atomic Proposition Negation Operator

For a given formula one mutant is created for each atomic proposition by negating the proposition.

$$
\begin{aligned}
ano(p) &= \{\neg p\} && \\
ano(X \star Y) &= \{x \star Y | x \in ano(X)\} \cup \{X \star y | y \in ano(Y)\} && \star \in \{\vee, \wedge, \rightarrow, \mathcal{U}, \mathcal{R}\} \\
ano(\star\, X) &= \{\star\, x | x \in ano(X)\} && \star \in \{\neg, \mathcal{F}, \mathcal{G}, \mathcal{N}\}
\end{aligned}
$$

Example:

$$
\begin{aligned}
ano(\phi) = \{&(\neg a)\mathcal{U}_{[0,1]}b \wedge \mathcal{F}_{[1,2]}(x > 3), \\
&a\mathcal{U}_{[0,1]}(\neg b) \wedge \mathcal{F}_{[1,2]}(x > 3), \\
&a\mathcal{U}_{[0,1]}b \wedge \mathcal{F}_{[1,2]}(\neg(x > 3))\}
\end{aligned}
$$

3.1.7 Expression Negation Operator

For a given formula one mutant is created for each logical expression by negating the expression. An expression in this case is every logical connective. Atomic propositions, however, are not expressions. Their negation is already accomplished by the Atomic Proposition Negation Operator described above.

$$
\begin{aligned}
eno(p) &= \{p\} \\
eno(\neg X) &= \{X\} \cup \{\neg x | x \in eno(X)\} \\
eno(X \star Y) &= \{neg(X \star Y)\} \cup \{x \star Y | x \in eno(X)\} \\
&\quad \cup \{X \star y | y \in eno(Y)\} && \star \in \{\vee, \wedge, \rightarrow\} \\
eno(X \star Y) &= \{x \star Y | x \in eno(X)\} \cup \{X \star y | y \in eno(Y)\} && \star \in \{\mathcal{U}, \mathcal{R}\} \\
eno(\star\, X) &= \{\star\, x | x \in eno(X)\} && \star \in \{\mathcal{F}, \mathcal{G}, \mathcal{N}\}
\end{aligned}
$$

Example:

$$eno(\phi) = \{\neg(a\mathcal{U}_{[0,1]}b \wedge \mathcal{F}_{[1,2]}(x > 3))\}$$

3.1.8 Stuck-At Operator

For a given formula two mutants are created for each atomic proposition by replacing the proposition once with $\top$ and once with $\bot$.

$$
\begin{aligned}
sto(p) &= \{\top, \bot\} \\
sto(X \star Y) &= \{x \star Y | x \in sto(X)\} \cup \{X \star y | y \in sto(Y)\} && \star \in \{\vee, \wedge, \rightarrow, \mathcal{U}, \mathcal{R}\} \\
sto(\star\, X) &= \{\star\, x | x \in sto(X)\} && \star \in \{\neg, \mathcal{F}, \mathcal{G}, \mathcal{N}\}
\end{aligned}
$$

Example:

$$
\begin{aligned}
sto(\phi) = \{&\top\mathcal{U}_{[0,1]}b \wedge \mathcal{F}_{[1,2]}(x > 3), \\
&\bot\mathcal{U}_{[0,1]}b \wedge \mathcal{F}_{[1,2]}(x > 3), \\
&a\mathcal{U}_{[0,1]}\top \wedge \mathcal{F}_{[1,2]}(x > 3), \\
&a\mathcal{U}_{[0,1]}\bot \wedge \mathcal{F}_{[1,2]}(x > 3), \\
&a\mathcal{U}_{[0,1]}b \wedge \mathcal{F}_{[1,2]}\top, \\
&a\mathcal{U}_{[0,1]}b \wedge \mathcal{F}_{[1,2]}\bot\}
\end{aligned}
$$

3.1.9 Missing Condition Operator

For each logical binary expression in a given formula two mutants are created that hold either the first or the second operand of the logical binary expression in the same place but without the logical operator and the respective other operand. Implications generate only one mutant that holds the second operator, thus only the condition for the implication is missing.

$$
\begin{aligned}
mco(p) &= \{p\} \\
mco(\neg X) &= \{\neg x | x \in mco(X)\} \\
mco(X \rightarrow Y) &= \{Y)\} \cup \{x \star Y | x \in mco(X)\} \\
&\quad \cup \{X \star y | y \in mco(Y)\} \\
mco(X \star Y) &= \{X, Y\} \cup \{x \star Y | x \in mco(X)\} \\
&\quad \cup \{X \star y | y \in mco(Y)\} && \star \in \{\vee, \wedge\} \\
mco(X \star Y) &= \{x \star Y | x \in mco(X)\} \cup \{X \star y | y \in mco(Y)\} && \star \in \{\mathcal{U}, \mathcal{R}\} \\
mco(\star X) &= \{\star x | x \in mco(X)\} && \star \in \{\mathcal{F}, \mathcal{G}, \mathcal{N}\}
\end{aligned}
$$

Example:

$$
\begin{aligned}
mco(\phi) = \{&\mathcal{F}_{[1,2]}(x > 3), \\
&a\mathcal{U}_{[0,1]}b\}
\end{aligned}
$$

3.1.10 Missing Temporal Operator

For each temporal expression in a given formula one mutant is created that holds the operand of the temporal expression in the same place but without the temporal operator. For binary temporal operators two mutants are created where one holds the first and one the second operand.

$$
\begin{aligned}
mto(p) &= \{p\} \\
mto(\neg X) &= \{\neg x | x \in mto(X)\} \\
mto(\star X) &= \{X\} \cup \{\star x | x \in mto(X)\} && \star \in \{\mathcal{F}, \mathcal{G}, \mathcal{N}\} \\
mto(X \star Y) &= \{X, Y\} \cup \{x \star Y | x \in mto(X)\} \\
&\quad \cup \{X \star y | y \in mto(Y)\} && \star \in \{\mathcal{U}, \mathcal{R}\} \\
mto(X \star Y) &= \{x \star Y | x \in mto(X)\} \cup \{X \star y | y \in mto(Y)\} && \star \in \{\vee, \wedge, \rightarrow\}
\end{aligned}
$$

Example:

$$\begin{aligned} mto(\phi) = \{&a \wedge \mathcal{F}_{[1,2]}(x > 3), \\ &a \wedge \mathcal{F}_{[1,2]}(x > 3), \\ &a\mathcal{U}_{[0,1]}b \wedge (x > 3)\} \end{aligned}$$

3.1.11 Temporal Insertion Operator

For each logical expression of a given formula three mutants per operand are created by inserting the temporal operators Finally, Globally and Next in front of the operand. The interval limits a and b for the new temporal operators have to be defined beforehand.

$$\begin{aligned} tio(p) &= \{\mathcal{F}_{[a,b]}p, \mathcal{G}_{[a,b]}p, \mathcal{N}_{[a]}p\} \\ tio(\neg X) &= \{\neg\mathcal{F}_{[a,b]}X, \neg\mathcal{G}_{[a,b]}X, \neg\mathcal{N}_{[a]}X\} \cup \{\neg x | x \in tio(X)\} \\ tio(\star\, X) &= \{\star\, x | x \in tio(X)\} && \star \in \{\mathcal{F}, \mathcal{G}, \mathcal{N}\} \\ tio(X \star Y) &= \{x \star Y | x \in tio(X)\} \cup \{X \star y | y \in tio(Y)\} && \star \in \{\mathcal{U}, \mathcal{R}\} \\ tio(X \star Y) &= \{(\mathcal{F}_{[a,b]}X) \star Y, (\mathcal{G}_{[a,b]}X) \star Y, \\ &\quad (\mathcal{N}_{[a]}X) \star Y, X \star (\mathcal{F}_{[a,b]}Y), \\ &\quad X \star (\mathcal{G}_{[a,b]}Y), X \star (\mathcal{N}_{[a]}Y)\} \cup \\ &\quad \{x \star Y | x \in tio(X)\} \cup \{X \star y | y \in tio(Y)\} && \star \in \{\vee, \wedge, \rightarrow\} \end{aligned}$$

Example where the interval limits for the new temporal operators are $a = 2$ and $b = 3$.

$$\begin{aligned} tio(\phi) = \{&\mathcal{F}_{[2,3]}(a\mathcal{U}_{[0,1]}b) \wedge \mathcal{F}_{[1,2]}(x > 3), \\ &\mathcal{G}_{[2,3]}(a\mathcal{U}_{[0,1]}b) \wedge \mathcal{F}_{[1,2]}(x > 3), \\ &\mathcal{N}_{[2]}(a\mathcal{U}_{[0,1]}b) \wedge \mathcal{F}_{[1,2]}(x > 3), \\ &a\mathcal{U}_{[0,1]}b \wedge \mathcal{F}_{[2,3]}(\mathcal{F}_{[1,2]}(x > 3)), \\ &a\mathcal{U}_{[0,1]}b \wedge \mathcal{G}_{[2,3]}(\mathcal{F}_{[1,2]}(x > 3)), \\ &a\mathcal{U}_{[0,1]}b \wedge \mathcal{N}_{[2]}(\mathcal{F}_{[1,2]}(x > 3))\} \end{aligned}$$

3.1.12 Associate Shift Operator

For every nested (temporal) logical expression with two operators in a given formula one mutated formula is created that holds the same expression but with different associativity of the two operators.

$$
\begin{aligned}
aso(p) &= \{p\} \\
aso(\star\ X) &= \{\star\ x | x \in aso(X)\} && \star \in \{\neg, \mathcal{F}_{[a,b]}, \mathcal{G}_{[a,b]}, \mathcal{N}_{[a]}\} \\
aso((X \star_1 Y) \star_2 Z) &= \{X \star_1 (Y \star_2 Z)\} \\
&\quad \cup \{(x \star_1 Y) \star_2 Z) | x \in aso(X)\} \\
&\quad \cup \{(X \star_1 y) \star_2 Z) | y \in aso(Y)\} \\
&\quad \cup \{(X \star_1 Y) \star_2 z) | z \in aso(Z)\} && \star_1, \star_2 \in \{\vee, \wedge, \rightarrow, \mathcal{U}_{[a,b]}, \mathcal{R}_{[a,b]}\} \\
aso(X \star_1 (Y \star_2 Z)) &= \{(X \star_1 Y) \star_2 Z\} \\
&\quad \cup \{(x \star_1 Y) \star_2 Z) | x \in aso(X)\} \\
&\quad \cup \{(X \star_1 y) \star_2 Z) | y \in aso(Y)\} \\
&\quad \cup \{(X \star_1 Y) \star_2 z) | z \in aso(Z)\} && \star_1, \star_2 \in \{\vee, \wedge, \rightarrow, \mathcal{U}_{[a,b]}, \mathcal{R}_{[a,b]}\}
\end{aligned}
$$

Example:

$$aso(\phi) = \{a\mathcal{U}_{[0,1]}(b \wedge \mathcal{F}_{[1,2]}(x > 3))\}$$

The mutation operators consist of different mutation rules. Applying a mutation rule once to a formula produces one mutant. Hence a mutation operator produces a set of mutants. One mutation rule is not applied more than once, hence we say the mutants are of first order. A question that arises in this context is, whether it is enough to use first order mutants for the test generation. It will often be the case that a wrong specification holds more than one error compared to the correct specification. Therefore it is not clear whether the tests that are generated using first order mutants can also distinguish a formula with multiple errors, a so-called higher order mutant, from the correct formalization. We do not have a proof certify that this is the case but we would like to refer the reader to Section 6.1. In Section 6.1 we discuss the evaluation of STLInspector that was also done on higher order mutants amongst others formulae. The evaluation therefore suggests that validation tests obtained through first order mutants also work on formulae with more than one error.

3.2 Unique First Cause

The next coverage criterion checks how single clauses of a formula independently affect the formula's satisfiability. A clause in this context is simply an individual component of a formula. We look at each clause separately to see how changing it alters the outcome of the formula it is a part of. The focus is laid on detecting unique first causes of a formula.

Definition 19. *Given a formula and a signal for it, we define a clause as the unique first cause if in the first state along the signal where the formula is (dis-)satisfied, it is (dis-)satisfied because of the specific clause [8].*

For a formula ϕ let ϕ^+ denote the set of test predicates to generate test signals that expose each clause of ϕ as a unique first cause satisfiying ϕ, and ϕ^- denote the set of test predicates for signals exposing the clauses as unique first causes dissatisfying ϕ. With p being an atomic proposition and X and Y being subformulae of a given formula ϕ, we can define the rules below to derive ϕ^+ and ϕ^-. All rules are based on work by Whalen et al. [9] and Fraser et al. [8] but were adapted to fit STL operators in this thesis:

$$
\begin{aligned}
p^+ &= \{p\} \\
p^- &= \{\neg p\} \\
(X \wedge Y)^{+/-} &= \{x \wedge Y | x \in X^{+/-}\} \cup \{X \wedge y | y \in Y^{+/-}\} \\
(X \vee Y)^{+/-} &= \{x \wedge \neg Y | x \in X^{+/-}\} \cup \{\neg X \wedge y | y \in Y^{+/-}\} \\
(\neg X)^{+/-} &= X^{-/+} \\
\mathcal{G}_{[a,b]} X^+ &= \{X \mathcal{U}_{[a,b]} (x \wedge \mathcal{G}_{[a,b]} X) | x \in X^+\} \\
\mathcal{G}_{[a,b]} X^- &= \{X \mathcal{U}_{[a,b]} x | x \in X^-\} \\
\mathcal{F}_{[a,b]} X^+ &= \{\neg X \mathcal{U}_{[a,b]} x | x \in X^+\} \\
\mathcal{F}_{[a,b]} X^- &= \{\neg X \mathcal{U}_{[a,b]} (x \wedge \mathcal{G}_{[a,b]} \neg X) | x \in X^-\} \\
\mathcal{N}_{[a]} X^{+/-} &= \{\mathcal{N}_{[a]} x | x \in X^{+/-}\} \\
(X \mathcal{U}_{[a,b]} Y)^+ &= \{(X \wedge \neg Y) \mathcal{U}_{[a,b]} ((x \wedge \neg Y) \wedge (X \mathcal{U}_{[a,b]} Y)) | x \in X^+\} \\
&\quad \cup \{(X \wedge \neg Y) \mathcal{U}_{[a,b]} y | y \in Y^+\} \\
(X \mathcal{U}_{[a,b]} Y)^- &= \{(X \wedge \neg Y) \mathcal{U}_{[a,b]} (x \wedge \neg Y) | x \in X^-\} \\
&\quad \cup \{(X \wedge \neg Y) \mathcal{U}_{[a,b]} (y \wedge \neg (X \mathcal{U}_{[a,b]} Y)) | y \in Y^-\}
\end{aligned}
$$

We now demonstrate the rules on the same example formula as in the previous section but with only Boolean atomic propositions for the sake of simplicity:

$$\phi = a \mathcal{U}_{[0,1]} b \wedge \mathcal{F}_{[1,2]} c$$

We first apply the rules on all subformulae and can then deduce the set of test predicates for (dis-)satisfying unique first causes of the whole formula.

$$
\begin{aligned}
(a)^+ &= \{a\}, (b)^+ = \{b\}, (c)^+ = \{c\} \\
(a)^- &= \{\neg a\}, (b)^- = \{\neg b\}, (c)^- = \{\neg c\} \\
(a \mathcal{U}_{[0,1]} b)^+ &= \{(a \wedge \neg b) \mathcal{U}_{[0,1]} ((x \wedge \neg b) \wedge (a \mathcal{U}_{[0,1]} b)) | x \in (a)^+\} \\
&\quad \cup \{(a \wedge \neg b) \mathcal{U}_{[0,1]} y | y \in (b)^+\}
\end{aligned}
$$

$$
\begin{aligned}
&= \{((a \wedge \neg b)\mathcal{U}_{[0,1]}((a \wedge \neg b) \wedge (a\mathcal{U}_{[0,1]}b))), ((a \wedge \neg b)\mathcal{U}_{[0,1]}b)\} \\
(a\mathcal{U}_{[0,1]}b)^{-} &= \{(a \wedge \neg b)\mathcal{U}_{[0,1]}(x \wedge \neg b) | x \in (a)^{-}\} \\
&\quad \cup \{(a \wedge \neg b)\mathcal{U}_{[0,1]}(y \wedge \neg(a\mathcal{U}_{[a,b]}b)) | y \in (b)^{-}\} \\
&= \{((a \wedge \neg b)\mathcal{U}_{[0,1]}(\neg a \wedge \neg b)), (a \wedge \neg b)\mathcal{U}_{[0,1]}(\neg b \wedge \neg(a\mathcal{U}_{[a,b]}b)))\} \\
(\mathcal{F}_{[1,2]}c)^{+} &= \{\neg c\mathcal{U}_{[1,2]}x | x \in (c)^{+}\} \\
&= \{(\neg c\mathcal{U}_{[1,2]}c)\} \\
\mathcal{F}_{[1,2]}c^{-} &= \{\neg c\mathcal{U}_{[1,2]}(x \wedge \mathcal{G}_{[1,2]}\neg c) | x \in (c)^{-}\} \\
&= \{(\neg c\mathcal{U}_{[1,2]}(\neg c \wedge \mathcal{G}_{[1,2]}\neg c))\} \\
&= \{(\mathcal{G}_{[0,3]}\neg c)\} \\
(\phi)^{+/-} &= \{x \wedge \mathcal{F}_{[1,2]}c | x \in (a\mathcal{U}_{[0,1]}b)^{+/-}\} \\
&\quad \cup \{(a\mathcal{U}_{[0,1]}b) \wedge y | y \in (\mathcal{F}_{[1,2]}c)^{+/-}\} \\
&\Rightarrow \\
(\phi)^{+} &= \{((a \wedge \neg b)\mathcal{U}_{[0,1]}((a \wedge \neg b) \wedge (a\mathcal{U}_{[0,1]}b))) \wedge \mathcal{F}_{[1,2]}c, \\
&\quad ((a \wedge \neg b)\mathcal{U}_{[0,1]}b) \wedge \mathcal{F}_{[1,2]}c, \\
&\quad ((a\mathcal{U}_{[0,1]}b) \wedge (\neg c\mathcal{U}_{[1,2]}c))\} \\
(\phi)^{-} &= \{(((a \wedge \neg b)\mathcal{U}_{[0,1]}(\neg a \wedge \neg b)) \wedge \mathcal{F}_{[1,2]}c), \\
&\quad (((a \wedge \neg b)\mathcal{U}_{[0,1]}(\neg b \wedge \neg(a\mathcal{U}_{[a,b]}b))) \wedge \mathcal{F}_{[1,2]}c), \\
&\quad ((a\mathcal{U}_{[0,1]}b) \wedge \mathcal{G}_{[0,3]}\neg c)\}
\end{aligned}
$$

Tests that are generated using the predicates derived from the unique first cause coverage criterion expose clauses as unique first causes. The user should correctly classify these clauses into satisfying and dissatisfying unique first causes.

3.3 Property Inactive Clause

The Property Inactive Clause criterion [9][8] constitutes test predicates for a given formula that identify clauses that can be changed without altering the formula's outcome. The idea behind this coverage criterion is that it might be important to know what modifications of a formula go unnoticed. For example if changing a safety critical clause does not influence the formula's behavior, this would be crucial to know. We will now state rules to derive test cases for the Property Inactive Clause criterion. The rules are inspired by [9][8] and adjusted for STL operators in this thesis. Let p be an an atomic proposition, X and Y subformulae of a given formula.

$$
\begin{aligned}
\rho(p) &= \{p, \neg p\} \\
\rho(X \wedge Y) &= \{x \wedge \neg Y | x \in \rho(X)\} \cup \{\neg X \wedge y | y \in \rho(Y)\}
\end{aligned}
$$

$$
\begin{aligned}
\rho(X \vee Y) &= \{x \wedge Y | x \in \rho(X)\} \cup \{X \wedge y | y \in \rho(Y)\} \\
\rho(\neg X) &= \rho(X) \\
\rho(\mathcal{G}_{[a,b]} X) &= \{X \mathcal{U}_{[a,b]} (x \wedge \mathcal{G}_{[a,b]} X) | x \in \rho(X)\} \\
\rho(\mathcal{F}_{[a,b]} X) &= \{\neg X \mathcal{U}_{[a,b]} (x \wedge (X \vee \mathcal{F}_{[a,b]} X)) | x \in \rho(X)\} \\
\rho(\mathcal{N}_{[a]} X) &= \{\mathcal{N}_{[a]} x | x \in \rho(X)\} \\
\rho(X \mathcal{U}_{[a,b]} Y) &= \{(X \wedge \neg Y) \mathcal{U}_{[a,b]} (x \wedge (Y \vee (X \mathcal{U}_{[a,b]} Y))) | x \in \rho(X)\} \\
&\quad \cup \{(X \wedge \neg Y) \mathcal{U}_{[a,b]} (y \wedge (Y \vee (X \mathcal{U}_{[a,b]} Y))) | y \in \rho(Y)\}
\end{aligned}
$$

We will now illustrate the rules for this criterion using again the example formulae

$$\phi = a \mathcal{U}_{[0,1]} b \wedge \mathcal{F}_{[1,2]} c$$

The following test predicates are produced using the rules from above:

$$
\begin{aligned}
\rho(a) &= \{a, \neg a\}, \rho(b) = \{b, \neg b\}, \rho(c) = \{c, \neg c\} \\
\rho(\mathcal{F}_{[1,2]} c) &= \{\neg c \mathcal{U}_{[1,2]} (x \wedge (c \vee \mathcal{F}_{[1,2]} c)) | x \in \rho(c)\} \\
&= \{(\neg c \mathcal{U}_{[1,2]} (c \wedge (c \vee \mathcal{F}_{[1,2]} c))), (\neg c \mathcal{U}_{[1,2]} (\neg c \wedge (c \vee \mathcal{F}_{[1,2]} c)))\} \\
&= \{(\neg c \mathcal{U}_{[1,2]} c), (\neg c \mathcal{U}_{[1,2]} (\neg c \wedge \mathcal{F}_{[1,2]} c))\} \\
\rho(a \mathcal{U}_{[0,1]} b) &= \{(a \wedge \neg b) \mathcal{U}_{[0,1]} (x \wedge (b \vee (a \mathcal{U}_{[0,1]} b))) | x \in \rho(a)\} \cup \\
&\quad \{(a \wedge \neg b) \mathcal{U}_{[0,1]} (y \wedge (b \vee (a \mathcal{U}_{[0,1]} b))) | y \in \rho(b)\} \\
&= \{((a \wedge \neg b) \mathcal{U}_{[0,1]} (a \wedge (b \vee (a \mathcal{U}_{[0,1]} b)))), \\
&\quad ((a \wedge \neg b) \mathcal{U}_{[0,1]} (\neg a \wedge (b \vee (a \mathcal{U}_{[0,1]} b)))), \\
&\quad ((a \wedge \neg b) \mathcal{U}_{[0,1]} (b \wedge (b \vee (a \mathcal{U}_{[0,1]} b)))), \\
&\quad ((a \wedge \neg b) \mathcal{U}_{[0,1]} (\neg b \wedge (b \vee (a \mathcal{U}_{[0,1]} b))))\} \\
&= \{((a \wedge \neg b) \mathcal{U}_{[0,1]} (a \wedge (b \vee (a \mathcal{U}_{[0,1]} b)))), \\
&\quad ((a \wedge \neg b) \mathcal{U}_{[0,1]} (\neg a \wedge (b \vee (a \mathcal{U}_{[0,1]} b)))), \\
&\quad ((a \wedge \neg b) \mathcal{U}_{[0,1]} b), \\
&\quad ((a \wedge \neg b) \mathcal{U}_{[0,1]} (\neg b \wedge (a \mathcal{U}_{[0,1]} b)))\} \\
&\Rightarrow \\
\rho(\phi) &= \{x \wedge \neg b | x \in \rho(a)\} \cup \{\neg a \wedge y | y \in \rho(b)\} \\
&= \{(((a \wedge \neg b) \mathcal{U}_{[0,1]} (a \wedge (b \vee (a \mathcal{U}_{[0,1]} b)))) \wedge \neg \mathcal{F}_{[1,2]} c), \\
&\quad (((a \wedge \neg b) \mathcal{U}_{[0,1]} (\neg a \wedge (b \vee (a \mathcal{U}_{[0,1]} b)))) \wedge \neg \mathcal{F}_{[1,2]} c), \\
&\quad (((a \wedge \neg b) \mathcal{U}_{[0,1]} b) \wedge \neg \mathcal{F}_{[1,2]} c), \\
&\quad (((a \wedge \neg b) \mathcal{U}_{[0,1]} (\neg b \wedge (a \mathcal{U}_{[0,1]} b))) \wedge \neg \mathcal{F}_{[1,2]} c), \\
&\quad (\neg (a \mathcal{U}_{[0,1]} b) \wedge (\neg c \mathcal{U}_{[1,2]} c)), \\
&\quad (\neg (a \mathcal{U}_{[0,1]} b) \wedge (\neg c \mathcal{U}_{[1,2]} (\neg c \wedge \mathcal{F}_{[1,2]} c)))\}
\end{aligned}
$$

The test predicates derived from the coverage criteria that were described above are formulae that represent properties we want to test for. Since they are systematically generated, resulting in a set of test predicates that satisfies the chosen coverage criterion, the test suite that is generated using the set of test predicates also fulfills the coverage criterion. Hence, we can deviate certain guarantees if the complete test suite is successfully run on the specification. Together with the original formula, the test predicates are input for the test generation that we will now proceed to.

4 Test Generation

In order to validate a given formula we need to generate tests for it. Input for the generation of a single test are the original formula ϕ and a test predicate ψ that is derived by means of a coverage criterion as described in Section 3. The tests are used to distinguish between the original formula and the test predicates. An example of such a distinction is described in the beginning of Section 3.1. There are two categories of tests (positive and negative ones) that are presented to the engineer who then decides whether the tests have been categorized in the right way.

Definition 20. *Given a formula ϕ and a test predicate ψ of ϕ. Let $\mathcal{F}$ be the combination $\phi \wedge \neg\psi$, i.e. $\mathcal{F} := \phi \wedge \neg\psi$. We call a test T a positive test if $T \models \mathcal{F}$.*

Definition 21. *Given a formula ϕ and a test predicate ψ of ϕ. Let $\mathcal{F}$ be the combination $\neg\phi \wedge \psi$, i.e. $\mathcal{F} := \neg\phi \wedge \psi$. We call a test T a negative test if $T \models \mathcal{F}$.*

A positive test is a test that satisfies the original formula. A negative test dissatisfies the original formula. If the satisfaction value of the formula differs from the satisfaction value of the test regarding the requirement, then we can infer that the formula differs from the requirement or, in other words, that the requirement was not formalized correctly. In summary there are four cases to be formulated given a test T that is generated for a proposed formalization ϕ using the test predicate ψ. If the user categorizes T as conforming to the requirement, we write $\uparrow$. If the user finds T not in conformance with the requirement, we write $\downarrow$.

1. T positive test, $\uparrow$: error represented by ψ is not present in ϕ.
2. T positive test, $\downarrow$: ϕ contains illegitimate behavior.
3. T negative test, $\uparrow$: ψ contains legitimate behavior that is missing in ϕ.
4. T negative test, $\downarrow$: error represented by ψ is not present in ϕ. [2]

Test predicates of a formula ϕ describe properties that are similar to properties of the original formula but slightly changed. Each test allows us to distinguish between the actual formula and the predicate that was used to generate the test. With a positive test we can show the absence of a property held by a predicate of ϕ. This is useful when we want, for example, to exclude errors described by test predicates obtained by property mutation such as unintended toggling of a relational expression (writing $\geq$ instead of $>$ for instance). With a negative test we can demonstrate that ϕ lacks a property that the predicate has. For example, we might detect that an atomic

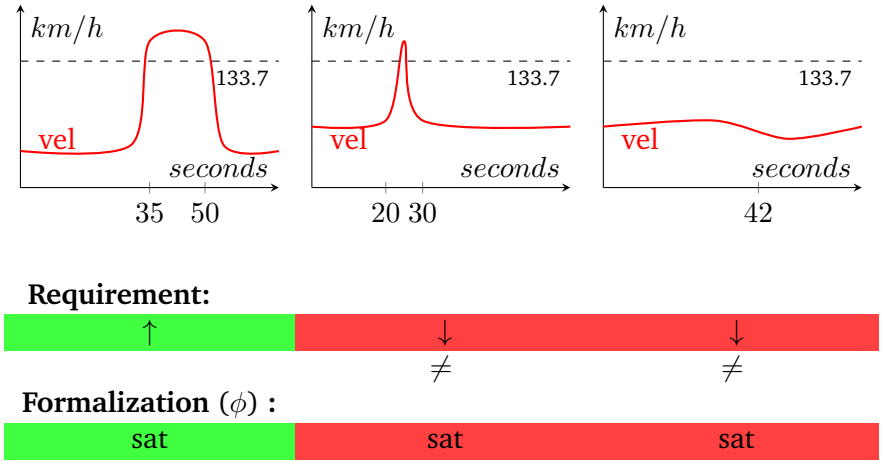

Fig. 4.1: Categorization shows that the formalization is not correct for the requirement

proposition of ϕ needs to hold globally as described by mutant ψ instead of becoming true only once (ψ replaces $\mathcal{F}$ by $\mathcal{G}$). Let us again take a look at the example from the introduction. The following requirement

At some point of time in the first 42 seconds the vehicle speed (vel) will rise above 133.7 km/h and from that point on stay above 133.7 km/h for at least 11 seconds.

is supposed to be specified by the STL formula

$$\phi = \mathcal{F}_{[0,42]}((vel > 133.7) \Rightarrow (\mathcal{G}_{[0,11]}(vel > 133.7)))$$

Assuming the three tests depicted in Figure 4.1 were generated and classified by an engineer. The table in Figure 4.1 then shows the satisfaction results for the formalization ϕ in comparison to the classification of the tests. Since for the last two tests the two values differ, we can deduce that ϕ is not the correct formalization for the given requirement. (The correct formalization would be $\mathcal{F}_{[0,42]}(\mathcal{G}_{[0,11]}(vel > 133.7))$.)

Now that we fully understand how positive and negative tests are used, the question arises how the tests are generated. The test generation procedure is composed of 4 steps that are portrayed in Figure 4.2. Figure 4.2 shows the generation of a positive test only (F would have to be $\neg\phi \wedge \psi$ in order to generate a negative test). We start with the combination F for which we obtain a satisfying assignment. This

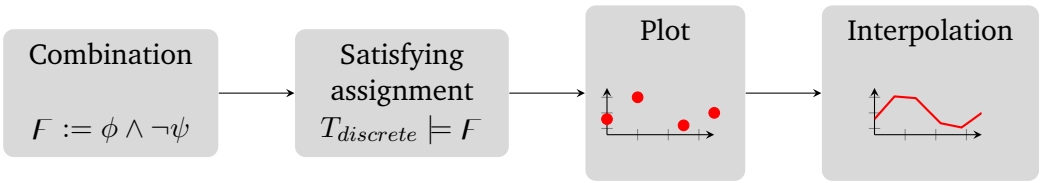

Fig. 4.2: Generation of a positive test for ϕ using test predicate ψ

assignment can be viewed as a set of sampled values for the atomic propositions of ϕ over the course of time. We call the assignment a discrete test and define with AP as the set of all atomic propositions:

Definition 22. *A discrete test $T_{discrete}$ is a mapping $(AP \times \mathbb{R}_0^+)^n \rightarrow (\mathbb{R} \cup \mathbb{B})^n$ of atomic propositions at different points of time to either real values, if the atomic proposition is of real format, or Boolean values otherwise.*

Then we can plot the sampled values of the discrete test as points which are finally connected to a continuous test signal using linear interpolation. It is important to keep in mind that tests are defined as *continuous* signals.

The problem we will address in the following section is how to obtain a discrete test $T_{discrete}$ from the combination $\mathcal{F}$. Specifically, how to receive a satisfying assignment for an STL formula.

4.1 SMT-based Temporal Logic Encoding

We use a satisfiability modulo theories solver (SMT-Solver) to generate satisfying assignments for a given logical formula. However, an SMT-Solver does not accept temporal logic formulae as input. Thus, the semantics of an STL formula is encoded by an SMT formula, precisely a formula in the existential fragment of first order theory of linear real arithmetic called Quantifier-Free Linear Real Arithmetic or simply QFLRA. Additionally, the transformation of an STL formula into a discrete QFLRA formula has to be achieved without loosing information about the continuity of the STL formula. To be explicit:

For a given STL formula $\mathcal{F}$ our goal is to find a QFLRA representation that, when used as input for an SMT Solver, results in a discrete test such that linear interpolation using this discrete test results in a continuous signal that satisfies $\mathcal{F}$.

Since temporal logic formulae represent sets of signals, the basic idea is to formulate a disjunction of a subset of possible signals when converting an STL formula to QFLRA. A subset is enough since we want to compute only a single signal that satisfies the formula and this single signal is internally represented by one single satisfying assignment. Let us introduce the following objects and some of their characteristics which will be used to encode a given STL formula into a valid QFLRA representation of the formula. The encoding is based on a similar Linear Temporal Logic (LTL) encoding defined in [10].

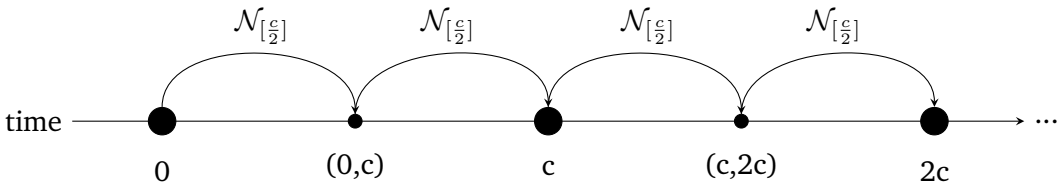

Fig. 4.3: The use of time step c

4.1.1 Time Discretization

Our goal is to receive a discrete QFLRA representation of a continuous STL formula $\mathcal{F}$. In order to achieve a discretization, we first partition time in an alternating sequence of points and open intervals [1]. We use a time step $c \in \mathbb{R}_0$ that corresponds to the length of one point of time $\{kc\}$ to the next point of time $\{(k+1)c\}$ or, respectively, from one open interval $(kc, (k+1)c)$ to the next open interval $((k+1)c, (k+2)c)$. Figure 4.3 depicts this scenario. A restriction on our time step c is that it has to divide all interval bounds of the temporal operators in $\mathcal{F}$. This is due to the fact that we want to partition all intervals of the temporal operators into fragments of size c. Hence, we can rewrite all intervals $[a, b]$ with $a = kc$, $b = k'c$ and $k' - k = k''$ as follows:

$$[a,b] = \{a\} \cup (a, a+c) \cup \{a+c\} \cup (a+c, 2\cdot(a+c)) \cup \cdots \cup ((k''-1)\cdot(a+c), b) \cup \{b\}$$

If we start at point of time 0 with a time step c then kc denotes a point of time and $k\frac{c}{2}$ an open interval. Therefore, the discrete test for an STL formula $\mathcal{F}$ we want to finally obtain will consist of values for the atomic proposition at every $\frac{c}{2}$ sampled time points. The encoding of $\mathcal{F}$ in the next section is based on dividing the STL formula into parts of length c to be able to sample every $\frac{c}{2}$ value.

4.1.2 Transformation into Quantifier-Free Linear Real Arithmetic

In this subsection we will introduce multiple formats that are used in intermediate steps that transform an STL formula into a QFLRA formula. Using a fixed time step c we start by establishing the following definition and propositions:

Definition 23. *An STL formula Φ is said to be in next normal form if the syntax of Φ consists of only the temporal operator Next, where Next has the form $\mathcal{N}_{[k\frac{c}{2}]}$ with $k \in \mathbb{N}_0$), the propositional operators disjunction ($\vee$), conjunction ($\wedge$), negation ($\neg$) and atomic propositions.*

Proposition 1. *Every atomic proposition is in next normal form.*

Proposition 2. *If the two STL formulae Φ and Ψ are in next normal form, then their disjunction $\Phi \vee \Psi$ is also in next normal form.*

Proposition 3. *If the two STL formulae Φ and Ψ are in next normal form, then their conjunction $\Phi \wedge \Psi$ is also in next normal form.*

Proposition 4. *If the STL formula Φ is in next normal form, then its negation $\neg\Phi$ is also in next normal form.*

Proposition 5. *If the STL formula Φ is in next normal form, then the STL formula $\mathcal{N}_{[k\frac{c}{2}]}(\Phi)$ with $k \in \mathbb{N}_0$ is also in next normal form.*

These propositions follow directly from the definition of the next normal form. We proceed with the following observations:

Proposition 6. *If the two STL formulae Φ and Ψ are in next normal form, then the STL formula $\Phi\, \mathcal{U}_{[0,0]}\, \Psi$ can also be converted into next normal form with*

$$\Phi\, \mathcal{U}_{[0,0]}\, \Psi \;\equiv\; \Psi$$

Proof. Since Ψ is supposed to be in next normal form the proposition is vacuously true. □

Proposition 7. *If the two STL formulae Φ and Ψ are in next normal form, then the STL formula $\Theta := \Phi\, \mathcal{U}_{[0,a]}\, \Psi$ with $a \neq 0$ can be converted into next normal form with*

$$\Theta \equiv \Psi \vee (\Phi \wedge \mathcal{N}_{[\frac{c}{2}]}\Phi \wedge \mathcal{N}_{[\frac{c}{2}]}\Psi) \vee (\Phi \wedge \mathcal{N}_{[\frac{c}{2}]}\Phi \wedge \mathcal{N}_{[c]}(\Phi \mathcal{U}_{[0,a-c]}\Psi))$$

if Θ is evaluated at a point of time and

$$\Theta \equiv \Phi \vee (\Phi \wedge \mathcal{N}_{[\frac{c}{2}]}\Psi) \vee (\Phi \wedge \mathcal{N}_{[\frac{c}{2}]}\Phi \wedge \mathcal{N}_{[c]}\Phi \wedge \mathcal{N}_{[c]}(\Phi \mathcal{U}_{[0,a-c]}\Psi))$$

if Θ is evaluated in an open interval.

Proof. The basic idea is to apply a transformation of recursively rewriting the formula in intervals of length c using only Next operators. This is achieved by disjoining all the different possibilities how the atomic proposition can hold at the sampled points 0, $\frac{c}{2}$ and c, where $\frac{c}{2}$ is the representative sampling point for the open interval $(0, c)$. We have to differentiate whether the formula Θ holds starting at a point of time or in an open interval, i.e. whether we jump with the recursion from a point of time to the next point or from an open interval to the next interval.

When we start at a point of time, the following possibilities of the two operands Φ and Ψ holding exist: Firstly, Ψ can hold right at the beginning, thus at point of time 0. This results in the term Ψ in the disjunction of all possibilities. Secondly, it can be the case that Φ holds in the interval $[0, t)$ from 0 to some point of time $t \in (0, c)$, at which Ψ holds. Then we receive the disjunction's term $\Phi \wedge \mathcal{N}_{[\frac{c}{2}]}\Phi \wedge \mathcal{N}_{[\frac{c}{2}]}\Psi$, where $\frac{c}{2}$ indicates the evaluation in the open interval $(0, c)$. Both Φ and Ψ must hold in

this interval due to the fact that with our chosen sampling, we cannot determine the exact point of time $t \in (0, c)$ where Ψ first becomes true. And thirdly, Ψ can hold exactly at or sometime after point of time c, which leaves Φ holding everywhere in the interval $[0, c)$. This gets us the term $\Phi \wedge \mathcal{N}_{[\frac{c}{2}]}\Phi \wedge \mathcal{N}_{[c]}(\Phi \mathcal{U}_{[0,a-c]}\Psi)$. Summarized we receive the decomposition over a point of time as stated in the proposition:

$$\Theta \equiv \Psi \vee (\Phi \wedge \mathcal{N}_{[\frac{c}{2}]}\Phi \wedge \mathcal{N}_{[\frac{c}{2}]}\Psi) \vee (\Phi \wedge \mathcal{N}_{[\frac{c}{2}]}\Phi \wedge \mathcal{N}_{[c]}(\Phi \mathcal{U}_{[0,a-c]}\Psi))$$

When we take a look at the last term of this disjunction, we see that the decomposition can be applied recursively. As the interval length decreases in every step of the recursion, we eventually obtain a formula completely in next normal form.

When we start during an open interval, the first possibility is that Ψ holds right from the beginning resulting in term Ψ of the disjunction just as above. However, the next possibility differs from the expansion above: Φ can hold in the open interval we started at and at the first point in time we encounter, Ψ holds. This is described by the term $\Phi \wedge \mathcal{N}_{[\frac{c}{2}]}\Psi$ of the disjunction. Please note that in this case $\mathcal{N}_{[\frac{c}{2}]}\Psi$ holds at a point of time since we started in an open interval. And finally, it can be the case that Ψ first holds at or sometime after the interval of length c we are expanding in the recursion step. This scenario is described by $\Phi \wedge \mathcal{N}_{[\frac{c}{2}]}\Phi \wedge \mathcal{N}_{[c]}\Phi \wedge \mathcal{N}_{[c]}(\Phi \mathcal{U}_{[0,a-c]}\Psi)$. In summary the decomposition over an open interval is as described in the proposition:

$$\Theta \equiv \Phi \vee (\Phi \wedge \mathcal{N}_{[\frac{c}{2}]}\Psi) \vee (\Phi \wedge \mathcal{N}_{[\frac{c}{2}]}\Phi \wedge \mathcal{N}_{[c]}\Phi \wedge \mathcal{N}_{[c]}(\Phi \mathcal{U}_{[0,a-c]}\Psi))$$

Also in this case the recursion reduces the length of the interval of Until and we hence ultimately receive a formula completely in next normal form. □

Proposition 8. *If the two STL formulae Φ and Ψ are in next normal form, then the STL formula $\Theta := \Phi \, \mathcal{U}_{[a,b]} \, \Psi$ with $0 < a \leq b$ can be converted into next normal form with*

$$\Theta \equiv \Phi \wedge \mathcal{N}_{[\frac{c}{2}]}\Phi \wedge \mathcal{N}_{[c]}(\Phi \mathcal{U}_{[a-c,b-c]}\Psi)$$

if Θ is evaluated at a point of time and

$$\Theta \equiv \Phi \wedge \mathcal{N}_{[\frac{c}{2}]}\Phi \wedge \mathcal{N}_{[c]}\Phi \wedge \mathcal{N}_{[c]}(\Phi \mathcal{U}_{[a-c,b-c]}\Psi)$$

if Θ is evaluated in an open interval.

Proof. As in the proof of the previous proposition we distinguish between the formula holding at a point of time or in an open interval. In both cases we want to ensure that Φ holds all the time up to the first interval bound a.

In the first case, this is achieved by the first two terms of the decompositions since they determine that Φ holds at every point of time up to a and in every open interval in between them. Afterwards, the recursion is applied until the first interval bound

has been reduced to 0. Then one of the previous propositions can be applied to finish the transformation into next normal form. Hence, we receive the following decomposition in total for the evaluation at a point of time:

$$\Phi \wedge \mathcal{N}_{[\frac{c}{2}]}\Phi \wedge \mathcal{N}_{[c]}(\Phi \mathcal{U}_{[a-c,b-c]}\Psi)$$

In the second case, the we start in an open interval, thus the sampling point $\frac{c}{2}$ lies at the point of time just outside of this interval. At this point of time Φ needs to hold to ensure that Φ holds at every point of time up to $a > \frac{c}{2} > 0$. The open interval after the point of time $\frac{c}{2}$ is at the sampling point c. In this interval we have to make sure that Φ also holds until possibly Ψ becomes true. Therefore we add the term $\mathcal{N}_{[c]}\Phi$ additionally to $\Phi \mathcal{U}_{[a-c,b-c]}\Psi$. Thus the decomposition is as stated in the proposition:

$$\Phi \wedge \mathcal{N}_{[\frac{c}{2}]}\Phi \wedge \mathcal{N}_{[c]}\Phi \wedge \mathcal{N}_{[c]}(\Phi \mathcal{U}_{[a-c,b-c]}\Psi)$$

□

Proposition 9. *Every STL formula Φ can be converted into next normal form.*

Proof. The formula Φ can be rewritten such that it contains only $\mathcal{N}_{[a]}$ and $\mathcal{U}_{[a,b]}$ with $a, b \in \mathbb{R}_0^+$ as temporal operators next to logical operators. This is due to the fact that we defined $\mathcal{F}_{[a,b]}$, $\mathcal{G}_{[a,b]}$ and $\mathcal{R}_{[a,b]}$ using the Until-operator (see Section 2.1). Now we can apply the propositions from above on all Until-operators and thereby receive a next normal form. □

Proposition 10. *If Φ is an STL formula in next normal form, then Φ can be converted into an equivalent QFLRA formula with*

$$E(\Phi, t) = \begin{cases} p_t & , \Phi = p \in AP \\ \neg E(\psi, t) & , \Phi = \neg\psi \\ E(\psi_1, t) \wedge E(\psi_2, t) & , \Phi = \psi_1 \wedge \psi_2 \\ E(\psi_1, t) \vee E(\psi_2, t) & , \Phi = \psi_1 \vee \psi_2 \\ E(\psi_1, t) \rightarrow E(\psi_2, t) & , \Phi = \psi_1 \rightarrow \psi_2 \\ E(\psi, t + a) & , \Phi = N_{[a]}\psi \end{cases}$$

where t equals zero when E is first applied and

$$p_t = \begin{cases} x_t & , p = x \in \chi^B \\ c^T \cdot x_t \sim b & , p = c^T \cdot x \sim b, x \in \chi^R \end{cases}$$

Proof. Since Φ is already in next normal form, it suffices to eliminate the Next operators to receive an equivalent QFLRA formula. For this, the function $E(\phi, t)$ is recursively called on the operand(s) of an operator until the recursion reaches an

atomic proposition $p \in AP$ where AP denotes the set of all atomic propositions. Then the function, specifically the definition of p_t, translates each atomic proposition at a sampling time point t into a QFLRA formula by annotating its variable x with an t as an index. By applying $E(\phi, t)$ with $t = 0$ in the beginning, we determine that the indices start at point of time 0. The logical connectives do not result in a shift in time, therefore t is not changed when applying the function on their operands. However, the Next operator denotes a shift in time and we therefore have to ensure that atomic propositions that are part of the operand of a Next operator are indexed for a later time point. This is achieved by increasing t by the value of the operator's bound. □

The propositions stated in this subsection allow for a transformation of an STL formula $\mathcal{F}$ into a QFLRA formula. In order to produce a continuous test signal that satisfies the STL formula $\mathcal{F}$, the QFLRA representation of $\mathcal{F}$ has to be improved regarding continuity. We have to make sure that although we only assign values to our atomic propositions at discrete sampling points of time, the linear interpolation cannot insert faulty values in between the sampled points when computing the continuous signal.

4.1.3 Ensuring Continuity of the Encoding

Boolean atomic propositions are assumed to have constant values in the open intervals. However, for real atomic propositions it is necessary to encode propositions that hold in an open interval differently from the ones holding at time points. This is due to the fact that an atomic proposition p in an interval $(0, c)$ is represented by $\mathcal{N}_{[\frac{c}{2}]}$ in the next normal form, thus by the single sample point at $\frac{c}{2}$. This representation does not convey any information about the values of the real atomic proposition all throughout the interval but only about the value at at exactly $\frac{c}{2}$. Therefore we need to make sure that the proposition holds a satisfying value such that it has at $\frac{c}{2}$ for the whole interval $(0, c)$. Let us illustrate this using the following STL formula as an example:

$$\Phi \;=\; (vel > 42)\, \mathcal{U}_{[1,2]}\, (vel \leq 42)$$

The time step for this STL formula is $c = 1$ since this is the greatest divisor of both interval bounds. With $c = 1$ the next normal form for ϕ holds the term $(vel > 42) \wedge \mathcal{N}_{[0.5]}(vel > 42) \wedge \mathcal{N}_{[1]}(vel \leq 42)$ which describes the possibility where the velocity is greater than 42 all throughout the interval $[0, 1)$ and at point of of time 1 the velocity becomes less than or equal to 42. The term is transformed into the QFLRA formula $(vel_0 > 42) \wedge (vel_{0.5} > 42) \wedge (vel_1 \leq 42)$. A valid satisfying assignment produced for this QFLRA formula could be $[vel_0 = 44, vel_{0.5} = 44, vel_1 = 40]$. This would produce the test signal shown left in Figure 4.4 after linear interpolation. As

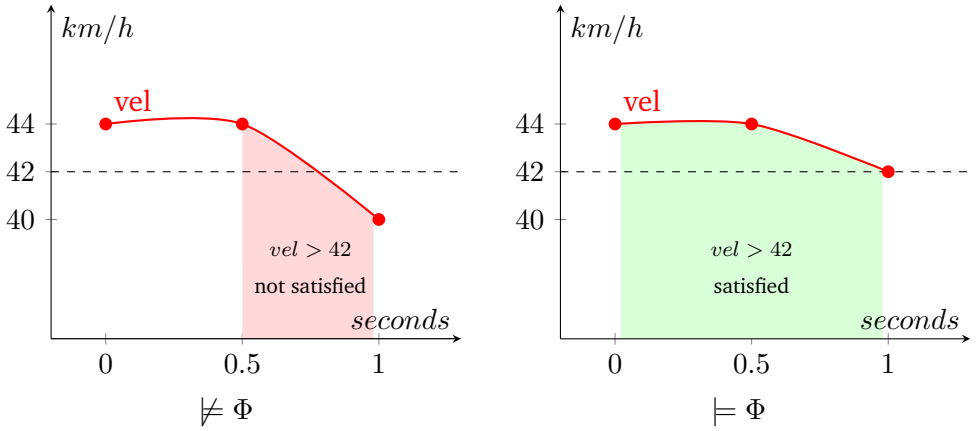

Fig. 4.4: Without special encoding in the open interval $(0, 1)$ for the atomic proposition $vel > 42$ the faulty test on the left is produced for formula $\Phi = (vel > 42)\, \mathcal{U}_{[1,2]}\, (vel \leq 42)$. With special encoding (see definition of $\overline{p}$) the produced test on the right conforms to Φ.

one can see, the test signal does not satisfy the original STL formula Φ because there is a time span in between 0 and 1, specifically in between 0.5 and 1, where the atomic proposition $(vel > 42)$ that should globally hold in the interval $[0, 1)$ is not fulfilled.

The problem lies within the sampling of points that represent open intervals. Let us disassemble the QFLRA term from above so we can analyze the three propositions more closely: For the two time points 0 and 1 the propositions $vel_0 > 42$ (the velocity has to be greater than 42 at point of time 0) and $vel_1 \leq 42$ (the velocity has to be equal or less than 42 at point of time 1) create no problem. However, the proposition $vel_{0.5} > 42$ should encode that the velocity is greater than 42 everywhere in the open interval $(0, 1)$. However, this is not ensured by choosing only 0.5 as a sampling time point. With our encoding we have to determine that an atomic proposition is equal to values at both time points right and left of the open interval such that the atomic proposition holds all throughout the open interval and, moreover, the produced signal is continuous at the interval bounds. In our case we have to choose a value at time point 1 that allows for the velocity to be above 42 during the open interval but also satisfies $vel \leq 42$. This is accomplished by adding a new atomic proposition to both time points that make up the bounds of the open interval. In this case we add $vel \geq 42$ to both time points 0 and 1 since this allows for continuity at the bounds of the interval and also for $vel > 42$ to be satisfied in the open interval. The new QFLRA formula is $(vel_0 > 42) \wedge (vel_0 > 42) \wedge (vel_{0.5} > 42) \wedge (vel \geq 42) \wedge (vel_1 \geq 42)$. This QFLRA representation of Φ produces a discrete test that after linear interpolation results in a test signal that conforms to Φ as shown in Figure 4.4 on the right.

The following definition demonstrates how continuity with regard to a real atomic proposition holding in an open interval is generally ensured:

Definition 24. *Let p be a real atomic proposition that is evaluated in the open interval $(0, c)$. Then we define $\overline{p}$ as*

$$\overline{p} = \overline{(c^T \cdot x \sim b)} = \begin{cases} p_{\frac{c}{2}}, & \sim \in \{\equiv\} \\ (c^T \cdot x \leq b)_0 \wedge p_{\frac{c}{2}} \wedge (c^T \cdot x \leq b)_c, & \sim \in \{<, \leq\} \\ (c^T \cdot x \geq b)_0 \wedge p_{\frac{c}{2}} \wedge (c^T \cdot x \geq b)_c, & \sim \in \{>, \geq\} \\ (c^T \cdot x \leq b)_0 \wedge p_{\frac{c}{2}} \wedge (c^T \cdot x \geq b)_c, & \sim \in \{\not\equiv\} \end{cases}$$

where the indices of the real atomic propositions denote the sample time points as used in the QFLRA encoding.

We require that when a formula is converted into next normal form the information about a real atomic proposition p that is evaluated in an open intervals is saved in the form of $\overline{p}$. The definition of $\overline{p}$ is not applied directly due to the following intermediate step that is based on encoding considerations regarding negation:

The conversion into next normal form can produce negated terms like $\neg\overline{p}$ for a real atomic proposition $p = vel > 42$ evaluated in some open interval. For the sake of simplicity we use the open interval $(0, c)$ for the following illustration: The negation of $vel > 42$ would be $vel \leq 42$ stating that the velocity should be less or equal to 42 in the open interval $(0, c)$. If we had added the clauses like in the definition of $\overline{p}$ before applying the negation, this would have resulted in

$$\begin{aligned} \neg\overline{p} &= \neg\overline{(vel > 42)} \\ &= \neg((vel \geq 42)_0 \wedge (vel > 42)_{\frac{c}{2}} \wedge (vel \geq 42)_c) \\ &= \neg(vel \geq 42)_0 \vee \neg(vel > 42)_{\frac{c}{2}} \vee \neg(vel \geq 42)_c \\ &= (vel < 42)_0 \vee (vel \leq 42)_{\frac{c}{2}} \vee (vel < 42)_c \end{aligned}$$

Due to De Morgan's laws the conjunctions in $\overline{p}$ are turned into disjunctions through the negation. The disjunction is already satisfied if only a single one of the terms at the sample points 0, $\frac{c}{2}$ or c holds. Therefore, the encoding would not ensure that $vel \leq 42$ holds everywhere in the open interval $(0, c)$ (and hence $vel > 42$ does not hold anywhere in $(0, c)$). If we first apply the negation and then insert the definition of $\overline{p}$ we receive the correct encoding:

$$\begin{aligned} \neg\overline{p} = &\neg\overline{(vel > 42)} \\ &\overline{(vel \leq 42)} \\ &(vel \leq 42)_0 \wedge (vel \leq 42)_{\frac{c}{2}} \wedge (vel \leq 42)_c \end{aligned}$$

By virtue of the considerations that were exemplified above, we need to bring a formula that is in next normal form into a negation normal form before we apply the definition of $\overline{p}$ on all real atomic propositions evaluated in an open interval.

Definition 25. *An STL formula Φ is said to be in negation normal form if the following holds:*

1. *Φ is in next normal form.*
2. *For every term $\neg\Psi$ that is part of Φ, Ψ is a Boolean atomic proposition. In other words, the negation operator is only applied to Boolean atomic propositions.*

Proposition 11. *Every STL formula Φ can be converted into negation normal form $negnf(\Phi)$ with*

$$negnf(\Phi) = \begin{cases} p & ,\Phi = p, p \in AP \\ \neg p & ,\Phi = \neg p,\ p \in AP \text{ is Boolean} \\ c^T \cdot x \leq b & ,\Phi = \neg p \vee \Phi = \neg\overline{p}, p = (c^T \cdot > b) \\ c^T \cdot x < b & ,\Phi = \neg p \vee \Phi = \neg\overline{p}, p = (c^T \cdot \geq b) \\ c^T \cdot x \geq b & ,\Phi = \neg p \vee \Phi = \neg\overline{p}, p = (c^T \cdot < b) \\ c^T \cdot x > b & ,\Phi = \neg p \vee \Phi = \neg\overline{p}, p = (c^T \cdot \leq b) \\ c^T \cdot x \equiv b & ,\Phi = \neg p \vee \Phi = \neg\overline{p}, p = (c^T \cdot \not\equiv b) \\ c^T \cdot x \not\equiv b & ,\Phi = \neg p \vee \Phi = \neg\overline{p}, p = (c^T \cdot \equiv b) \\ negnf(\psi) & ,\Phi = \neg\neg\psi \\ \mathcal{N}_{[a]}(negnf(\psi)) & ,\Phi = \mathcal{N}_{[a]}(\psi) \\ \mathcal{N}_{[a]}(negnf(\neg\psi)) & ,\Phi = \neg\mathcal{N}_{[a]}(\psi) \\ negnf(\neg\psi_1) \vee negnf(\neg\psi_2) & ,\Phi = \neg(\psi_1 \wedge \psi_2) \\ negnf(\neg\psi_1) \wedge negnf(\neg\psi_2) & ,\Phi = \neg(\psi_1 \vee \psi_2) \\ negnf(\neg\psi_1 \vee \psi_2) & ,\Phi = \psi_1 \rightarrow \psi_2 \end{cases}$$

Proof. We have already proven that every STL formula can be converted into next normal form. Thus the first condition of a negation normal form can be achieved by transforming Φ into next normal form. The second condition stated in the definition of a next normal from is achieved by the recursive definition of $negnf$. The definition is recursively applied on the operands of the logical operators and the Next operator until atomic propositions are reached. When the definition is applied on negated operators, the negation is is applied on the operand instead of the operator. In the case of logical connectives the law's of De Morgan apply. Therefore, the negation is "pushed inwards until it is only applied to atomic propositions. If the atomic proposition is Boolean, we can end the transformation since the definition of the negation normal form allows for negated Boolean atomic propositions. However, if the transformation results in a negated real atomic proposition, a new atomic proposition takes its place where the relational operator is replaced by its opposite. In summary, both conditions for the negation normal form are met using previous propositions and the recursive definition of $negnf$. □

Our goal was to be able to encode an STL formula F into an equivalent QFLRA formula. An equivalent QFLRA formula was defined as a formula that could be used to produce a discrete test which would result in a continuous test signal that satisfies F after linear interpolation. The encoding described in this section achieves the goal by defining

1. a next normal form that achieves discretization necessary to produce discrete tests
2. a special encoding of atomic propositions $\overline{p}$ necessary for linear interpolation over a discrete test to result in a continuous test signal satisfying F
3. a function $E(F, t)$ to transform the next normal form of F into pure QFLRA such that it can be used as input for an SMT Solver that outputs a discrete test

With this encoding for STL formulae we can proceed with the algorithms used for the test generation.

Remark 5. *There exists satisfiable STL formulae which cannot be satisfied by continuous signals. An example for such an STL formula is* $(x \leq 3\, \mathcal{U}_{[1,2]}\, (x \geq 5)$. *For these formulae we could produce test signals assuming that the values for the atomic propositions are constant in open intervals. These signals are not necessarily continuous but they satisfy the STL formula. However, a scenario like this does not have to be regarded since the premise of this thesis is to generate contiuous test signals only.*

4.2 Algorithm for one test predicate

The definitions from above can now be employed on the test generation for a given formula ϕ and one of its test predicates ψ. The combination F of the two formulae is first converted into next normal form and every atomic proposition p that holds in an open interval is marked as $\overline{p}$. Afterwards, the negation normal form transformation is applied and then the helper clauses for atomic propositions $\overline{p}$ are inserted as described by the definition of $\overline{p}$. Finally, this next normal form that ensures the continuity with regard to STL is converted into QFLRA and used as input for an SMT-Solver. The solver outputs a discrete test which we can plot and through linear interpolation receive a test signal as depicted by the last two steps in Figure 4.2.

So far we have shown the test generation for ϕ and exactly one test predicate of ϕ. The following section deals with the complete test generation algorithm. This algorithm produces a test for each of the test predicates.

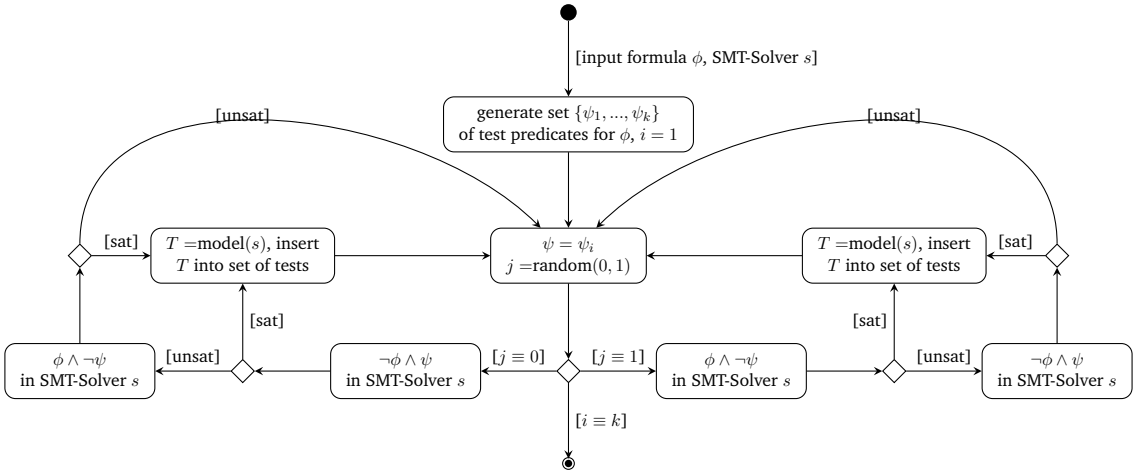

Fig. 4.5: Test generation algorithm for STLInspector

4.3 Algorithm for a list of test predicates

Lets assume the formula ϕ is to be tested. In order to do so we need to generate a set of test predicates using the coverage criteria described in the previous section. Figure 4.5 shows the procedure of the test generation. For each test predicate ψ either a positive or a negative test is computed. What kind of test is generated is randomly chosen. It is determined by a variable j, where $j \equiv 1$ produces a positive and $j \equiv 0$ a negative test. If the chosen kind of test cannot be generated for an individual predicate, the algorithm tries to generate the respective other kind first before "giving up" and continuing with the next predicate. It can be the case that neither positive nor negative tests can be generated if the predicate is semantically equivalent to the original formula. The two conjunctions $\phi \wedge \neg\psi$ and $\neg\phi \wedge \psi$ represent new temporal logic formulae that are converted into QFLRA and then input to a satisfiability solver. The solver outputs a satisfying assignment (model) for each conjunction where the model for $\phi \wedge \neg\psi$ is considered a positive test for ϕ and $\neg\phi \wedge \psi$ a negative test as described above.

4.4 Test Reduction

The way that the test generation algorithm is described above, we derive one test per test predicate (as long as this test predicate is not semantically equivalent to the formula under test). This would result in too many tests to be checked by the engineer due to the large number of test predicates a formula can produce using the coverage criteria of Section 3. Therefore, it is necessary to reduce the number of tests. The next remark describes the idea the test reduction is based on:

Remark 6. *Let ϕ denote an STL formula under test and $TP = \{\psi_1, ..., \psi_k\}$ be the set of test predicates generated for ϕ. Furthermore, let T be a test produced with the test*

predicate ψ_i, hence T distinguishes ϕ and ψ_i (T kills ψ_i). Then it can be the case that there exists another test predicate ψ_j that can also be distinguished (killed) by T, i.e. for a positive test T we have

$$T \models (\phi \wedge \neg\psi_i) \ \wedge \ T \models (\phi \wedge \neg\psi_j)$$

and for a negative test T we have

$$T \models (\neg\phi \wedge \psi_i) \ \wedge \ T \models (\neg\phi \wedge \psi_j)$$

The following algorithm demonstrates the iterative reduction of test cases for ϕ using this strategy for only positive tests. The mechanism needs only slight changing to be applied on negative tests also. Input is ϕ itself and the set TP of test predicates:

1. Let ψ be the first test predicate that was not yet chosen by the algorithm. Calculate a model for $\phi \wedge \neg\psi$ that will be called the test case T.

2. For each $\psi' \in TP$ with $\psi' \neq \psi$ check if $\neg\psi' \wedge T$ is satisfiable. Only if it is satisfiable, T lies outside of the set of signals accepted by ψ'. Thus T can serve as a test to rule out ψ'. We therefore do not have to produce any tests for ψ' and delete it from TP. We insert T into the set of test cases S produced for ϕ.

3. We continue with the first step until all predicates of TP have been checked. Finally, S holds all produced tests cases.

The algorithm works analogously for negative tests except that in the second step one has to check if $\psi' \wedge T$ is satisfiable. In the overall test generation procedure the algorithm is included after obtaining a new test. All predicates that are ruled out by this test will not be considered afterwards. We can now define the coverage of a test:

Definition 26. *Let ϕ be an STL formula, $TP = \{\psi_1, ..., \psi_k\}$ the set of test predicates generated for ϕ and T a test that was generated using the predicate ψ_i. In the following $|X|$ denotes the cardinality of a set X. If T is a positive test, we define the coverage of T as:*

$$coverage(T) := \frac{|\{\psi \in TP \ : \ T \models (\phi \wedge \neg\psi)\}|}{|TP|}$$

If T is a negative test, then we define its coverage as:

$$coverage(T) := \frac{|\{\psi \in TP \ : \ T \models (\neg\phi \wedge \psi)\}|}{|TP|}$$

Thus, the coverage of a test describes the relation of the number of predicates the test kills to the total number of predicates that were generated for the given formula.

Example:
We will now illustrate the test reduction on the following concrete example formula $\phi = \mathcal{F}_{[0,1]}(p)$, where p denotes some atomic proposition. As a coverage criterion we use property mutation. For simplicity we consider only the mutants $m_1 = \mathcal{N}_{[1]}(p)$, $m_2 = \mathcal{G}_{[0,1]}(p)$, and $m_3 = \mathcal{F}_{[0,1]}(\neg p)$ to make up the set of test predicates TP. We assume that calling the SMT solver on $\phi \wedge \neg m_1 = \mathcal{F}_{[0,1]}(p) \wedge \neg\mathcal{N}_{[1]}(p)$ returns the assignment $p_0 = \top, p_1 = \bot$ as a model. Hence T equals $p_0 \wedge \neg p_1$. Obviously $\neg m_2 \wedge T = \neg\mathcal{G}_{[0,1]}(p) \wedge (p_0 \wedge \neg p_1)$ is satisfiable since the atomic proposition p does not hold in the second time step as requested by $\neg\mathcal{G}_{[0,1]}(p)$. Thus T can also serve as a test case to rule out m_2. For m_3, however, $\neg m_3 \wedge T = \neg\mathcal{F}_{[0,1]}(\neg p) \wedge (p_0 \wedge \neg p_1)$ is not satisfiable since p is false in the second time step which is not acceptable by $\neg\mathcal{F}_{[0,1]}(\neg p)$. In this case, we therefore need to calculate an own test for m_3. We finally end up with only two tests produced for the three mutants. The first test has a coverage of $\frac{2}{3}$, the second a coverage of $\frac{1}{3}$.

5 Implementation

> *If the implementation is hard to explain, it's a bad idea. If the implementation is easy to explain, it may be a good idea.*
>
> — **Tim Peters** (1999)
> Python Software Engineer

All theoretical aspects from above are implemented in the tool STLInspector. The tool is open source and issued unter the Apache License Version 2.0. The implementation is written in Python, the graphical user interface (GUI) in Javascript. Furthermore, the tool depends on the theorem prover Z3 [11] and language recognition tool ANTLR [12]. We will later go into detail on these two dependencies but first examine the architecture of STLInspector.

5.1 Architecture

STLInspector can be split into four main components as depicted in Figure 5.1. The parser class is responsible for converting a formula that was entered by the user in a defined textual format into an internally used temporal logic object. The representation of such temporal logic objects is part of the core component. The core further includes methods for the creation of temporal logic objects that are used as test predicates based on the coverage criteria discussed in chapter 3. The solver part of the program holds methods for test case generation and finally, the GUI displays all test results and is responsible for the interaction with the user. The following sections provide more information on these four components.

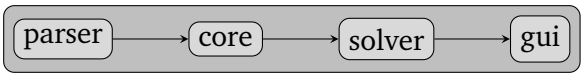

Fig. 5.1: Architecture of STLInspector

5.2 Parsing

The parser that comes with STLInspector allows reading formulae from both direct user input and text files. It parses the input with the help of ANTLR [12]. The procedure of defining a grammar and automatically creating lexer and parser classes using ANTLR is described in detail in Section 5.7 below.

Users have to abide by some rules when entering a formula. For example, the symbols for the temporal logic operators are restricted to only a couple of those commonly used: Globally can be expressed via `G` or `[]`, Finally via `F` or `<>`, Next via `N` or `o`, and Until and Release via `U` and `R` respectively. Their intervals are written in square brackets such as `[42, 1337]`. The logical operators can be declared by `&` (conjunction), `|` (disjunction), `->` (implication) and `!` (negation). All other rules, especially those for brackets, can be found in the documentation of STLInspector that is included in the appendix of this thesis (see 7.2).

When directly parsing a formula from a string, the parser operates as shown below. This method is used for obtaining formulae entered by users of STLInspector. We will parse the STL formula known from previous examples (see for instance Section 1.2 or 4):

```
> import Parser
> p = Parser()
> s = "F[0,42](vel > 133.7 -> (G[0,11] vel > 133.7))"
> formula = p.parse(s)
> print formula

(F[0,42](vel > 133.7 -> (G[0,11] vel > 133.7)))
```

Apart from parsing direct user input the implementation also provides a method that takes in a path to a file containing a formula and outputs the formula as a temporal logic object. Let us assume that a file *formula.txt* contains the same string that was parsed above. Then parsing it with our tool can be achieved via:

```
> import Parser
> p = Parser()
> formula = p.parse("path/to/formula.txt")
> print formula

(F[0,42](vel > 133.7 -> (G[0,11] vel > 133.7)))
```

5.3 Core

The core consists of implementations for STL and the coverage criteria. Each (temporal) logic operator is represented as a class. Hence the classes NOT, AND, OR, IMPLIES, FINALLY, GLOBALLY, NEXT, RELEASE and UNTIL exist. Since many methods are similar for these classes they all inherit from a super class called Clause. Every Clause object possesses methods to transform it into next normal form, negation normal form and encode it in a way that it can be passed into the Theorem Solver Z3. Moreover, a class representing atomic propositions is present. Its name is AP. The following example shows how to create a Clause object for the formula $(a\ \mathcal{U}_{[0,1]}\ b)\ \wedge\ (\mathcal{F}_{[1,2]}\ c)$ with the Boolean propositions a, b and c.

```
> import temporallogic
> a = AP("a")
> b = AP("b")
> c = AP("c")
> formula = AND(UNTIL(a, b, 0, 1), FINALLY(c, 1, 2))
> print formula

((a U[0,1] b) & F[1,2] (c))
```

All mutation operators described in Section 3.1 are implemented for the Clause objects. Each operator returns a list of mutants for the considered formula. Syntactically equivalent mutants are filtered. The next example shows the use of the temporal replacement operator for the formula that was defined in the example above.

```
> mutants = tro(formula)
> for m in mutants:
>       print m

((a R[0,1] b) & F[1,2] (c))
((a U[0,1] b) & N[2] (c))
((a U[0,1] b) & G[1,2] (c))
```

Besides mutation, the source code also holds implementations of the coverage criteria unique first cause and property inactive clause as described in Section 3. Since one can distinguish between the unique first causes satisfying a formula and those dissatisfying a formula there are two different methods for the unique first cause criterion. Users of STLInspector are not yet able to choose test predicate generation via these two coverage criteria. However, this feature can be used independently of the GUI and will also be incorporated shortly. The code snippet below portrays the method for computing predicates through UFC and PICC. We use a different example formula than the one in the previous examples in order to simplify the illustration.

```
> import coveragecriteria
> formula = FINALLY(AP("a"))
> generate_coveragecriteria_mutants(f)

UFC+ MUTANTS FOR F (a):
(! (a) U a)

UFC- MUTANTS FOR F (a):
(! (a) U (! (a) & G (! (a))))

PICC MUTANTS FOR F (a):
(! (a) U (a & (a | F (a))))
(! (a) U (! (a) & (a | F (a))))
```

5.4 Solver

The test predicates for a given formula that were parsed in the very beginning of the program are finally passed on from the core to the solver. The solver uses the mutants when generating tests as described by the theory from Section 4. The main method is the test generation method that randomly generates either a satisfying or a dissatisfying kind of test with regard to the original formula. Its structure roots from the algorithm described in Section 4.3 and uses the Z3 Theorem Prover as a satisfiability solver. More information about this dependency is given in the next subsection.

The encoding of the formulae is performed directly on the class objects. It is achieved according to the theories of Section 4.1. Last but not least does the test generation include the test reduction algorithm from Section 4.4. Besides the test generation method, this component also includes a method to convert the models returned by the SAT solver, thus the so-called discrete tests, into plottable format that is passed onto the GUI.

The solver interacts closely with the GUI since we want to enable the user to change her formula if she encounters a test that does not fit with the requirements. This is due to the fact that the main idea of STLInspector is to deepen the understanding of an STL formula inserted by the user and hence she must be able to improve the formalization of her requirements based on knowledge she gained using the program. The general interaction of the program with the user is summarized in the activity chart of Figure 5.2.

5.5 Graphical User Interface

The graphical user interface[1] (GUI) is responsible for the interaction with the user. That means the GUI passes user input to the parser and the solver and the other way

[1] I would like to thank my supervisor Hendrik Röhm for his support with the implementation of the graphical user interface of STLInspector

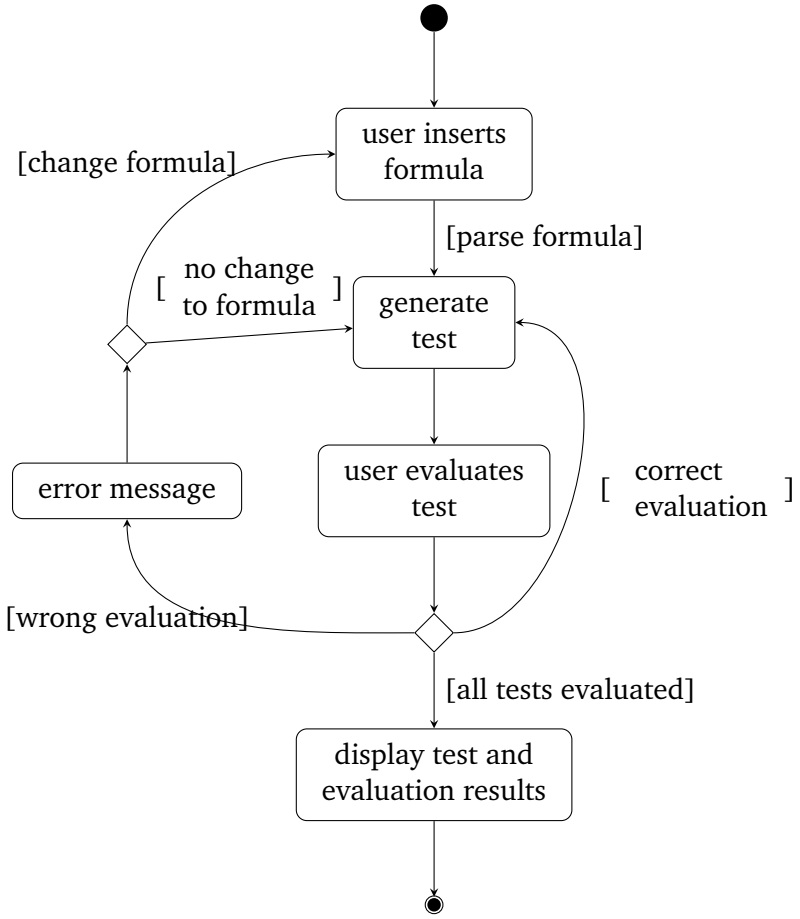

Fig. 5.2: Program flow of STLInspector

around displays the output of the solver to the user. User input passed to the parser consists of formulae as mentioned in Section 5.2. The communication between the solver and the GUI involves the tests and evaluations as illustrated in Figure 5.2.

STLInspector has three main GUI components: the start page (Figure 5.3), the project page (Figures 5.4 and 5.6) and the test display (Figure 5.5). The start page lists all requirements projects that either have been created in the current session or are saved and can be loaded from the disk. The page also shows the mutation coverage that has been achieved by the evaluations of the projects.

Fig. 5.3: STLInspector - Startpage: Overview of requirement projects.

Each requirements project consists of a textual requirement and a formalization of this requirement in form of a temporal logic formula. Both of these items can be set in the overview for a specific project as depicted in Figure 5.4. After they have been

Fig. 5.4: STLInspector - Projectpage: Requirement and STL formalization.

entered, users can start to evaluate the tests for the formula (Figure 5.5). After each test STLInspector shows how many mutants have been killed by the test (coverage). If the user's evaluation of the test differs from the actual kind, an error message is shown and the coverage becomes red indicating that a wrong evaluation has been chosen.

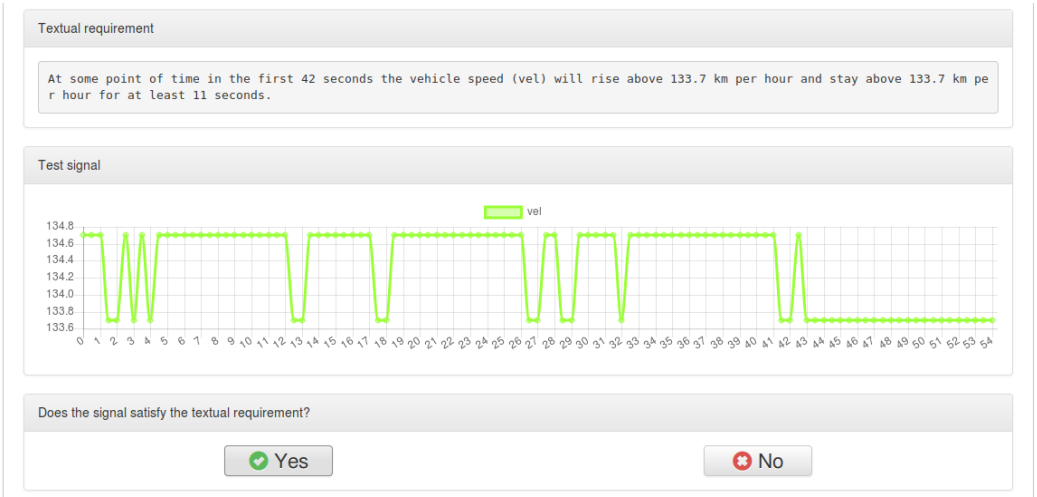

Fig. 5.5: STLInspector - Test display: Categorization of test.

Evaluations by different users can be made separately and their results can be compared on the project page to allow for discussion about the formalization. This is shown in Figure 5.6.

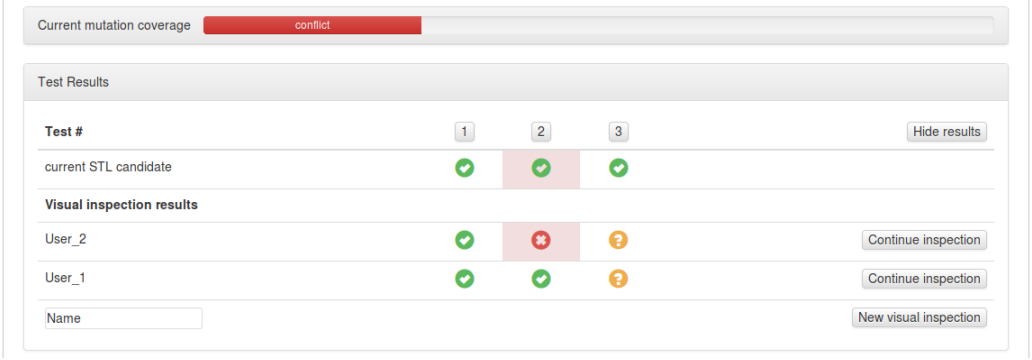

Fig. 5.6: STLInspector - Evaluation overview: Comparison of different user evaluations. One categorization is in conflict with the STL candidate's satisfiability.

5.6 Z3 Theorem Prover

The Z3 Theorem Prover [11] is a theorem prover developed by Microsoft Research. Licensed under the MIT license the code is open source and can be found on Github. Z3 supports arithmetic, fixed-size bit-vectors, extensional arrays, datatypes, uninterpreted functions and quantifiers as stated on the official website by Microsoft Research. In our case the solver was to generate satisfying assignments for QFLRA formulae as described in Section 4.2. Since our tool is written in Python all interactions with Z3 are done using Z3Py, the Z3 API for Python.

```
> import z3
> a = Real("a")
> b = Bool("b")
> formula = And(a >= 42, b)
> s = Solver()
> s.add(formula)
> if s.check() == sat:
>       print s.model()

[b = True, a = 42]
```

5.7 ANTLR

ANTLR [12] is short for ANother Tool For Language Recognition. Given a grammar for a specific language, ANTLR generates source code for a lexer and a parser of the language described by the grammar. It is mainly used with Java but also supports C, C#, Python, Ruby, Perl, JavaScript and some other minor programming languages. ANTLR is free software and published under the BSD-License.

Grammars used with ANTLR have to be context free and written somehow similar to a Backus-Naur-Form (see illustration below). ANTLR distinguishes between lexer and parser rules. Lexer rules define tokens of the language. Tokens define how input

is split up into multiple parts that can then be processed separately. Parser rules read the token streams that were generated by the lexer and output into abstract syntax trees. The trees may further contain transformation rules on input or tree nodes.

The following example shows how to define a grammar for a language built of STL Globally operators and Boolean atomic propositions. The second part then shows how to generate Python code using ANTLR for recognition of this language. We will exemplify the parsing of a simple Globally operation. For the sake of simplicity we ignore the interval of the temporal operator.

```
grammar Globally;

@header {
from temporallogic import *
}

globallyParserRule returns [Clause c]:
        TOKEN_GLOBALLY TOKEN_AP
        {$c = GLOBALLY(AP($TOKEN_AP.text))};

TOKEN_AP                : [A-Za-z0-9_]+;
TOKEN_GLOBALLY          : 'G';
```

```
> from antlr4 import *
> from GloballyLexer import GloballyLexer
> from GloballyParser import GloballyParser
> lexer = GloballyLexer(InputStream("G p"))
> stream = CommonTokenStream(lexer)
> parser = GloballyParser(stream)
> print parser.globallyParserRule().c

(G p)
```

6 Conclusion

6.1 Evaluation

We evaluate the effectiveness of STLInspector in a case study based on an online survey[1] by Dokhanchi et al. [5]. In the online survey users had to formalize textual requirements using STL. For 6 requirements we received access to the input of 11 different users and are thus able to test the efficiency of mutation-based test signals in finding errors on a total of 66 formalizations. The 6 requirements can be found in Figure 6.1.

Requirement	**STL formalization used as correct candidate**
At some point in time in the first 30 seconds, vehicle speed (v) will go over 100 and stay above for 20 seconds.	$\mathcal{F}_{[0,30]}(\mathcal{G}_{[0,20]}v > 100)$
At every point in time in the first 40 seconds, vehicle speed (v) will go over 100 in the next 10 seconds.	$\mathcal{G}_{[0,40]}(\mathcal{F}_{[0,10]}v > 100)$
If, within 40 seconds, vehicle speed (v) is above 100 then, within 30 seconds from time 0, engine speed (w) should be over 3000.	$(\mathcal{F}_{[0,40]}v > 100)$ $\rightarrow (\mathcal{G}_{[0,30]}w > 3000)$
If, at some point in time in the first 40 seconds, vehicle speed (v) goes over 80 then from that point on, for the next 30 seconds, engine speed (w) should be over 4000.	$\mathcal{F}_{[0,4]}((v > 80)$ $\rightarrow (\mathcal{G}_{[0,30]}w > 4000))$
In the first 40 seconds, vehicle speed (v) should be less than 100 and engine speed (w) should be under 4000.	$\mathcal{G}_{[0,40]}((v < 100) \wedge (w < 4000))$
In the first 40 seconds, whenever the vehicle goes into gear 4, then after 1 second, the engine speed (w) should be below 4000 until the vehicle speed settles in the range 100 to 110 in less than 10 seconds.	$\mathcal{G}_{[0,40]}(gear4 \rightarrow$ $(\mathcal{N}_{[1]}((w < 4000)\mathcal{U}_{[0,10]}$ $((v \geq 100) \wedge (v \leq 110)))))$

Fig. 6.1: Requirements formalized in STL from the online survey by Dokhanchi et. al. [5]

For each of the formalizations that were entered by the participants of the online survey, we check whether STLInspector generates at least one test based on the user's formalization that is able to kill the correct STL formalization as listed in

[1] We gratefully acknowledge the support of Bardh Hoxha and his colleagues to get access to some results of their survey

the second column of the table in Figure 6.1. In other words, we analyze whether STLInspector produces test signals that allow us to distinguish between a potentially wrong formalization and the correct one.

	total	unique
Formalizations	66	31
Faulty formalizations	44	26
Faulty formalizations detected by STLInspector	44	26

	min	⌊avg⌋	max
Mutants per formalization:	22	40	60
Tests per formalization:	3	6	11

Fig. 6.2: Evaluation results

Figure 6.2 gives some numbers on the process of the evaluation. Input to STLInspector are only 31 unique formalizations of the total of 66 since there are multiple cases were users had entered the same formulae. For each unique formalization, STLInspector generates a test suite with 100% mutation coverage that consists of 6 test signals on average (minimum 3, maximum 11). Out of the 66 total (31 unique) formalizations we are able to distinguish all of the 44 faulty ones (26 unique ones) using STLInspector. Therefore, the evaluation succeeded with 100% detection rate of errors based on the mutation operators used.

Most formalizations that were entered by the participants of the online survey contain multiple errors and can thus be considered higher order mutants of the correct STL candidate. As mentioned in Section 3.1, the mutation operators produce only first order mutants. Although we have no proof for the first order mutants to be sufficient to also kill higher order mutants, the evaluation suggests that this is possible since also faulty formulae with more than one error are detected.

6.2 Future Work

STLInspector is open source and we hope for the community to help us improve and extend the program in the future. Especially regarding error classification and localization there are still many aspects that could be incorporated into STLInspector. We will briefly sketch out some of these ideas:

1. Error Classification: At the moment all that a user knows after the validation is whether the formula contains any error described by the mutation operators used for test case generation of STLInspector. It would be helpful for the user of STLInspector to know exactly which mutants were killed by a test she is

evaluating. Only then would she know *what kind of errors* were made in the formalization.

2. Error localization: Furthermore, it would be an advantageous feature to be able to validate only subformulae of a given STL candidate. This would increase the chances for the user to locate *where the error lies* in a wrong formula.
3. In addition to the features above one could extend STLInspector with solutions facing problems that are relevant for CPS and embedded systems in general. An example for this would be the integration of signals with a lot of noise.

Besides implementing additional features, the theory that STLInspector is based on can also be pursued further. For example one could research whether there exists a proof that higher order mutants can be killed by tests generated using first order mutants.

6.3 Summary

The goal of this Bachelor's thesis was to develop a tool for the mutation-based validation of temporal logic specifications that enables for certain guarantees to be insured. This goal was achieved by the implementation of STLInspector [2]. In this composition we covered the following aspects for the realization of STLInspector: To begin with, we laid the basis of the temporal logic validated by STLInspector that is called Signal Temporal Logic (STL) [4]. We then expounded how the theory of various coverage criteria can be applied to STL formulae in order to formulate guarantees about the correctness of the specification after successful validation. This part included the extension of mutation operators on STL that had been defined in previous literature [8, 9] only for Linear Temporal Logic (LTL) formalizations. Likewise, the extension of an SMT-encoding for STL formulae that is based on existing encodings for LTL was discussed. This extended encoding constitutes the basis for test generation. Since STL formulae are continuous, the extension focused especially on ensuring continuity of the encoding. All theory was implemented in the tool STLInspector, which was finally evaluated and shown to be effective for the desired validation of STL formulae. The user-friendly interface that is deployed with the tool visualizes the validation tests as graphs. This simple test format permits even users without knowledge of STL to validate specifications. Therefore, STLInspector makes the assessment of formal specifications accessible to a wide range of developers in industry [2]. The bachelor's thesis hence contributes to improve the application of formal specifications in industrial practice.

7 Appendix

7.1 Implementation Problems

I would like to document an issue I encountered during the implementation of STLInspector. The issue was raised when I tried to speed up the conversion of STL formulae into Z3 instances. The conversion is performed since STL formulae are internally represented as `Clause` objects which cannot be input to the SMT solver. Therefore, there is the need for a method to convert a `Clause` object into a Z3 instance. Let us assume that this method is called `encode()`. It encorporates all transformations described in Section 4.1 of a `Clause` object into, for example, next normal form or negation normal form and finally creates Z3 instances using the completely transformed `Clause` object.

In the beginning, the method `encode()` performed all operations consecutively and every transformation was called recursively on every operand of the given `Clause` object until the object was completely converted. This approach turned out to be too slow for large formulae. Already on formulae containing more than three nested temporal operands `encode()` took several minutes until termination. Hence the goal was to optimize its run time.

The first approach was to use threading for the conversions into next normal form and negation normal form. Although this resulted in a run time that was faster, the run time was not yet satisfactory. I discovered that the delays were made during the creation of Z3 instances and were not due to the next normal form or negation normal form. Since the Z3 instance creation was something not easily threadable in our implementation, I began to search for alternative ways to enhance the run time for `encode()`.

Following suggestions on Stackoverflow (see this post of mine) I started by building a string in Z3Py syntax for a `Clause` object when I wanted to create a Z3 instance for the object. The string was input to the Python function `eval` which then returned a Z3 instance that corresponded to the original `Clause` object. This solution worked on small formulae and was also much faster than the recursive implementation from before. However, for larger formulae a `Memory Error` was thrown by Python. The reasons for this are Python internal memory restrictions on the stack of `eval` that

are described in the answer of my post on Stackoverflow but are irrelevant for this thesis.

So finally I implemented `encode()` such that it created Z3 instances from a string using the API `Z3_parse_smtlib_string`. In order to do so I had to encode the formulae into SMT 2.0 format. This change of the implementation achieved the desired run time and, due to the fact that the `Z3_parse_smtlib_string` API allowed for a larger stack than `eval` when parsing a string into a Z3 instance, all formulae could be converted successfully.

7.2 Parser Documentation

The following section holds the documentation for the parser of STLInspector. It explains how users have to enter STL formulae in order for it to be valid input for the program.

Input: Atomic Propositions
Atomic Propositions can be either Boolean variables or relational expressions. Boolean variables are made up of only alpha-numerical letters and cannot be named just 'G', 'F', 'N', 'o', 'U' or 'R' since these are keywords of the grammar. Also, variables are not allowed to start with a number.
Examples: `a`, `VAR` or `MyVar42`.

Relational expressions have the form `c^T x operator b` where `c` is a vector of numbers, `x` denotes a vector of variables, `operator` is one of the relational operators `>`, `<`, `>=`, `<=`, `==`, `!=`, and `b` is a single number.
Examples: `(1, 2)^T (x, y) <= 3` or `(1, 10, 100, 1000)^T (w, x, y, z) == 42`.

To avoid the use of one-dimensional vectors, we also allow the format `c x operator b` where `c` is a single number that is optional and `x` is a single variable.
Example: `3x < 4`, `42y != 42` or `x >= 9`.

Input: Propositional Logic
Conjunctions are written with `&` as the operator. When only atomic propositions are conjoined, we do not need brackets around the operands.
Examples: `a & b`, `a & b & c` or `(1,2)^T (x,y) != 4 & booleanVar & 3x > 4`.

However, as soon as one of the operators is another (temporal) logical formula, brackets are needed around this operand.
Examples: `(1,2)^T (x,y) != 4 & (booleanVar | 3x > 4)` or `a & (F[1,2] b) & (a U[3,4] b)`.

Disjunctions are written with | as the operator. The same rules as for conjunctions apply concerning brackets.
Examples: `a | b`, `booleanVar | (F[1,2] (1, 2)^T (x, y) <= 3)` or `a | (b & c) | d`.

Implications are written with -> as the operator. The same rules as for conjunctions and disjunctions apply concerning brackets. When no brackets are used for implications such as `a -> b -> c` the formula is interpreted as `((a -> b) -> c)`.
Examples: `a -> b`, `a -> (F[1,2] c)` or `(1, 2)^T (x, y) <= 3 -> (4, 5)^T (x, y) <= 6`.

Negation is done via the `!` operator. Brackets are set the same way as for Finally, Globally and Next - see rules below.
Examples: `! a`, `! (1, 2)^T (x, y) <= 3`, `! (a | b)` or `! (a | (b U[1,2] !c))`.

Input: Signal Temporal Logic
All following signal temporal logic operators have to be bounded by an interval. The interval is written as `[number, number]`. Moreover, all operands that consist of single atomic propositions do not need to be enclosed in brackets. However, operands that consist of connectives have to be set in brackets.

The temporal operator Finally can be expressed by either `F` or `<>`.
Examples: `F[1,2] singleAP`, `<>[0,1] (1, 2)^T (x, y) <= 3` or `F[0,42](a & b)`.

Globally is written as `G` or `[]`.
Examples: `G[1,2] singleAP`, `[][0,1] (1, 2)^T (x, y) <= 3`, `G[0,42](a | b)` or `G[3,4] (a -> (b U[4,3] c))`.

Next is written as `N` or `o`. Its interval consists of a single number that represents the step that is taken.
Examples: `o[1] a`, `N[1] (1, 2)^T (x, y) <= 3`, `N[42] (a | b)` or `N[2] (a | (b U[3,4] c))`.

The temporal operators Until and Release are written as `U` and `R` respectively. For brackets the same rules apply as for conjunctions, disjunctions and implications - see rules above.
Examples:`a U[1,2] b`, `a R[1,2] b`, `a U[0,42] (b | c)` or `(a U[3,5] c) R[0,2] (b U[7,11] c)`.

Bibliography

[1] H. Röhm, J. Oehlerking, T. Heinz and M. Althoff, *STL Model Checking of Continuous and Hybrid Systems*, In Automated Technology for Verification and Analysis, Volume 9938 of the series Lecture Notes in Computer Science, 14th International Symposium, ATVA 2016, Chiba Japan, October 17-20, 2016, Proceedings, pp. 412-427, (2016).

[2] H. Röhm, T. Heinz and E. C. Mayer, *STLInspector: STL Validation with Guarantees*, January 2017. Submitted to the 29th International Conference on Computer-Aided Verification (CAV), 22-28 July 2017, Heidelberg, Germany.

[3] V. Raman, A. Donzé, D. Sadigh, R. M. Murray and S. A. Seshia, *Reactive synthesis from signal temporal logic specifications*, Proceeding HSCC '15 Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, Pages 239-248, April 2015.

[4] O. Maler and D. Nickovic, *Monitoring temporal properties of continuous signals*, In Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings, pp. 152-166 (2004).

[5] A. Dokhanchi, B. Hoxha, G. E. Fainekos, *Metric interval temporal logic specification elicitation and debugging*, 13. ACM/IEEE International Conference on Formal Methods and Models for Codedesign, MEMOCODE 2015, Austin, TX, USA, September 21-23, 2015. pp 70-79 (2015).

[6] F. Kröger and S. Merz, *Temporal Logic and State Systems*, Texts in Theoretical Computer Science. An EATCS Series, Springer, 2008.

[7] A. J. Offutt and R. H. Untch, *Mutation 2000: Uniting the Orthogonal*, Mutation Testing for the New Century, Volumne 24 of the series The Springer International

Series on Advances in Database Systems, pp. 34-44, Springer, 2001.

[8] Gordon Fraser and Franz Wotawa, *Complementary Criteria for Testing Temporal Logic Properties* TAP '09 Proceedings of the 3rd International Conferences on Tests and Proofs, Pages 58 - 73, 2009.

[9] M.W. Whalen, A. Rajan, M.P. Heimdahl, S.P. Miller, *Coverage Metrics for Requirements-based Testing*, ISSTA 2006: Proceedings of the 2006 International Symposium on Software Testing and Analysis, Pages 25-36, ACM Press, New York, 2006.

[10] I. Pill and T. Quaritsch, *An LTL SAT Encoding for Behavioral Diagnosis*, Proceedings of the 23^{rd} International Workshop of Principles of Diagnosis, Malvern, UK, 2012.

[11] L. de Moura and N. Bjorner, *Z3: An efficient SMT solver*, Volumne 4963 of Lecture Notes in Computer Science, pages 337-340, Springer, 2008.

[12] T.J. Parr and R. W. Quong, *ANTLR: A Predicated-LL (k) Parser Generator*, SoftwarePractice and Experience Volume 25, Issue 7, Pages 789-810, July 1995.

List of Figures

Colophon

This thesis was typeset with LaTeX 2_ε. It uses the *Clean Thesis* style developed by Ricardo Langner. The design of the *Clean Thesis* style is inspired by user guide documents from Apple Inc.

Download the *Clean Thesis* style at `http://cleanthesis.der-ric.de/`.

Declaration

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Tübingen, Germany,

Eva Charlotte Mayer