Visual Explanations of Runtime Verification Verdicts

Master Thesis

Edo Mangelaars
Computing Science

Supervisor UU:
Dr. Wishnu Prasetya
Department of Information and Computing Sciences

Supervisor TNO:
Dr. Jacques Verriet
TNO-ESI

Second examinor:
Prof. dr. Gabriele Keller
Department of Information and Computing Sciences

August 2023

Abstract

In runtime verification, the behavior of a program or system is verified by automatically checking a specification expressed as a property in temporal logic on a finite execution trace. Understanding *why* a property is satisfied or violated is crucial in order to identify and repair defects and refine the specification, but for long traces and complex properties this may be difficult.

Eclipse TRACE4CPS™ is an open-source tool for the visualization and analysis of timed execution traces, with functionality for verification of real-time properties in a language based on Metric Temporal Logic (MTL). It uses informative prefix semantics to check whether the property is TRUE or FALSE, or NON-INFORMATIVE if the trace was truncated before a verdict could definitively be determined. Intermediate values of the checking procedure can be visualized in the trace view as an explanation of the verdict.

In this thesis, the presentation of the results of runtime verification in TRACE4CPS is improved in three ways, in order to increase the understandability of verdicts. First, the verification algorithm of TRACE4CPS is extended using a lattice-based four-valued informative prefix semantics to determine whether a NON-INFORMATIVE property is satisfied or violated *so far* (STILL_TRUE, STILL_FALSE). Secondly, the explanation features are extended to allow arbitrary subformulas of the property and the real-time constraints of the temporal operators to be visualized in an interactive visual explanation process. Finally, a method for finding the cause of LTL property violations based on Halpern and Pearl's definition of causality is extended to find the causes of timed MTL properties with FALSE or STILL_FALSE verdicts. These are then used to generate more concise and accurate universal visual explanations of property violations.

Acknowledgements

During the research project for this thesis, which was carried out as a research internship at TNO, I was fortunate to enjoy the support, guidance and encouragement of two supervisors. I would like to thank both dr. Wishnu Prasetya at Utrecht University and dr. Jacques Verriet at TNO-ESI for taking the time to steer me in the right direction, for allowing me the freedom to pursue my own path, and for giving excellent feedback and advice along the way.

I would also like to thank my colleagues at TNO-ESI for their time and feedback during the interim evaluation of the visualizations.

Lastly, thanks to dr. Martijn Hendriks for his helpful advice on TRACE4CPS and the MTL checking algorithm, and to dr. Clemens Dubslaff for the useful pointers to papers and an insightful discussion about causality in verification.

This research was carried out as part of the ITEA3 18030 MACHINAIDE project under the responsibility of TNO. The MACHINAIDE research is supported by the Netherlands Organisation for Applied Scientific Research TNO and Netherlands Ministry of Economic Affairs and Climate.

Table of Contents

1 Introduction

As software and computing systems become more ubiquitous – in traditional applications running on our desktop computers or mobile phones, or embedded in cars, medical systems and power plants – we increasingly depend on their correctness and reliability. This applies especially to safety-critical areas, where failures may have serious consequences for human lives and our environment. At the same time, these systems are becoming increasingly complex: functionality is distributed across multiple processing units and devices, and we require them to perform more and more complicated tasks.

One way to ensure reliability of these increasingly complex systems is through formal verification: the application of mathematically rigorous techniques to prove that a system correctly performs its intended functions. This can be done, for example, through *theorem proving* – the use of mathematical reasoning to obtain a proof of the correctness of a program – or through *model checking* – systematically determining whether all possible executions of a program satisfy a property, usually specified in a logical formula [Baier and Katoen 2008]. This requires that a program or system is expressed as a model that allows for formal reasoning, either through model-driven development – where the model is at the center of the development process and source code, documentation and specifications can be generated from it – or by formulating a model in parallel to development of the system.

Depending on the industry or application, this may be costly or incompatible with other desired development practices or tools. Furthermore, verifying a complete model of the behavior of a system is computationally very expensive, especially in cases where combinatorial explosion in the number of states that describe a program can occur, requiring more sophisticated techniques for representing the system or resorting to approximations.

1.1 Runtime verification

An alternative, more lightweight, approach is *runtime verification* [Leucker and Schallhart 2009]. Instead of verifying whether a specified property holds on all possible executions of a program, a single execution, or *run*, is verified. The specification of the desired behavior of a system is done using the same languages as those used in model checking, but the goal is to detect the occurrence of a violation of a property in an actual (or simulated) system. This is performed by a *monitor*, a device or program that reads a trace and yields a certain *verdict* [Leucker and Schallhart 2009]. Checking the live execution of a running system is called *online monitoring*, and when we check a recorded execution we speak of *offline monitoring*. We use the terms *offline* and *online runtime verification* to refer to runtime verification using offline and online monitoring, respectively.

Of course, when using runtime verification we cannot obtain the same level of confidence in the correctness of a system as we can with model checking: we can only verify that an error did not occur in the interval in which a program was monitored or a trace was recorded, but we cannot be assured of the total absence of errors. In return, however, we gain the possibility to use formalisms and techniques from the body of knowledge surrounding model checking even where a full-blown verification solution would be impossible or too expensive.

Another difference between runtime verification and model checking is that, while in model checking programs are usually modeled as having infinite executions, recorded execution traces are usually finite. We differentiate between the case where an execution is finite because the program naturally terminates, or because we were only able to capture part of an execution (e.g. because of practical constraints in the length of a trace and the amount of memory we have to store it). This requires us to adapt the semantics of the specification languages used, and to deal with the possibility that a trace does not contain enough information to obtain a verdict.

1.2 Visualization and explanation

Despite the fact that formal verification enjoys increased applicability with runtime verification, there are barriers that prevent it from being widely used.

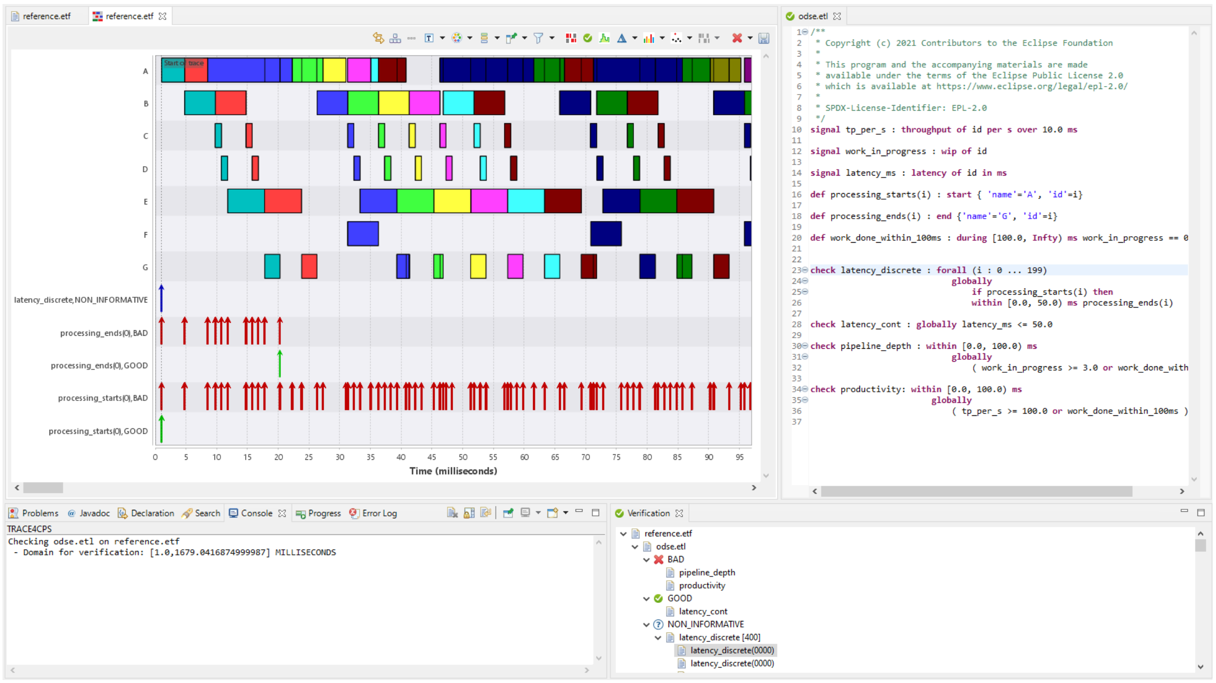
Formulating properties is specialized work. Finding the property that exactly captures the desired behavior of a program is not easy, and using the languages that are used to specify properties about programs is a separate skill from programming in general-purpose programming languages. The expression of properties in these languages requires a high level of insight into how an execution of a program and a property interact. At minimum, a runtime verification tool allows a user to check a property on a trace or on a running program, and it returns a verdict on whether the property holds or not. However, this may not be enough information for the user to determine whether a fault lies in the system being verified or in the property that is still being formulated. Often, a runtime verification tool will show at which point in the execution a property was falsified, but this may still be insufficient when the property is complex [Beer et al. 2012].

When verifying a property, the quality of feedback of runtime verification is important for any verdict. When given a negative result, a clear explanation can help a user debug the system being verified, potentially saving a lot of time spent localizing the error in the system. When given a positive result, explaining how this was determined increases the user's trust in the result. Even when the trace does not contain enough information to obtain a verdict there is often useful information to be obtained from partial results of checking the property.

1.3 Improving verification feedback of Eclipse TRACE4CPS™

Eclipse TRACE4CPS™ is a visualization and analysis tool for the performance engineering of cyber-physical systems [Hendriks et al. 2023], written in Java as a plug-in for the Eclipse platform. It was first developed by TNO-ESI (as ESI TRACE), and has recently become an open-source project hosted by the Eclipse Foundation [*Eclipse TRACE4CPS™* 2022].

TRACE4CPS works on execution traces specified in a human-readable file format that can come from any domain, source or level of abstraction. A trace contains timestamped and user-annotated events (which have a single timestamp), claims on resources (which have a start and end timestamp) and continuous signals (which are defined continuously within a time domain). TRACE4CPS has functions for visualizing traces and analyzing several aspects of performance. It allows for specifying and checking properties (of both the performance and the behavior of a system) using ETL, a property specification language based on MTL (a temporal logic for the



```
/**
 * Copyright (c) 2021 Contributors to the Eclipse Foundation
 *
 * This program and the accompanying materials are made
 * available under the terms of the Eclipse Public License 2.0
 * which is available at https://www.eclipse.org/legal/epl-2.0/
 *
 * SPDX-License-Identifier: EPL-2.0
 */
signal tp_per_s : throughput of id per s over 10.0 ms

signal work_in_progress : wip of id

signal latency_ms : latency of id in ms

def processing_starts(i) : start { 'name'='A', 'id'=i}

def processing_ends(i) : end {'name'='G', 'id'=i}

def work_done_within_100ms : during [100.0, Infty) ms work_in_progress == 0


check latency_discrete : forall (i : 0 ... 199)
                             globally
                                 if processing_starts(i) then
                                 within [0.0, 50.0) ms processing_ends(i)

check latency_cont : globally latency_ms <= 50.0

check pipeline_depth : within [0.0, 100.0) ms
                             globally
                                 ( work_in_progress >= 3.0 or work_done_with

check productivity: within [0.0, 100.0) ms
                         globally
                             ( tp_per_s >= 100.0 or work_done_within_100ms )
```

```
TRACE4CPS
Checking odse.etl on reference.etf
 - Domain for verification: [1.0,1679.0416874999987] MILLISECONDS
```

```
reference.etf
  odse.etl
    BAD
      pipeline_depth
      productivity
    GOOD
      latency_cont
    NON_INFORMATIVE
      latency_discrete [400]
        latency_discrete(0000)
        latency_discrete(0000)
```

Figure 1.1: A screenshot of the results of verification in TRACE4CPS (before)

specification of the behavior of timed systems [Hendriks et al. 2016a]) and STL (a variant of MTL for continuous-time signals [Hendriks et al. 2023]). Thus, TRACE4CPS can be used as an offline runtime verification tool.

The goal of the research project of this thesis was to improve the understandability of how the results of the verification of ETL properties on a trace are presented in TRACE4CPS. Thus, the central research question is as follows:

RQ0. How can the presentation of runtime verification results be improved to help the user understand a verdict on an execution trace in TRACE4CPS?

The insights gained by answering this question are expected to be applicable to any application of runtime verification of temporal properties, but this thesis is focused on the application of techniques for improving the understandability of verdicts in TRACE4CPS. To further bound the scope of the project, the research is limited to the verification and feedback of properties on the discrete parts of traces (events and claims, specified using MTL properties) and we leave improving the feedback of properties on continuous parts of traces (signals, specified using STL properties) for future work.

Three areas in which the verification feedback of TRACE4CPS was deemed to be lacking were identified, which were used to divide the central research questions into three sub-questions.

1.3.1 Verdicts

After checking a property on a trace, the results are presented as a *verdict*, which can be either GOOD, meaning that the trace satisfies the property, BAD, meaning that the trace violates the property, or NON-INFORMATIVE, which signifies that the trace does not contain enough information to definitively conclude the satisfaction or violation of the property. The verdicts of a set of properties are shown in the lower right of Figure 1.1.

The validity of properties on the events and claims in a trace is based on the assumption that traces are truncated versions (or *prefixes*) of longer or possibly infinite sequences of events. If the validity of a property is dependent on the information contained in the unknown continuation of the trace, the verdict could still become GOOD or BAD in the future. For some properties, a GOOD or BAD verdict cannot be reached on any finite prefix of an infinite sequence.

Thus, many practical properties and traces result in NON-INFORMATIVE verdicts, which give the user little useful information on whether the system under verification displays the desired behavior, even though the trace may contain useful information about the property's validity.

This leads to the following research question:

RQ1. How can the user, upon receiving a NON-INFORMATIVE verdict, be given information about the validity of the property so far?

1.3.2 Explanations

When the user double-clicks on the name of a checked MTL property, an *explanation* of the property is added to the visualized trace. The explanation is a visualization of the values in a memoization table that was generated during the checking process. A set of virtual events is generated corresponding to the events and (the left and right endpoints of) the claims that were evaluated in order to reach the verdict, and displayed as part of the trace visualization.

This explanation has two characteristics that limit its usefulness.

Firstly, a property specified in ETL is composed of a top-level formula, and can optionally be factored out into named subformulas. The ETL formula is then translated into an internal representation of the equivalent MTL formula. The virtual events in the explanation are only shown for the top-level formula and the named subformulas. If a more detailed explanation is desired, the user must factor out the property into subformulas, give them names and re-check the property. This can be a lot of effort and the resulting formulas may not be the most natural expression of the property.

On the other hand, visualizing the values of all subformulas on all checked events can quickly become overwhelming. Furthermore, the subformulas of the internal MTL representation of the property may not correspond to how the property was expressed by the user in ETL, and showing the intermediate values of a property in terms of this internal representation might be confusing to for user.

This leads to the next research question:

RQ2. How can all subformulas of an ETL property be explained without becoming overwhelming?

Secondly, a property explanation often contains many events that are not directly relevant to the violation or satisfaction of a property. For example, if a property is violated at a time late in the trace, virtual events for all events leading up to that point are generated even when they have no impact on the verdict. Determining which events are relevant to the verdict can be time-consuming, and the irrelevant events can even obscure the actual cause of the verdict.

Furthermore, if a property is a complex combination of conditions, it may be difficult or time-consuming to determine which of these conditions are responsible for a verdict.

To make the property explanations more useful in these situations, we pose the final research question:

RQ3. How can property explanations clearly explain the cause of the verdict?

1.4 Contribution

Through the process of improving the feedback of verification in TRACE4CPS and answering the research questions posed in Section 1.3, the following contributions are made in this thesis.

(**RQ1** – Chapter 4) A four-valued informative prefix semantics is formulated for timed traces and MTL formulas that splits the NON-INFORMATIVE verdict into a STILL_TRUE and a STILL_FALSE verdict. The algorithm in TRACE4CPS that computes an informative prefix verdict for a timed trace and an MTL formula, and generates the truth values used in the property explanations (developed by Hendriks et al. [2016a]), is extended to simultaneously compute a finite verdict which is used to decide between STILL_TRUE and STILL_FALSE. The extension preserves the optimizations made in the existing algorithm and the correctness of the intricacies of the informative prefix semantics, which is checked using property-based testing.

The four-valued semantics for MTL is reformulated as an elegant, directly inductive definition, rather than one in terms of the strong, weak and finite semantics of MTL, by using the observation that the four truth values form a lattice. The algorithm expressed in terms of partial order operations on this lattice is more concise and arguably more natural than the original algorithm, while maintaining its structure, computational complexity and optimizations.

(**RQ2** – Chapter 5) The property explanation functionality in TRACE4CPS is extended so that visual explanations can be generated for arbitrary subformulas of an ETL property, by creating a mapping between nodes in the abstract syntax tree of the ETL expression and those of the MTL formula in the translation procedure from ETL formulas to MTL formulas. An additional explanation type is created that shows the absolute intervals of constrained temporal operators as the property is checked on the trace.

A visual representation of the tree structure of the formula, together with the resulting array of explanation options, supports an interactive process through which the user can develop their understanding of the verdict.

(**RQ3** – Chapter 6) The approximate cause algorithm for LTL from [Beer et al. 2012] is applied to TRACE4CPS to analyze which events and which attributes/atomic propositions are the cause of a violation of an LTL property. The algorithm is extended to explain LTL properties containing a *next* operator that causes a STILL_FALSE verdict. Finally, the algorithm is modified and integrated into the MTL checking algorithm to generate cause explanations for MTL properties with real-time constraints.

A visual explanation feature is developed, based on the approximate cause algorithm, which offers a clear and concise indication of the reason that a property is violated that is independent of the structure of the MTL formula.

⋆ ⋆ ⋆

The new features of TRACE4CPS were developed in a fork of the TRACE4CPS code repository, and can be found at `https://gitlab.eclipse.org/edocodes/trace4cps/-/tree/thesis`

2 Background

In this chapter, a theoretical background of the subsequent chapters is given, which was gathered during a literature study at the beginning of the research project, as well as a review of existing tools in the area of visualization and explanation of verification results.

Section 2.1 introduces a set of logics with which properties of systems can be specified: LTL, for infinite sequences of states, and some extensions of LTL that capture notions of finite and truncated traces, and traces with real-time timestamps of states. A combination of MTL (Section 2.1.6), LTL$^{\mp}$ (Section 2.1.3) and STL (Section 2.1.7) is used in TRACE4CPS to check properties of traces of timestamped events. RV-LTL (Section 2.1.5) is used in Chapter 4 as inspiration to formulate a a four-valued logic to provide more detailed verdicts of property verification in TRACE4CPS.

Section 2.2 presents a theoretical notion of causality which is used in Chapter 6 to add new functionality to TRACE4CPS to explain why a verdict of property verification was reached.

Finally, Section 2.3 reviews some tools that offer features to visualize or explain the results of (runtime) verification, which were used as inspiration for the features developed in Chapter 5.

2.1 Runtime verification using temporal logic

To check formal specifications about the behavior of programs or systems, a program or system is modeled as the set of all possible executions of the program or system [Baier and Katoen 2008].

An execution (or *path*, or *trace*) is modeled as an infinite ordered sequence $\sigma = s_0, s_1, \dots$ or finite ordered sequence $\sigma = s_0, s_1, \dots, s_n$, where we call a $s \in \Sigma$ a state, and Σ is the set of all states. We use Σ^ω to denote the set of all infinite sequences, and Σ^* the set of all finite sequences of states.

To encode properties, we use a set AP of *atomic propositions*, such that each $s \in \Sigma$ is labeled with a subset of the atomic propositions which hold in that state by a labeling function $L : \Sigma \to 2^{AP}$. A property over the set AP is modeled as a (possibly infinite) subset of all possible sequences of labelings (i.e. a subset of $(2^{AP})^\omega$ or $(2^{AP})^*$) that exhibit some specified behavior. A sequence $\sigma = s_0, s_1, \dots$ satisfies a property P if and only if $L(s_0), L(s_1), \dots \in P$, which is denoted $\sigma \models P$. In this case, P is said to hold on σ. Sometimes, as a shorthand, states in a sequence are identified with the set of atomic propositions that hold in that state, so that an execution is a sequence in $(2^{AP})^\omega$ or $(2^{AP})^*$ which satisfies a property P if and only if $\sigma \in P$.

We can use logical formulas as a compact representation to express properties. A simple example is an invariant property, which is a property that specifies that all states in a sequence should fulfil some propositional logic formula. More expressive are formulas in a class of logics called temporal logic, in which properties can be specified so that properties of states can depend on those of other states in the sequence. We shall see several temporal logics in this chapter.

In *model checking*, a program or system is modeled as a transition system that encodes all possible executions. A state in such a model can, for example, represent the values of all variables of a program at some point in time. Using a variety of algorithms, one can efficiently check whether every execution in the (possibly infinite) set of all sequences generated by the model satisfy a property [Baier and Katoen 2008].

In contrast, in *runtime verification*, typically only a single execution is checked, either of a running system or a recorded trace of a previously run or simulated one. Because this is a technically much easier problem than model checking, algorithms and techniques can be used that would be intractible on transition systems [Markey and Schnoebelen 2003].

2.1.1 Infinite executions – LTL

A common temporal logic used for specifying the behavior of systems is LTL, Linear Temporal Logic [Pnueli 1977]. The syntax of LTL formulas is defined as follows.

> **Definition 2.1 (Syntax of LTL)** [Baier and Katoen 2008]
> Let p be an atomic proposition from a set AP. The set of LTL formulas is inductively defined by the following grammar:
>
> $$\varphi \coloneqq \mathsf{true} \mid p \mid \varphi_1 \land \varphi_2 \mid \neg\varphi \mid \mathsf{X}\,\varphi \mid \varphi_1\,\mathsf{U}\,\varphi_2$$

The *next* operator $\mathsf{X}\,\varphi$, often written as $\bigcirc\varphi$, specifies that φ should hold in the next state in the sequence. The *until* operator $\varphi_1\,\mathsf{U}\,\varphi_2$ specifies that there is some state (either the current state or some future state) in which φ_2 holds, and φ_1 holds in all states from now until that state.

Using the Boolean connectives $\land$ and $\neg$, the other Boolean connectives can be derived as usual:

$$\begin{aligned}\mathsf{false} &\overset{\mathsf{def}}{=} \neg\mathsf{true}\\ \varphi_1 \lor \varphi_2 &\overset{\mathsf{def}}{=} \neg(\neg\varphi_1 \land \neg\varphi_2)\\ \varphi_1 \rightarrow \varphi_2 &\overset{\mathsf{def}}{=} \neg\varphi_1 \lor \varphi_2\end{aligned}$$

The *until* operator can be used to derive the following two very useful temporal operators:

$$\begin{aligned}\mathsf{F}\,\varphi &\overset{\mathsf{def}}{=} \mathsf{true}\,\mathsf{U}\,\varphi\\ \mathsf{G}\,\varphi &\overset{\mathsf{def}}{=} \neg\,\mathsf{F}\,\neg\varphi\end{aligned}$$

The *finally* operator $\mathsf{F}\,\varphi$, often written as $\Diamond\varphi$ and alternatively called *eventually*, specifies that φ holds either in the current state or in some future state. The *globally* operator $\mathsf{G}\,\varphi$, often written as $\Box\varphi$ and alternatively called *henceforth*, specifies that φ holds in all states starting from the current state.

Finally, two derived operators that are sometimes used are the *weak until* operator:

$$\varphi_1\,\mathsf{W}\,\varphi_2 \overset{\mathsf{def}}{=} (\varphi_1\,\mathsf{U}\,\varphi_2) \lor \mathsf{G}\,\varphi_1$$

which acts like the *until* operator, except it does not require that a state be reached where φ_2 holds, and the *release* operator:

$$\varphi_1\,\mathsf{R}\,\varphi_2 \overset{\mathsf{def}}{=} \neg(\neg\varphi_1\,\mathsf{U}\,\neg\varphi_2)$$

which specifies that φ_2 holds up until and including a state where φ_1 also holds, or φ_2 always holds (φ_1 "releases" φ_2).

The semantics of LTL are formally specified as follows:

Definition 2.2 (Semantics of LTL on infinite sequences) [Baier and Katoen 2008]
Let $\sigma = s_0, s_1, \ldots$ be an infinite sequence, φ, φ_1 and φ_2 LTL formulas, i a position with $i \geq 0$, $p \in AP$ an atomic proposition, and let $L(s_j)$ be the set of atomic propositions which hold in state s_j, for all $j = 0, \ldots$.
$\sigma^i \models \varphi$ denotes that σ satisfies a property φ starting at position i. $\models$ is inductively defined as the smallest satisfaction relation such that

$$\begin{array}{lll}
\sigma^i \models \mathsf{true} & & \\
\sigma^i \models p & \text{iff} & p \in L(s_i) \\
\sigma^i \models \varphi_1 \wedge \varphi_2 & \text{iff} & \sigma^i \models \varphi_1 \text{ and } \sigma^i \models \varphi_2 \\
\sigma^i \models \neg\varphi & \text{iff} & \sigma^i \not\models \varphi \\
\sigma^i \models \mathsf{X}\,\varphi & \text{iff} & \sigma^{i+1} \models \varphi \\
\sigma^i \models \varphi_1 \,\mathsf{U}\, \varphi_2 & \text{iff} & \exists_{j \geq i} [\sigma^j \models \varphi_2 \text{ and } \forall_{i \leq k < j}\, \sigma^k \models \varphi_1]
\end{array}$$

We say that a sequence σ satisfies φ, denoted by $\sigma \models \varphi$, if and only if $\sigma^0 \models \varphi$.

In propositional logic, conjunction ($\wedge$) and disjunction ($\vee$) are called *dual* because of the following equivalences:

$$\neg(a \vee b) \equiv \neg a \wedge \neg b \qquad \neg(a \wedge b) \equiv \neg a \vee \neg b$$

In LTL, in addition to the propositional connectives, the operators F and G are dual, as well as the operators U and R, and the operator X is self-dual:

$$\begin{array}{lll}
\neg\,\mathsf{F}\,a \equiv \mathsf{G}\,\neg a & \neg(a \,\mathsf{U}\, b) \equiv \neg a \,\mathsf{R}\, \neg b & \neg\,\mathsf{X}\,a \equiv \mathsf{X}\,\neg a \\
\neg\,\mathsf{G}\,a \equiv \mathsf{F}\,\neg a & \neg(a \,\mathsf{R}\, b) \equiv \neg a \,\mathsf{U}\, \neg b &
\end{array}$$

2.1.2 Finite executions – LTL$_f$

In the semantics of LTL given above, it is assumed that executions are infinite. To deal with a program that may terminate or deadlock the last state can simply be repeated indefinitely [Manna and Pnueli 1992]. In (offline) runtime verification, however, we inherently deal with finite sequence, since we only have a finite amount of memory to store a trace recorded from an actual or simulated execution.[1] Even for programs that do not terminate, we would like to not have to wait until the end of an infinite computation to verify it. Moreover, in an online monitoring situation, it is useful to have a continuous verdict of the computation so far, rather than having to wait for a computation to finish.

To deal with this, we can explicitly define our temporal logics on finite executions. The first logic we will examine, which we will call LTL$_f$, is often called the traditional LTL semantics over finite paths, and is named FLTL in [Bauer et al. 2008]. In LTL$_f$, sequences are assumed to

[1] An infinite execution can be stored in a finite amount of memory by representing a repeating pattern as a cycle of a finite subsequence, but that does not apply when the execution does not come from a model but an actual system being recorded.

be finite but maximal. That is, the last state in the sequence represents the actual end of the computation, so the validity of the property depends entirely on the information that is available in the sequence.

To extend the previous definition of LTL to finite executions, we need to split the *next* operator into two versions [Manna and Pnueli 1992]: a *strong* version denoted $\mathsf{X}\,\varphi$ and a *weak* version denoted $\overline{\mathsf{X}}\,\varphi$. The strong version requires that there is a next state and that φ holds in that state, while the weak version holds either when there is no next state or φ holds in the next state.

The *until* operator is interpreted as a strong operator, so that it is only satisfied if its right operand is present in the trace. Its formal semantics is updated to account for finite sequences, and its weak counterpart W is defined as before.

Definition 2.3 (Semantics of LTL$_f$)

Let $\sigma = s_0, s_1, \ldots, s_n$ be a finite, non-empty sequence of length $|\sigma| = n + 1$. We use $\models_f$ to denote the satisfaction for LTL on finite sequences. The *strong next* and *until* operators for LTL on finite sequences are defined as follows:

$$\begin{array}{lll} \sigma^i \models_f \mathsf{X}\,\varphi & \text{iff} & i < n \text{ and } \sigma^{i+1} \models_f \varphi \\ \sigma^i \models_f \varphi_1 \,\mathsf{U}\, \varphi_2 & \text{iff} & \exists_{i \leq j \leq n} [\sigma^j \models_f \varphi_2 \text{ and } \forall_{i \leq k < j}\ \sigma^k \models_f \varphi_1] \end{array}$$

The other operators are the same as in Definition 2.2.

The *weak next* operator can be derived from the *strong next* operator:

$$\overline{\mathsf{X}}\,\varphi \overset{\mathsf{def}}{=} \neg\,\mathsf{X}\,\neg\varphi$$

In LTL$_f$, X is no longer self-dual. Instead, X and $\overline{\mathsf{X}}$ are dual (by definition).

To see why splitting the next operator into a strong and a weak version is necessary, consider the property $\mathsf{X}\,\neg a$. On any infinite sequence, the self-duality of X is valid: $\mathsf{X}\,\neg a \equiv \neg\,\mathsf{X}\,a$. On any sequence with a single state, however, $\mathsf{X}\,\neg a$ is false, but $\neg\,\mathsf{X}\,a$ is true, so the equivalence no longer holds. By separating the strong and weak *next* operator, we can instead say that $\mathsf{X}\,\neg a \equiv \neg\,\overline{\mathsf{X}}\,a$, which is valid on any finite (as well as any infinite) sequence.

2.1.3 Truncated executions – LTL$^{\mp}$

A finite sequence may not always be maximal, but can also be a truncated version of a longer or infinite execution. Truncated executions arise naturally in runtime verification, simulation or bounded model checking, where we may only have recorded or generated an execution up to a point in order to make verification practical. We say that a *truncated path* is a finite sequence that is interpreted as a *prefix* of an infinite sequence.

Consider the property $\varphi = \neg p \,\mathsf{U}\, \mathit{init}$, which specifies that p must not occur before *init* occurs. If we have a path in which we encounter p before we have encountered *init*, we know that φ is violated. However, if we have a path in which we have not yet encountered either p or *init*, we do not know if p occurs before *init*, so we should not conclude that φ is satisfied or violated. Yet, using the semantics of LTL$_f$ would give a false verdict for both paths.

To be able to distinguish between these scenarios we use the notion of *informative prefixes*. If at the end of a truncated path the evaluation of a formula φ is completed, we know the truth value on the execution of which the path is a prefix. In this case, we call the path *informative* [Kupferman and Vardi 2001].[2]

When a path is not informative for φ, it does not contain all the information necessary to conclude whether φ does or does not hold. This can be because there exist a continuation of the prefix that satisfies φ and a continuation that violates it, but it can also be because the knowledge of whether a formula can be satisfied or violated in a continuation is not contained in the path. For example, the formula $\mathsf{G}\,\mathsf{true}$ is satisfied by any infinite sequence because it is a tautology. However, there is no finite sequence that is informative for $\mathsf{G}\,\mathsf{true}$, because the fact that it holds on any continuation of such a sequence requires knowledge that it is a tautology, which cannot be gained from the prefix alone.

To allow us to reason about LTL on truncated paths, Eisner et al. [2003] propose a formalism based on two entwined versions of LTL (together called LTL$^{\mp}$ in [Bauer et al. 2008]) which differ in what to return when there is doubt about the truth of a formula at the end of a prefix.

In the *weak view* a property is satisfied when there is doubt: a property holds on a sequence if it does not contain evidence that the property is violated in the sequence (and thus any extension of the sequence). In the weak view, the formula $\mathsf{F}\,a$ holds for any finite sequence, and $\mathsf{G}\,b$ holds only if b holds in every state on the sequence. In the *strong view* a property is not satisfied when there is doubt: a property holds on a sequence if it contains all evidence needed to conclude that the property holds on any extension of the sequence. In the strong view, the formula $\mathsf{F}\,a$ holds only if a holds at some state in the sequence, and $\mathsf{G}\,b$ does not hold for any finite sequence. The semantics of LTL$_f$ is called the *neutral view*.

Definition 2.4 (Semantics of LTL$^{\mp}$) [Eisner et al. 2003]
Let $\sigma = s_0, s_1, \dots, s_n$ be a finite sequence of length $|\sigma| = n+1$, $p \in AP$ an atomic proposition, i a position with $i \geq 0$, and let $L(s_j)$ be the set of atomic propositions that hold at state s_j, for all $j = 0, \dots, n$. $\models^-$ is the satisfaction relation that corresponds to the weak view, with

$$\begin{array}{lll}
\sigma^i \models^- \mathsf{true} & & \\
\sigma^i \models^- p & \text{iff} & i > n \text{ or } p \in L(s_i) \\
\sigma^i \models^- \varphi_1 \wedge \varphi_2 & \text{iff} & \sigma^i \models^- \varphi_1 \text{ and } \sigma^i \models^- \varphi_2 \\
\sigma^i \models^- \neg\varphi & \text{iff} & \sigma^i \not\models^+ \varphi \\
\sigma^i \models^- \mathsf{X}\,\varphi & \text{iff} & \sigma^{i+1} \models^- \varphi \\
\sigma^i \models^- \varphi_1 \,\mathsf{U}\, \varphi_2 & \text{iff} & \exists_{j \geq i} [\sigma^j \models^- \varphi_2 \text{ and } \forall_{i \leq k < j}\, \sigma^k \models^- \varphi_1]
\end{array}$$

and $\models^+$ the satisfaction relation that corresponds to the strong view, with

$$\begin{array}{lll}
\sigma^i \models^+ \mathsf{true} & \text{iff} & i \leq n \\
\sigma^i \models^+ p & \text{iff} & i \leq n \text{ and } p \in L(s_i) \\
\sigma^i \models^+ \varphi_1 \wedge \varphi_2 & \text{iff} & \sigma^i \models^+ \varphi_1 \text{ and } \sigma^i \models^+ \varphi_2 \\
\sigma^i \models^+ \neg\varphi & \text{iff} & \sigma^i \not\models^- \varphi \\
\sigma^i \models^+ \mathsf{X}\,\varphi & \text{iff} & \sigma^{i+1} \models^+ \varphi \\
\sigma^i \models^+ \varphi_1 \,\mathsf{U}\, \varphi_2 & \text{iff} & \exists_{j \geq i} [\sigma^j \models^+ \varphi_2 \text{ and } \forall_{i \leq k < j}\, \sigma^k \models^+ \varphi_1]
\end{array}$$

[2]In [Kupferman and Vardi 2001], *informative* is defined in the context of safety properties, which can only be violated in finite time, and as such only refers to violations. We use an extended notion of informativeness that is also symmetrically defined for general LTL formulas when they are satisfied.

Note that in these definitions for the *next* and *until* operators, instead of requiring that the index j or $i+1$ is smaller than the length of the sequence in which it is used, "overflow" of the indices is used, combined with the fact that the semantics are defined past the end of the sequence: given a formula φ, a finite sequence $\sigma = s_0, \ldots, s_n$, and any $j > n$, $\sigma^j \models^- \varphi$ and $\sigma^j \not\models^+ \varphi$.

For example, consider a formula $\varphi = \mathsf{true}\ \mathsf{U}\ a$ and a prefix of length $n+1$ in which a does not occur. In the weak semantics, there exists a position $j \geq i$ such that a holds (namely, any $j > n$) so φ is weakly satisfied. In the strong semantics, a does not hold for any $j > n$, so φ is not strongly satisfied.

The negation operator switches between the strong and weak semantics, so the semantics form a coupled dual pair.

These semantics correspond to LTL$_f$ as follows:

> **Proposition 2.5 (Strength relation theorem)** [Eisner et al. 2003]
> Let φ be an LTL formula and σ a non-empty sequence.
>
> $$\sigma \models^+ \varphi \Longrightarrow \sigma \models_f \varphi \quad \text{and} \quad \sigma \models_f \varphi \Longrightarrow \sigma \models^- \varphi$$

Consequently, a sequence that violates a property in the weak semantics also violates it in the finite semantics and in the strong semantics. We also have that once a path satisfies a property in the strong semantics, any finite or infinite extension of that path also satisfies it in the strong semantics. Dually, once a path violates a property in the weak semantics, any finite or infinite extension of that path also violates it in the weak semantics. On infinite sequences, the weak, strong and finite semantics are all equivalent.

2.1.4 Truncated executions – LTL$_3$

An alternative way to reason about truncated sequences is proposed by Bauer et al. [2006] in a logic called LTL$_3$. Instead of recognizing informative prefixes, like LTL$^{\mp}$, LTL$_3$ is based on the concept of good and bad prefixes:

> **Definition 2.6 (Good/bad prefix for LTL)** [Kupferman and Vardi 2001]
> Let $\sigma = s_0, \ldots, s_n \in \Sigma^*$ be a finite sequence of length $n+1$, and φ an LTL formula. Given a sequence σ and σ', let $\sigma \cdot \sigma'$ denote the sequence that is the concatenation of σ and σ'.
>
> $$\sigma \text{ is a } \textit{bad prefix} \text{ for } \varphi \overset{\mathsf{def}}{\Longleftrightarrow} \forall \sigma' \in \Sigma^\omega : \sigma \cdot \sigma' \models \varphi$$
> $$\sigma \text{ is a } \textit{good prefix} \text{ for } \varphi \overset{\mathsf{def}}{\Longleftrightarrow} \forall \sigma' \in \Sigma^\omega : \sigma \cdot \sigma' \not\models \varphi$$

In other words, σ is a good prefix if every infinite continuation of σ is true, and a bad prefix if every infinite continuation of σ is false. Every informative prefix is either good or bad, but there are good prefixes and bad prefixes that are not informative. Going back to the example in Section 2.1.3 of the formula $\varphi = \mathsf{G}\,\mathsf{true}$: every finite sequence is a good prefix for φ, since φ is a tautology, but not an informative prefix for the reason given before.

LTL$_3$ uses a semantics $\models_3$ which is a function that evaluates a formula and finite sequence to one of the truth values in $\mathbb{B}_3 = \{\top, \bot, ?\}$. $\mathbb{B}_3$ is defined as a De Morgan lattice with $\bot \sqsubseteq ? \sqsubseteq \top$, with $\bot$ and $\top$ being complementary to each other, and $?$ being complementary to itself. The idea of the semantics of LTL$_3$ is that if every infinite sequence with prefix σ evaluates to the same truth value $\bot$ or $\top$, then $[\sigma \models_3 \varphi]$ also evaluates to this truth value, and if different continuations of σ yield different truth values it evaluates to ?.

It is defined formally as follows:

Definition 2.7 (Semantics of LTL$_3$) [Bauer et al. 2008]
Let $\sigma = s_0, \ldots, s_n \in \Sigma^*$ be a finite sequence of length $n+1$. The truth value of a LTL$_3$ formula φ with respect to σ is an element of $\mathbb{B}_3$, and is defined as

$$[\sigma \models_3 \varphi] \overset{\text{def}}{=} \begin{cases} \top & \text{if } \forall \sigma' \in \Sigma^\omega : \sigma \cdot \sigma' \models \varphi \\ \bot & \text{if } \forall \sigma' \in \Sigma^\omega : \sigma \cdot \sigma' \not\models \varphi \\ ? & \text{otherwise} \end{cases}$$

Because LTL$_3$ recognizes all good and bad prefixes, it is able to correctly give a positive or negative verdict for finite sequences where based on LTL$^{\mp}$ we could only conclude that it is non-informative.

However, as opposed to the previously defined logics, LTL$_3$'s semantics is not defined in an inductive manner (that is, by the meaning of its subformulas). The reason for this is as follows. Consider a proposition p with respect to the empty sequence ϵ. In LTL$_3$, both $[\epsilon \models_3 p]$ and $[\epsilon \models_3 \neg p]$ evaluate to ?. The join of these values is thus ?. However, given the above definition, $[\epsilon \models_3 p \vee \neg p] = \top$, since $p \vee \neg p$ is a tautology. Bauer et al. [2008] argue that any inductive definition of LTL$_3$ operating on $\mathbb{B}_3$ would not preserve this property.

According to Bartocci et al. [2018], evaluating $[\sigma \models_3 \varphi]$ on a finite sequence σ is a PSPACE-complete problem. It is implemented in an automata-based monitor procedure in [Bauer et al. 2006].

2.1.5 Truncated executions – RV-LTL

Using a three-valued logic for verdicts on finite sequence, either by saying that a sequence is *non-informative* using LTL$^{\mp}$ or by the addition of ? to the set of truth values in LTL$_3$, allows us to distinguish between a property being satisfied or violated and there not being enough information to give a definite verdict of a property. Whenever there is a continuation of a finite sequence that satisfies the property and a continuation that violates it, we give an inconclusive verdict.

However, there are situations where this is not satisfactory. Consider the following example from [Bauer et al. 2008] concerning the property $\mathsf{G}(r \to \mathsf{F}\, a)$, which says that all requests (r) must be acknowledged (a) eventually. Let r^ω and a^ω be the infinite sequences repeating r and a ad infinitum. For every finite prefix σ, we have that $\sigma \cdot r^\omega \not\models \mathsf{G}(r \to \mathsf{F}\, a)$ and $\sigma \cdot a^\omega \models \mathsf{G}(r \to \mathsf{F}\, a)$. Therefore, in LTL$_3$, $[\sigma \models_3 \mathsf{G}(r \to \mathsf{F}\, a)] = [\sigma \models_3 \neg\, \mathsf{G}(r \to \mathsf{F}\, a)] = ?$ for *all* finite words σ.

We call such a property *non-monitorable*. Even if we can never obtain a conclusive result, we might still want to perform verification using non-monitorable properties. In the case of the property $\mathsf{G}(r \to \mathsf{F}\, a)$, for example, we might want to distinguish between a sequence where all requests have been acknowledged and a sequence where there is a request that has not been acknowledged yet, even if we do not know if this will change in a continuation of the sequence.

To do this, Bauer et al. [2008] propose a four-valued variant of LTL called RV-LTL (Runtime Verification-LTL), which uses the syntax of LTL$_f$ (with separate weak and strong next operators) and draws verdicts from a set of four truth values $\mathbb{B}_4 = \{\top, \bot, \top^p, \bot^p\}$, where $\top^p$ stands for *presumably true* and $\bot^p$ stands for *presumably false*. $\mathbb{B}_4$ forms a De Morgan lattice with with $\bot$ and $\top$ being complementary, and the same for $\bot^p$ and $\top^p$.

The semantics of RV-LTL is a combination of LTL$_f$ and LTL$_3$ where the truth value of LTL$_3$ is taken when it is conclusive, and if LTL$_3$ provides an inconclusive verdict, LTL$_f$ is used to determine whether the property is presumably true or presumably false.

Definition 2.8 (Semantics of RV-LTL) [Bauer et al. 2008]
Let $\sigma = s_0, \ldots, s_n \in \Sigma^*$ be a finite sequence, and let $\models_3$ denote the satisfaction relation of LTL$_3$ and $\models_f$ that of LTL$_f$. The truth value of a RV-LTL formula φ with respect to σ is an element of $\mathbb{B}_4$ and is defined as

$$[\sigma \models_{\text{RV}} \varphi] \overset{\text{def}}{=} \begin{cases} \top & \text{if } [\sigma \models_3 \varphi] = \top \\ \bot & \text{if } [\sigma \models_3 \varphi] = \bot \\ \top^p & \text{if } [\sigma \models_3 \varphi] = \,?\, \text{ and } \sigma \models_f \varphi \\ \bot^p & \text{if } [\sigma \models_3 \varphi] = \,?\, \text{ and } \sigma \not\models_f \varphi \end{cases}$$

Like $\models_3$, the semantics for $\models_{\text{RV}}$ is not defined inductively for the same reason.

2.1.6 Timed executions – MTL

Metric Temporal Logic (MTL) [Koymans 1990], is an extension of LTL in which properties of *timed systems* can be specified. A timed system is a system where the correctness of its behavior is dependent on the timing of events. While in LTL it is possible to write properties about the (qualitative) relative ordering of events in time, MTL has the ability to specify (quantitative) real-time constraints. This allows us to express properties such as "A follows B within at most 5 time units", or "C happens at most once per 2 time units".

A *timed sequence* $\rho = (s_0, t_0), (s_1, t_1), \ldots$ is a sequence of pairs in $\Sigma \times T$, where Σ is the set of states, each paired with a timestamp from T, which is some totally ordered time domain such as (a subset of) $\mathbb{N}$ or $\mathbb{R}$. The sequence is ordered by ascending timestamps, so for every i, $t_i \leq t_{i+1}$.

As before, there is a set AP of atomic propositions and a labeling function $L : \Sigma \to 2^{AP}$ that assigns to each state $s \in \Sigma$ the atomic propositions that are true in that state.

Definition 2.9 (Syntax of MTL) [Ho et al. 2014]

Let $I \subseteq [0, \infty)$ be an interval in T, either of the form $[a, b]$, $[a, b)$, $(a, b]$, or (a, b), and $p \in AP$ an atomic proposition. The set of MTL formulas is inductively defined by the following grammar:

$$\varphi := \mathsf{true} \mid p \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \varphi_1 \,\mathsf{U}_I\, \varphi_2$$

The temporal operator $\varphi_1 \,\mathsf{U}_I\, \varphi_2$ now expresses that φ_2 holds within some time interval I (relative to the time of the current state being evaluated), and φ_1 holds from the current state until then. Thus, the right side of the operator is constrained by the interval, but the left side is not. The *unconstrained until* operator $\mathsf{U}_{[0,\infty)}$ is abbreviated as U.

The *finally* and *globally* operators can again be derived from *until*, and now also carry a time interval:

$$\mathsf{F}_I\, \varphi \overset{\text{def}}{=} \mathsf{true}\, \mathsf{U}_I\, \varphi$$
$$\mathsf{G}_I\, \varphi \overset{\text{def}}{=} \neg\, \mathsf{F}_I\, \neg\varphi$$

The notion of a "next" state is not as clear in a timed sequence as in a regular ordered sequence: there may be multiple states with the same timestamp, and their relative ordering is often not as relevant in timed systems as it is in non-timed systems. Thus, the *next* operator is often left out, but in an application of MTL in which the ordering of states with identical timestamps is well-defined, the unconstrained *next* operator or the constrained *next* operator X_I, which additionally specifies that the next state occurs within the interval I, can be included.

The semantics of MTL for infinite timed sequences is defined as follows [Ho et al. 2014].

Definition 2.10 (Semantics of MTL on infinite sequences) [Ho et al. 2014]

Let $\rho = (s_0, t_0), \ldots$ be an infinite timed sequence, $p \in AP$ an atomic proposition, $I \subseteq [0, \infty)$ an interval, φ, φ_1, and φ_2 MTL formulas, and i a position with $i \geq 0$. The satisfaction relation $\models$ is defined as

$$\begin{array}{lll}
\rho^i \models \mathsf{true} & & \\
\rho^i \models p & \text{iff} & p \in L(s_i) \\
\rho^i \models \varphi_1 \wedge \varphi_2 & \text{iff} & \rho^i \models \varphi_1 \text{ and } \rho^i \models \varphi_2 \\
\rho^i \models \neg\varphi & \text{iff} & \rho^i \not\models \varphi \\
\rho^i \models \varphi_1\, \mathsf{U}_I\, \varphi_2 & \text{iff} & \exists_{j \geq i}[\rho^j \models \varphi_2 \text{ and } t_j - t_i \in I \text{ and } \forall_{i \leq k < j}\, \rho^k \models \varphi_1]
\end{array}$$

Weak and strong semantics of MTL, in the same vein as LTL$^{\mp}$, are defined as follows.

Definition 2.11 (Semantics of MTL on truncated sequences) [Ho et al. 2014]

Let $\rho = (s_0, t_0), \ldots, (s_n, t_n)$ be a finite timed sequence of length $n + 1$, $p \in AP$ an atomic proposition, $I \subseteq [0, \infty)$ an interval of the form $[\inf(I), \sup(I))$, φ, φ_1, and φ_2 MTL formulas, and i a position with $0 \leq i \leq n$. The weak satisfaction relation $\models^-$ is defined as

$$\begin{array}{lll}
\rho^i \models^- \mathsf{true} & & \\
\rho^i \models^- p & \text{iff} & p \in L(s_i) \\
\rho^i \models^- \varphi_1 \wedge \varphi_2 & \text{iff} & \rho^i \models^- \varphi_1 \text{ and } \rho^i \models^- \varphi_2 \\
\rho^i \models^- \neg\varphi & \text{iff} & \rho^i \not\models^+ \varphi \\
\rho^i \models^- \varphi_1\, \mathsf{U}\, \varphi_2 & \text{iff} & \exists_{i \leq j \leq n}[\rho^j \models^- \varphi_2 \text{ and } t_j - t_i \in I \text{ and } \forall_{i \leq k < j}\, \rho^k \models^- \varphi_1] \\
& & \text{or } [t_n - t_i < \sup(I) \text{ and } \forall_{i \leq k \leq n}\, \rho^k \models^- \varphi_1]
\end{array}$$

and the strong satisfaction relation $\models^+$ is defined as

$$\begin{array}{lll}
\rho^i \models^+ \mathsf{true} & & \\
\rho^i \models^+ p & \text{iff} & p \in L(s_i) \\
\rho^i \models^+ \varphi_1 \wedge \varphi_2 & \text{iff} & \rho^i \models^+ \varphi_1 \text{ and } \rho^i \models^+ \varphi_2 \\
\rho^i \models^+ \neg\varphi & \text{iff} & \rho^i \not\models^- \varphi \\
\rho^i \models^+ \varphi_1 \mathsf{U}_I \varphi_2 & \text{iff} & \exists_{i \le j \le n} [\rho^j \models^+ \varphi_2 \text{ and } t_j - t_i \in I \text{ and } \forall_{i \le k < j}\, \rho^k \models^+ \varphi_1]
\end{array}$$

In the second condition of the weak semantics in Definition 2.11, the right endpoint of I is assumed to be open and finite. The semantics can easily be extended to any open, closed or half-open intervals: if the right endpoint of I is closed, the expression $t_n - t_i < \sup(I)$ is replaced by $t_n - t_i \le \sup(I)$, and if the upper bound of I is ∞, it is always true.

An LTL formula (assuming it does not contain X) can be expressed in MTL simply by taking $[0, \infty)$ for all intervals. An untimed sequence can be represented as a timed sequence by setting $T = \mathbb{N}$, and for every state s_i the timestamp $t_i = i$ is the position of the state in the sequence.

A variant of MTL called MITL, Metric Interval Temporal Logic [Alur et al. 1996], is the subset of MTL where the intervals are required to be non-singleton. That is, the left and right endpoints of the intervals are required to be distinct so that the interval covers more than a single time point. This restriction is necessary in order to make some verification algorithms decidable [Ouaknine and Worrell 2008].

2.1.7 Continuous signals and STL

MTL as defined above has semantics defined on sequences $\rho = (s_0, t_0), (s_1, t_1), \ldots$ where every element $s_i \in \Sigma$ corresponds to a set $L(s_i) \in 2^{AP}$ of atomic propositions that hold at time $t_i \in T$.

In the context where the sequence encodes the behavior of a program, we could see (s_i, t_i) as an event corresponding to a change in the state of the program, and say that the program is in that state from time t_i until the next state change at time t_{i+1}. In this interpretation, a state really describes the program in the interval on the number line from t_i up to t_{i+1}, but because we only ever evaluate the state at the points in T that occur in the sequence, this is equivalent to the interpretation that a state relates to a point on the number line. The interpretation of MTL over sequences of states or events is also called the *point-wise semantics* [Ouaknine and Worrell 2008].

An alternative interpretation of MTL, called the *continuous semantics* [Ouaknine and Worrell 2008], interprets specifications not over a sequence of discrete timed states or events, but continuously over time using a function $f : \mathbb{R}_+ \to 2^{AP}$ mapping every (non-negative real-numbered) time value $t \in \mathbb{R}_+$ to the set $f(t)$ of propositions that hold at time t. Given an MTL formula φ over the set of atomic propositions AP, the satisfaction relation $f \models \varphi$ is now defined for every point on the positive number line with the following rule for the *until* operator (and the rest omitted for brevity), where given some t, f^t denotes the signal $f^t(s) = f(t + s)$:

$$f \models \varphi_1 \mathsf{U}_I \varphi_2 \;\text{ iff }\; \exists_{t \in I} [f^t \models \varphi_2 \text{ and } \forall_{u \in [0,t)}\, f^u \models \varphi_1]$$

Verification of MTL formulas on continuous signals requires different algorithms than that of sequences, and decision problems involving the continuous semantics are often undecidable or very complex. One way to solve this is by *discretizing* the trace through sampling of the signal, and using algorithms for verifying discrete-time MTL specifications to obtain a (possibly incomplete) verdict [Furia and Rossi 2006].

STL, Signal Temporal Logic [Maler and Nickovic 2004], is a temporal logic based on MTL with continuous semantics for the verification of real-valued continuous signals. Formulas in STL use the syntax of MTL, with atomic propositions of the form $s \leq x$ or $s \geq x$, with $s : \mathbb{R} \to \mathbb{R}$ being a continuous function and $x \in \mathbb{R}$ a threshold value.

An example property for some signal x is $\mathsf{F}_{[0,10)}(x \geq 0.5)$, which states that at some point within 10 time units the value of x is larger than 0.5.

2.2 Causality

In runtime verification, we are interested in detecting whether the recorded behavior of a system violates a specification. However, once we have detected a violation, in order to proceed to fix the problem or refine the specification we need to understand *why* the system has violated the specification. For simple properties and short traces, it might be easy to locate the error and understand how it arises intuitively. In more complicated situations, however, finding the cause of a verification result is not trivial.

This section presents a theoretical background for the application of causality to verification. It is based on a mathematical formulation of causality by Halpern and Pearl [2005], called HP causality, presented in Section 2.2.2. Section 2.2.3 present some subsequent results on HP causality. Section 2.2.4 is a short review of how causality is used (formally or informally) in other areas of verification and software testing.

2.2.1 Formalizing causality

Two notions of causality can be distinguished: type causality (concerning general statements such as “smoking causes cancer”) and actual causality (focusing on particular events: “the fact that David smoked like a chimney for 30 years caused him to get cancer last year”) [Halpern 2015]. The latter is where interest in the use of causality in verification has focused.

Finding a good definition for actual causality has proved difficult. Going back to the philosopher Hume [1739] and more recently Lewis [1973], definitions of causality have involved counterfactuality: statements about situations that run counter to fact. In the sentence “if A had not happened, B would not have happened”, we would say that B has a counterfactual dependence on A, and the main definitions of causality are inspired by this relationship. Unfortunately, this notion is not enough to capture all the subtleties involved with causality, and the literature concerning causality tells a story of publications attempting to formulate a formal definition of causality and publications providing counterexamples that are not correctly analyzed by the definitions.

To see that counterfactual dependence is not enough to define causality, consider the following story taken from Hall [2004]:

> Suzy and Billy both pick up rocks and throw them at a bottle. Suzy’s rock gets there first, shattering the bottle. Since both throws are perfectly accurate, Billy’s would have shattered the bottle had it not been preempted by Suzy’s throw.

Here, using naive counterfactual reasoning, one would conclude that Suzy did not cause the bottle to shatter, since if she had not thrown her rock, the bottle would have shattered anyway. However, most people would say that Suzy’s throw should be a cause of the bottle shattering in this situation. Formally analyzing a situation such as this so that it is consistent with our intuitions depends on how exactly the situation is modeled and how causality itself is formalized, and no single method and definition has been found that fits all situations.

A definition of causality that is most widely used in the domain of verification is one formulated by Halpern and Pearl, often shortened to *HP causality.* It comes in three versions: an original version described in [Halpern and Pearl 2001], an updated version in the journal version of that paper [Halpern and Pearl 2005], and a modified version [Halpern 2015]. In the domain of formal verification, the 2005 updated definition finds the most use, so we will examine it in more detail.

2.2.2 HP causality

In HP causality, the world is modeled using a collection of variables, divided into *exogenous variables* (variables that are determined by factors outside the model, and whose values we must assume) and *endogenous variables* (variables within the model whose values are determined by other endogenous and exogenous variables). How the endogenous variables take their values is described by a set of *structural equations*.

Definition 2.12 (Causal model) [Halpern and Pearl 2005]

A causal model M is a pair $(\mathcal{S}, \mathcal{F})$.

$\mathcal{S}$ is the *signature*, which is a tuple $\mathcal{S} = (\mathcal{U}, \mathcal{V}, \mathcal{R})$, where $\mathcal{U}$ is the set of exogenous variables, $\mathcal{V}$ is the set of endogenous variables and $\mathcal{R}$ is a function that associates every variable with the range of possible values for that variable.

$\mathcal{F}$ encodes the structural equations of the variables in the model. It is a function that associates every endogenous variable $X \in \mathcal{V}$ with a function F_X such that $F_X : (\times_{U \in \mathcal{U}} \mathcal{R}(U)) \times (\times_{Y \in \mathcal{V} - \{X\}} \mathcal{R}(Y)) \to \mathcal{R}(X)$. That is, for every endogenous variable X, F_X determines the value of X given the values of the other variables.

The variables in $\mathcal{U}$, which are used to model variables that we need to assume, are assigned values from a *context* $\vec{u}$ (denoted $\mathcal{U} \leftarrow \vec{u}$).

Given a model M, a vector $\vec{X}$ of variables in $\mathcal{V}$, a vector $\vec{x}$ of values for the variables in $\vec{X}$, and a context $\vec{u}$, we can assign the values $\vec{x}$ to $\vec{X}$ (denoted $\vec{X} \leftarrow \vec{x}$) to obtain a new causal model $M_{\vec{X} \leftarrow \vec{x}}$ where the value of every variable $X \in \vec{X}$ is set to the corresponding value $x \in \vec{x}$ by removing the variables $\vec{X}$ from $\mathcal{V}$ and replacing the occurrences of the variables in $\vec{X}$ in $\mathcal{F}$ with the values in $\vec{x}$.

The dependencies between the variables in M can be described using a *causal network*: a directed graph with nodes corresponding to the variables in $\mathcal{V}$ and an arc from one variable to another if the value of the second variable depends on the value of the first. If the causal network is a connected directed acyclic graph, then given a context $\vec{u}$, there is a unique solution for all the equations. This corresponds to there being at least one total ordering $\sqsubseteq$ of $\mathcal{V}$ such that if $X \sqsubseteq Y$, then X is independent of Y (but Y may depend on X). To find the unique solution of the equations, we simply solve for the variables in the order given by one such $\sqsubseteq$.

The definition of causality is expressed using a structure model in a language of causal formulas and events.

Definition 2.13 (Causal events and formulas) [Halpern and Pearl 2005]

Let $M = (\mathcal{S}, \mathcal{F})$ be a causal model with $\mathcal{S} = (\mathcal{U}, \mathcal{V}, \mathcal{R})$ a signature.

A *primitive event* is a formula of the form $X = x$, for $X \in \mathcal{V}$ and $x \in \mathcal{R}(X)$.

A *basic causal formula* is a formula of the form $[Y_1 \leftarrow y_1, \ldots, Y_k \leftarrow y_k]\eta$ where η is a Boolean combination of primitive events, $Y_1, \ldots, Y_k$ are endogenous variables in $\mathcal{V}$ and $y_i \in \mathcal{R}(Y_i)$ for all $i = 1, \ldots, k$. It is abbreviated $[\vec{Y} \leftarrow \vec{y}]\eta$, or simply η if $k = 0$. It holds if η holds in $M_{\vec{Y} \leftarrow \vec{y}}$. A *causal formula* is a Boolean combination of basic causal formulas.

We write $(M, \vec{u}) \models \psi$ if the causal formula ψ is true in the model M given context $\vec{u}$.

To summarize the above, $(M, \vec{u}) \models [\vec{Y} \leftarrow \vec{y}](X = x)$ means that the variable X has value x in the unique solution to the equations in the model that is obtained by setting the values of the variables in $\vec{Y}$ to $\vec{y}$ in the model M given context $\vec{u}$.

Using this language, we can formally define what it means for one event to cause another.

As a warm-up, we can express a *but-for cause*, or a counterfactual dependence of a combination of events η on an event $X = x$, using the language of causal models as follows:

> **Definition 2.14 (But-for cause)** [Halpern and Pearl 2005]
>
> $X = x$ is a *but-for cause* of η in $(M, \vec{u})$ if:
>
> – $(M, \vec{u}) \models X = x$ and $(M, \vec{u}) \models \eta$, and
>
> – There exists a setting x' such that $(M, \vec{u}) \models [X \leftarrow x']\neg\eta$

In other words, $X = x$ is a but-for cause of η if $X = x$ is true and η is true, but if X were set to some other value x' then η would not be true.

The definition of an actual cause according to HP causality is as follows:

> **Definition 2.15 (Cause – updated definition)** [Halpern and Pearl 2005]
>
> $\vec{X} = \vec{x}$ is a *cause* of η in $(M, \vec{u})$ if the following conditions hold:
>
> AC1. $(M, \vec{u}) \models \vec{X} = \vec{x}$ and $(M, \vec{u}) \models \eta$.
>
> AC2. There exists a partition of $\mathcal{V}$ into disjoint subsets $\vec{Z}$ and $\vec{W}$ with $\vec{X} \subseteq \vec{Z}$ and some setting $\vec{x}'$ and $\vec{w}'$ of the variables in $\vec{X}$ and $\vec{W}$ such that
>
> (a) $(M, \vec{u}) \models [\vec{X} \leftarrow \vec{x}', \vec{W} \leftarrow \vec{w}']\neg\eta$. That is, changing $\vec{X}, \vec{W}$ from $\vec{x}, \vec{w}$ to $\vec{x}', \vec{w}'$ changes η from true to false.
>
> (b) Let $\vec{z}$ be such that $(M, \vec{u}) \models \vec{Z} = \vec{z}$. For all subsets $\vec{W}^* \subseteq \vec{W}$ and $\vec{Z}^* \subseteq \vec{Z}$ we have $(M, \vec{u}) \models [\vec{X} \leftarrow \vec{x}, \vec{W}^* \leftarrow \vec{w}', \vec{Z}^* \leftarrow \vec{z}]\eta$.[3] That is, setting any subset of $\vec{W}$ to the values in $\vec{w}'$ has no effect on η as long as $\vec{X}$ has the value $\vec{x}$, even if all the variables in any subset of $\vec{Z}$ are set to their original values in the context $\vec{u}$.
>
> AC3. $\vec{X}$ is minimal, that is, no proper subset of $\vec{X}$ satisfies AC2.

The definition can be explained informally as follows, taken from [Halpern and Pearl 2005; Chockler et al. 2008].

AC1 specifies that A cannot be a cause of B unless both A and B are true. AC3 is a minimality condition to prevent, for example, Suzy throwing the rock and sneezing from being the cause of the bottle shattering.

The core of the definition lies in AC2. The variables in $\vec{Z}$ should be thought of as describing the "active causal process" from $\vec{X}$ to η: they take part in the propagation of values from $\vec{X}$ to η in the structural equations.

[3]There is a slight abuse of notation here, since the subset $\vec{W}^*$ has fewer variables than $\vec{w}'$ has values. $\vec{W}^* \leftarrow \vec{w}'$ here means that every $W \in \vec{W}^*$ is assigned the value $w \in \vec{w}'$ corresponding to that variable in $\vec{W}$, and the values corresponding to the variables not in $\vec{W}^*$ are ignored, and similarly for $\vec{Z}^* \leftarrow \vec{z}$.

AC2(a) is reminiscent of a but-for cause, but is more permissive; it allows the dependence of η on $\vec{X}$ to be tested under some special *structural contingencies*, in which the variables $\vec{W}$ are held constant at some setting $\vec{w}'$.

AC2(b) is an attempt to counteract the "permissiveness" of AC2(a) with regard to structural contingencies. Essentially, it ensures that $\vec{X}$ alone suffices to bring about the change from η to $\neg\eta$; setting $\vec{W}$ to $\vec{w}'$ merely eliminates spurious side effects that tend to mask the action of $\vec{X}$.

To illustrate the above definitions, let us look at an example.

Example 2.16 [Halpern and Pearl 2005]

Remember the story about Suzy and Billy from [Hall 2004]:

> Suzy and Billy both pick up rocks and throw them at a bottle. Suzy's rock gets there first, shattering the bottle. Since both throws are perfectly accurate, Billy's would have shattered the bottle had it not been preempted by Suzy's throw.

We define a model M with five endogenous boolean variables:

- ST for "Suzy throws"
- BT for "Billy throws"
- BH for "Billy's rock hits the (intact) bottle"
- SH for "Suzy's rock hits the bottle"
- BS for "Bottle shatters"

We assume that the context contains variables that determine that ST and BT are both *true* and that Suzy throws first. We define $\mathcal{F}$ such that it encodes the following equations (considering only the endogenous variables for simplicity):

- $SH = ST$
- $BH = BT \wedge \neg SH$
- $BS = SH \vee BH$

Now, to determine whether Suzy's throw ($ST = \mathit{true}$) is a cause of the bottle shattering ($BS = \mathit{true}$), we take $\vec{X} = \{ST\}$, $\vec{x} = \{\mathit{true}\}$, and $\eta = (BS = \mathit{true})$. We see that $(M, \vec{u}) \models ST = \mathit{true} \wedge BS = \mathit{true}$, so AC1 holds. AC3 is trivial. To satisfy AC2, we choose $\vec{W} = \{BT\}$ and $\vec{w} = \{\mathit{false}\}$, and $\vec{Z} = \{ST, SH, BH\}$. If we set both ST and BT to *false* and solve the resulting structural equations, we see that BS is no longer *true*. Both AC2(a) and AC2(b) hold with this choice of $\vec{W}$ and $\vec{Z}$, so we conclude that $ST = \mathit{true}$ is a cause for $BS = \mathit{true}$.

Next, to determine whether Billy's throw ($BT = \mathit{true}$) is a cause of the bottle shattering ($BS = \mathit{true}$), we take $\vec{X} = \{BT\}$, $\vec{x} = \{\mathit{true}\}$, and $\eta = (BS = \mathit{true})$. AC1 and AC3 still hold. For, AC2, attempting the symmetric choice of $\vec{W} = \{ST\}$, $\vec{w} = \{\mathit{false}\}$, and $\vec{Z} = \{BT, BH, SH\}$, we see that AC2(b) is violated when we take $\vec{Z}^* = \{BH\}$. If BH is kept at its original value of *false*, we see that even when Suzy's rock would not hit the bottle, Billy's throw does not cause the bottle to shatter. There is no other partition into $\vec{W}$ and $\vec{Z}$ that does satisfy AC2, so we conclude that $BT = \mathit{true}$ is not a cause for $BS = \mathit{true}$.

These conclusion are consistent with our intuition that Suzy broke the bottle, and Billy did not. This example also shows that the choice of how we model the situation is important in how causality is determined. If we had taken a simpler model with only the variables BT, ST, BS, and had no variables that captured the fact that Billy's rock did not hit an intact bottle, we would have concluded that both Billy's and Suzy's throws had caused the bottle to shatter.

The modified definition of Halpern [2015] is simpler, and focuses on variables that are *frozen* in their original values, rather than considering all possible contingencies. In some of the problematic cases that have been raised since the updated definition was published, the modified definition tends to match people's intuitions better. However, existing work on the applications of causality to formal verification has not (yet) been updated to use this new definition.

2.2.3 HP causality for Boolean circuits

HP causality in its full generality is quite subtle, but instances are often able to make assumptions that result in much simpler definitions.

In [Chockler et al. 2008], an instance of HP causality is presented for Boolean circuits. A Boolean circuit is a representation of a Boolean propositional formula where the leaves represent atomic propositions (inputs) and the interior nodes represent the binary operations $\neg$, $\wedge$, and $\vee$ (gates). Without loss of generality, the formulas are assumed to be in *positive normal form*, so that negations occur only in the level directly above the leaves.

Let $g : \{0,1\}^n \to \{0,1\}$ be a Boolean function on n variables, let $\mathcal{C}$ be a Boolean circuit that computes g, and let $\vec{X}$ be the set of variables of $\mathcal{C}$. A truth assignment f to the set $\vec{X}$ is a function $f : \vec{X} \to \{0,1\}$. The value of a gate w of $\mathcal{C}$ under an assignment f is defined as the value of the function of this gate under the same assignment. For an assignment f and a variable X, we denote by $\tilde{f}_X$ the truth assignment that differs from f in the value of X: $\tilde{f}_X(Y) = f(Y)$ for all $Y \neq X$, and $\tilde{f}_X(X) = \neg f(X)$. Similarly, for a set $\vec{Z} \subseteq \vec{X}$, $\tilde{f}_{\vec{Z}}$ is the truth assignment that differs from f in the values of variables in $\vec{Z}$.

Boolean circuits are a special case of binary causal models, where each gate of the circuit is a variable of the model ($\mathcal{V}$), and values of inner gates are computed based on values of the inputs to the circuits and the Boolean functions of the gates ($\mathcal{F}$). A context $\vec{u}$ (for $\mathcal{U}$) is a setting to the input variables of the circuit. For ease of presentation, an explicit notion of *criticality* is defined, which captures the notion of counterfactual causal dependence:

> **Definition 2.17 (Criticality for Boolean circuits)** [Chockler et al. 2008]
>
> Consider a Boolean circuit $\mathcal{C}$ over the set $\vec{X}$ of variables, an assignment f, a variable $X \in \vec{X}$, and a gate w of $\mathcal{C}$. We say that X is *critical* for w under f if $\tilde{f}_X(w) = \neg f(w)$.

If a variable X is critical for the output gate of a circuit $\mathcal{C}$, changing the value X alone causes a change in the value of $\mathcal{C}$, and otherwise changing the value of that variable alone does not. However, it may be the case that changing the value of X together with several other variables causes a change in the value of $\mathcal{C}$. The definition of a *cause* can be rewritten for Boolean circuits as follows:

> **Definition 2.18 (Causality for Boolean circuits)** [Chockler et al. 2008]
>
> Consider a Boolean circuit $\mathcal{C}$ over the set $\vec{X}$ of variables, an assignment f, a variable $X \in \vec{X}$, and a gate w of $\mathcal{C}$. A (possibly empty) set $\vec{Z} \subseteq \vec{X} \backslash \{X\}$ *makes X critical for w* if $\tilde{f}_{\vec{Z}}(w) = f(w)$ and X is critical for w under $\tilde{f}_{\vec{Z}}$. (The value of) X is a *cause* for (the value of) w if there is some $\vec{Z}$ that makes X critical for w.

Eiter and Lukasiewicz [2002] showed that testing causality in general causal models is Σ_2^{P}-complete (Σ_2^{P} is a superset of NP), and that testing causality in binary causal models is NP-complete. Chockler et al. [2008] concluded that testing causality in Boolean circuits is NP-complete too.

2.2.4 Causality in verification

[Beer et al. 2012] presents a method for finding causes for the violation of LTL formulas in counterexample paths generated by a model checker by formulating the problem as an instance of HP causality. A cause for the violation of an LTL formula is defined as a state in the path combined with an atomic proposition in the formula such that some criteria inspired by the notions of criticality and cause shown in Section 2.2.3 hold. An efficient approximate algorithm is used to find the approximate set of causes for a single violation of the formula. It is explained in detail in Section 6.1.

In [Leitner-Fischer and Leue 2013] a method is presented for generating causes for the violation of LTL-definable safety properties in transition system models, based on a causality definition inspired by HP causality. Causes in this approach are expressed using a logic called event order logic, and defined so that they can capture – in addition to the occurrence of events – the relative ordering and non-occurrence of events. Because of this, the causes found using this approach are more expressive and often more accurate than those of [Beer et al. 2012], but they require knowledge of the entire transition system, rather than only a single execution path. The method is similar in structure to explicit state model checking, and is implemented in the tool SpinCause [Leitner-Fischer and Leue 2014]. In [Beer et al. 2015] this approach is implemented using bounded model checking in a SAT-solver. The approach is extended to general LTL properties in [Caltais et al. 2019].

In [Dubslaff et al. 2022] causes for the satisfaction or violation of functional and non-functional requirements are expressed as propositional logic formulas representing configurations of features of configurable software systems. The approach is inspired by HP causality, and generalizes to arbitrary Boolean functions outside the feature-oriented software engineering domain.

In [Baier et al. 2022] the notions of *necessary* and *sufficient causes* for reachability properties of transition systems, where both causes and effects are (sets of) states in transition systems, are characterized using LTL. Additionally, quantitative measures of degree of necessity and sufficiency are defined. These notions of causality (which are orthogonal to HP causality) are elegantly expressed using the language of LTL, and their quantitative counterparts offer more flexibility in the selection of good causes. However, the approach is restricted to reachability properties expressed as a set of states in the transition system, rather than general LTL-definable properties, and consequently, causes do not directly indicate individual propositions or variables.

Baier et al. [2021] provide an overview of further instances in which causal reasoning is used in verification. In their view, general principles of causality have been used, often implicitly, in formal verification for a long time. For example, in *program slicing* [Harman and Hierons 2001], an approximation of an actual cause of a software failure is found by deleting those parts of a program that have no effect on the outcome.

Causality is also a key concept in *error localization*, the problem of reducing a counterexample path for ease of debugging [Baier et al. 2021]. For example, in [Zeller 2002] the *Delta Debugging* algorithm is used to isolate the relevant states of a program (containing the variables and their values at some time during execution) that cause a failure. It requires two executions of a program, one where the failure occurs and one where it does not, and attempts to isolate a cause for the failure that is as precise as possible, which they call the cause-effect chain.

In [Ball et al. 2003], [Renieris and Reiss 2003], and [Groce 2004] the cause of an error in a counterexample path (e.g. from model checking) or a failing test is localized using the existence of correct sequences or by deriving alternative sequences that do not contain the error.

In [Wang et al. 2006] the cause of an error in a model checking counterexample is explained by using a weakest pre-condition algorithm to find a minimal set of predicates that explains why the execution fails.

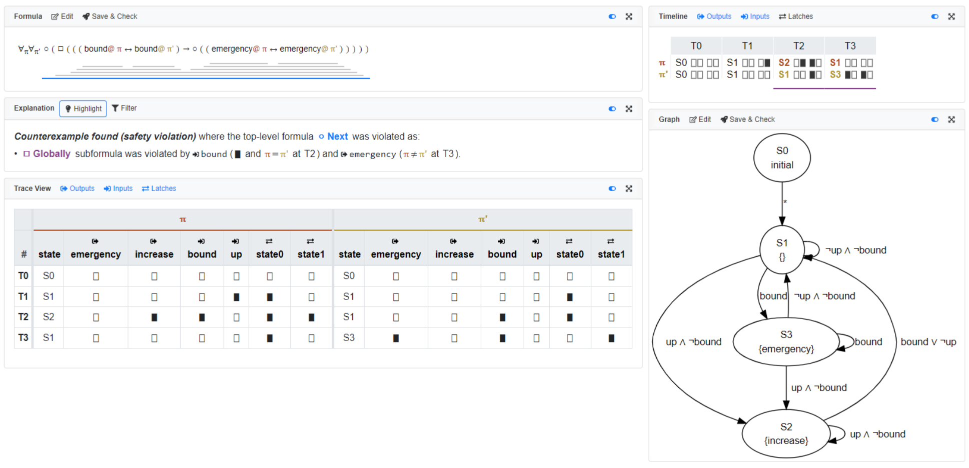
2.3 Verification explanation tools

A review of tools that visualize or explain results of formal verification is presented.

HyperVis [Horak et al. 2022; Horak et al. 2023] is a tool for visualizing model checking counterexamples of hyperproperties (properties relating different executions of one model).

The system is specified as logical circuit, and the hyperproperty is specified as an LTL formula in the form $\forall\pi\forall\pi'(\dots)$ where π and π' are execution paths, and a variable v is used as $v@\pi$ and $v@\pi'$ for its value in either path. HyperVis looks for a counterexample consisting of two paths π and π' that together violate the property.

The visualization consist of four interacting parts: the system graph, the trace view, the formula and the explanation. The system is visualized as a transition graph. The two paths are shown as two traces with all variables at each time step. A textual explanation of the counterexample is given that shows which subformulas are violated by which values. The relevant subformulas and the corresponding causal values given different colors, and the user can hover over the parts of the formula, explanation, trace and graph to highlight them in each of the four views. The trace can be filtered to only show the relevant variables, and the system graph can be filtered to only show the relevant states.



$\forall_\pi\forall_{\pi'} \circ (\Box (((\text{bound}@\pi \leftrightarrow \text{bound}@\pi') \rightarrow \circ ((\text{emergency}@\pi \leftrightarrow \text{emergency}@\pi')))))$

Counterexample found (safety violation) where the top-level formula ○ **Next** was violated as:

- □ **Globally** subformula was violated by bound (■ and $\pi=\pi'$ at T2) and emergency ($\pi\neq\pi'$ at T3).

	π							π'						
#	state	emergency	increase	bound	up	state0	state1	state	emergency	increase	bound	up	state0	state1
T0	S0	□	□	□	□	□	□	S0	□	□	□	□	□	□
T1	S1	□	□	□	■	■	□	S1	□	□	□	□	■	□
T2	S2	□	■	■	□	■	■	S1	□	□	■	□	■	□
T3	S1	□	□	□	□	■	□	S3	■	□	■	□	□	■

	T0	T1	T2	T3
π	S0 □□ □□	S1 □□ □■	S2 □■ ■□	S1 □□ □□
π'	S0 □□ □□	S1 □□ □□	S1 □□ ■□	S3 ■□ ■□

Figure 2.1: HyperVis

RuleBase PE [Beer et al. 1996] is the tool in which the counterexample explanation algorithm of Beer et al. [2012] is implemented. This algorithm is presented in detail in Section 6.1.

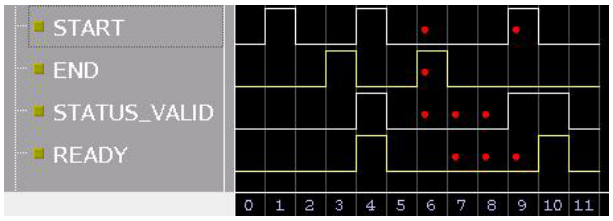
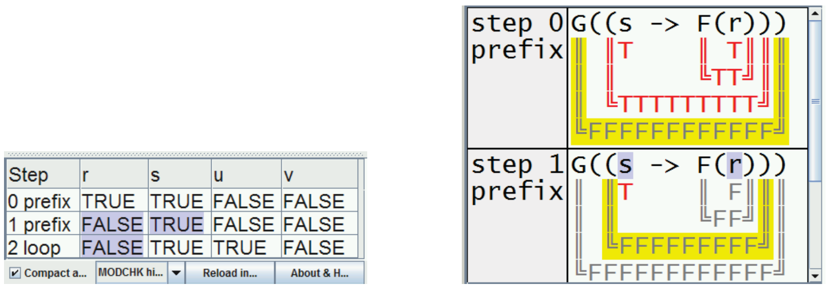


Figure 2.2: Counterexample explanation in RuleBase PE

MODCHK [Pakonen et al. 2018] is a model checking tool for systems expressed as functional block diagrams which offers counterexample visualization.

It extends the causality algorithm from RuleBase PE. It shows *enhanced atomic causes*, by adding to the atomic causes of Beer et al. [2012] information about where in the formula the propositions are important. It also allows the user to interactively select counterexample explanations of subformulas. Clicking on a value annotation of a (sub)formula shows the *immediate cause* of that subformula value, allowing the user to iteratively descend to the (enhanced) atomic causes.



Step	r	s	u	v
0 prefix	TRUE	TRUE	FALSE	FALSE
1 prefix	FALSE	TRUE	FALSE	FALSE
2 loop	FALSE	TRUE	TRUE	FALSE

```
step 0 G((s -> F(r)))
prefix    T       T
               TT
          TTTTTTTTTT
        FFFFFFFFFFFF
step 1 G((s -> F(r)))
prefix    T       F
               FF
          FFFFFFFFF
        FFFFFFFFFFFF
```

Figure 2.3: Two counterexample visualization views of MODCHK

OERITTE is another tool for visualizing and explaining model checking counterexamples of functional block diagrams. Explanations are given in terms of both the LTL formula and the model in the form of paths from causes to target values [Ovsiannikova et al. 2021].

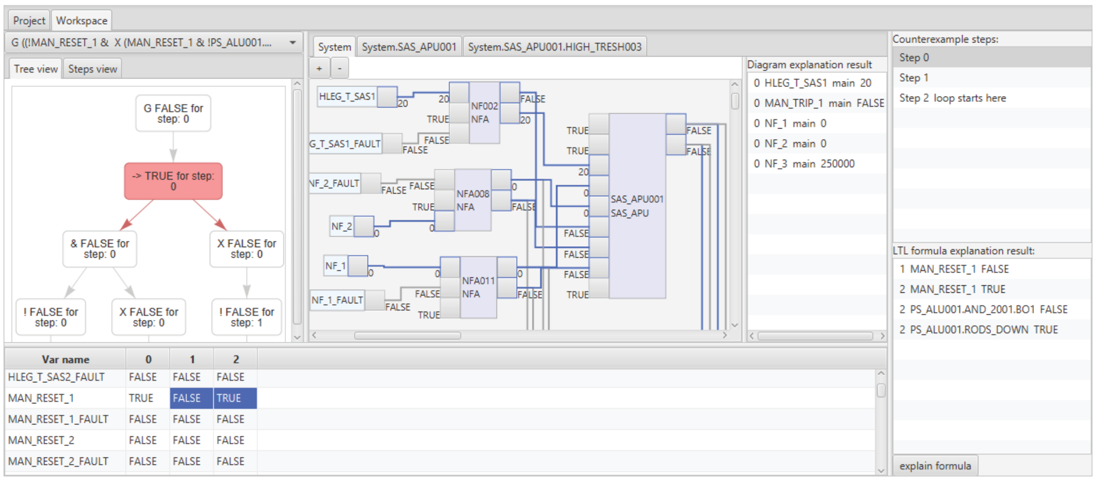


Diagram explanation result
0 HLEG_T_SAS1 main 20
0 MAN_TRIP_1 main FALSE
0 NF_1 main 0
0 NF_2 main 0
0 NF_3 main 250000

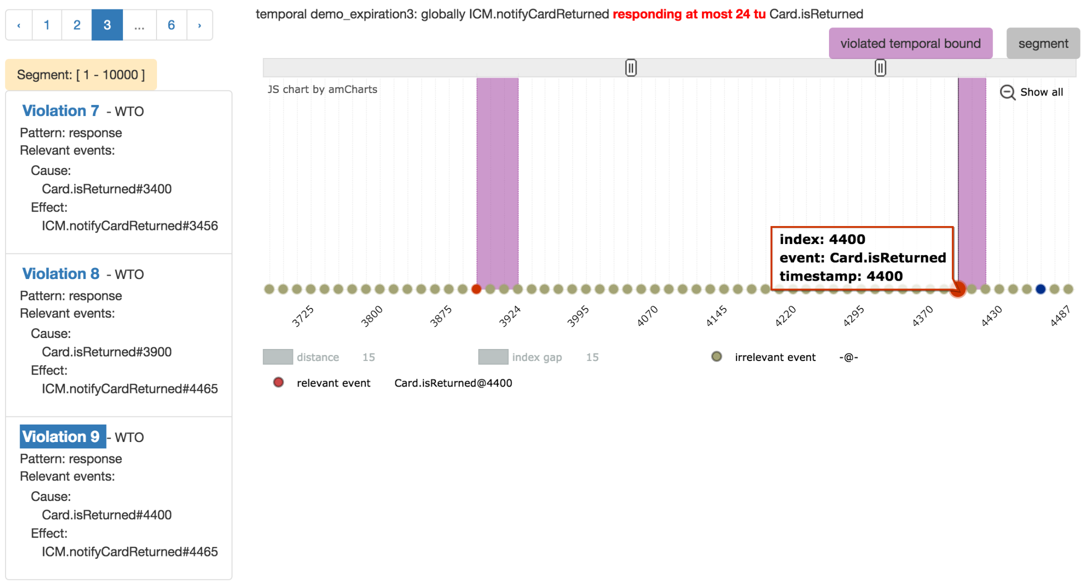
Counterexample steps:
Step 0
Step 1
Step 2 loop starts here

LTL formula explanation result:
1 MAN_RESET_1 FALSE
2 MAN_RESET_1 TRUE
2 PS_ALU001.AND_2001.BO1 FALSE
2 PS_ALU001.RODS_DOWN TRUE

Var name	0	1	2
HLEG_T_SAS2_FAULT	FALSE	FALSE	FALSE
MAN_RESET_1	TRUE	FALSE	TRUE
MAN_RESET_1_FAULT	FALSE	FALSE	FALSE
MAN_RESET_2	FALSE	FALSE	FALSE
MAN_RESET_2_FAULT	FALSE	FALSE	FALSE

Figure 2.4: OERITTE

TemPsy [Dou et al. 2018] is a temporal property specification language based on the temporal specification patterns of [Dwyer et al. 1999], with a corresponding visualization tool of property violations. Because the patterns of [Dwyer et al. 1999] are often able to more directly express a requirement than LTL, violations can be characterized as being due to unexpected occurrences, non-occurence, or wrong temporal order, etc., depending on the temporal pattern used in the property. In the visualization tool, each event that violates the property is highlighted and annotated with the reasons for the violation.


temporal demo_expiration3: globally ICM.notifyCardReturned **responding at most 24 tu** Card.isReturned

Segment: [1 - 10000]

Violation 7 - WTO
Pattern: response
Relevant events:
Cause:
Card.isReturned#3400
Effect:
ICM.notifyCardReturned#3456

Violation 8 - WTO
Pattern: response
Relevant events:
Cause:
Card.isReturned#3900
Effect:
ICM.notifyCardReturned#4465

Violation 9 - WTO
Pattern: response
Relevant events:
Cause:
Card.isReturned#4400
Effect:
ICM.notifyCardReturned#4465

index: 4400
event: Card.isReturned
timestamp: 4400

Figure 2.5: TemPsy visualization tool

3 TRACE4CPS

TRACE4CPS[1] is a tool for the visualization and analysis of execution traces for performance engineering of cyber-physical systems.

Traces in TRACE4CPS can have discrete elements (events and actions on resources) as well as continuous elements (real-valued signals over time), and can capture any level of abstraction and come from any domain. Examples of things that can be captured by the trace format are log files of software systems, executions of discrete-event simulation models, traces of scopes from physical machines, paths through a state transition graph in model checking, and performance logs of the CPU and memory usage of programs.

TRACE4CPS has functions for analyzing several aspects of the performance and behavior of the system that generated the trace, and allows for checking temporal properties on traces specified in a property language that combines MTL and STL.

3.1 Execution traces

An execution trace in TRACE4CPS consists of four types of elements [*TRACE4CPS User Manual* 2021]: events with timestamps, claims of resources with begin and end timestamps, dependencies between events or claims, and continuous signals. An element of an execution trace has a number of user-defined attributes with values. A trace is specified in a textual human-readable format in an ETF file.

A TRACE4CPS trace can be formally described as follows [Hendriks et al. 2023].

> **Definition 3.1 (Trace in TRACE4CPS)** [Hendriks et al. 2023]
>
> Let $\mathbf{A}$ be a set of attributes, $\mathbf{V}$ be a set of attribute values and $\mathbf{M} : \mathbf{A} \rightharpoonup \mathbf{V}$ be the set of partial functions from attributes to values. Each element of $\mathbf{M}$ is a value assignment to a subset of attributes. For example, the attribute mapping $m \in \mathbf{M}$ in which an attribute "name" has value "B" and attribute "id" has value "2" is notated as $m = \{name \mapsto B, id \mapsto 2\}$.
>
> An *event* is specified by a tuple (t, m) where $t \in \mathbb{R}$ is the timestamp of the event, and $m \in \mathbf{M}$ specifies the event's attributes.
>
> There is a set D of dependencies and a set R of resources, the details of which are not relevant to the verification process.
>
> A *claim* of a resource is specified (slightly simplified) by a tuple (t_0, t_1, r, m), where $t_0, t_1 \in \mathbb{R}$ (with $t_0 \leq t_1$) are the start and end time of the claim, $r \in R$ is the resource, and $m \in \mathbf{M}$ specifies the claim's attributes.

[1] https://www.eclipse.org/trace4cps/

> A claim $c = (t_0, t_1, r, \{v \mapsto a, \dots\})$ is associated with two events: $(t_0, \mathit{start}\ \{v \mapsto a, \dots\})$ and $(t_1, \mathit{end}\ \{v \mapsto a, \dots\})$.
>
> A *signal* is a tuple (f, m) where $f : \mathbb{R} \to \mathbb{R}$ is a piecewise second-order polynomial over some domain of the number line (the exact specification of which is out of the scope of this thesis), and $m \in \mathbf{M}$ specifies the signal's attributes.
>
> An *execution trace* is a tuple (E, D, R, C, S) where E is a set of events, D is a set of dependencies, R is a set of resources, C is a set of claims on R, and S is a set of signals. The start and end events of the claims in C are included in E.

3.2 Trace visualization

Traces are visualized in a Gantt chart, a diagram often used to display production schedules, where time is on the horizontal axis and the vertical axis is divided into a number of vertical sections called swimlanes. Events and claims are distributed across the swimlanes, and each signal is displayed in its own section. An event is displayed as an upwards-pointing arrow at the timestamp of the event. Claims are displayed as blocks with the left endpoint and the right endpoint at the times of the start and end timestamps of the claim.

The swimlanes can be defined by the user by specifying a grouping of claims and events. The user chooses which attributes are used to create groups, and a section is created for each value (or combination of values) of the chosen attributes. The user can also choose which attributes are used to decide the colors of the events and claims, so that every event or claim with the same combination of values for the chosen attributes gets the same color. Trace elements can be filtered according to user-specified filters over the values of the attributes.

Figure 3.1 shows the visualization of a (part of a) trace from a simulated image-processing system. It consists of a set of claims where each claim has an attribute *id* corresponding to the image being processed, and an attribute *name* corresponding to one of 7 image processing tasks (named A through G). Some tasks can be performed concurrently. In the interface of TRACE4CPS, the trace has been filtered to only show jobs 1, 2, and 3. The first image shows the default view, where every combination of attribute values is displayed in its own swimlane. In the second image, the attribute *id* was selected for coloring and the attribute *name* for grouping.

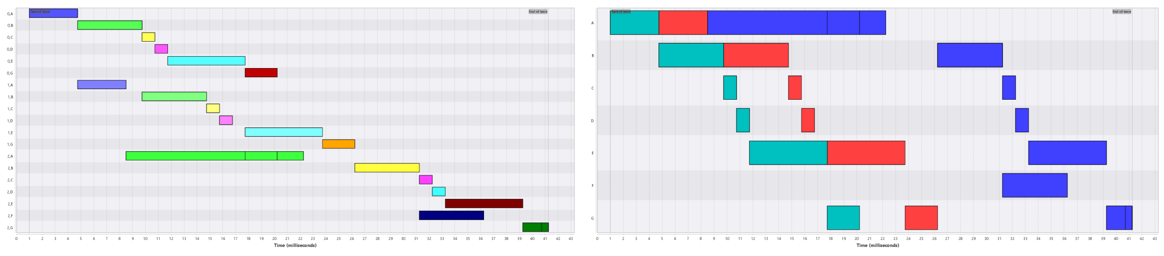


Figure 3.1: Visualization of an execution trace. Default view (left) and grouped by *name* and colored according to *id* (right)

3.3 Runtime verification

Properties in TRACE4CPS are written as formulas in a textual language in an ETL file. A *definition* is a named formula, and can be referred to in other formulas, which allows modularity and reuse of properties. Definitions can be parameterized, and the parameter can be used as an attribute value. A *check* is a top-level, named formula that is checked by TRACE4CPS when the ETL file is selected for verification on a trace. A check can contain a variable that quantifies over a range of integers, and that variable can be used as a parameter in a reference to a definition or as an attribute value.

Formulas, checks and definitions follow the following grammars:

Definition 3.2 Syntax of ETL formulas

$$\begin{aligned}\varphi \coloneqq\; & p \mid \texttt{not}\ \varphi \mid \texttt{if}\ \varphi_1\ \texttt{then}\ \varphi_2 \mid \texttt{(}\varphi_1\ \texttt{and}\ \varphi_2\texttt{)} \mid \texttt{(}\varphi_1\ \texttt{or}\ \varphi_2\texttt{)} \\ & \mid \texttt{globally}\ \varphi \mid \texttt{finally}\ \varphi \mid \texttt{during}\ I\ \varphi \mid \texttt{within}\ I\ \varphi \\ & \mid \texttt{until}\ \varphi_2\ \texttt{we have that}\ \varphi_1 \mid \texttt{by}\ I\ \varphi_1\ \texttt{and until then}\ \varphi_2 \\ & \mid \textit{identifier} \mid \textit{identifier}\ \texttt{(}\ \textit{parameter}\ \texttt{)}\end{aligned}$$

where $p \in AP$ (atomic propositions are elaborated further on), $I \subseteq [0, \infty)$ an open, closed or half-open interval with the left endpoint in $\mathbb{R}$ and the right endpoint in $\mathbb{R}$ or if it is open in the right endpoint in $\mathbb{R} \cup \{\infty\}$, and *identifier* refers to the name of a definition.

A definition is specified as follows:

$$\begin{aligned}\textit{definition} \coloneqq\; & \texttt{def}\ \textit{identifier}\ \texttt{:}\ \varphi \\ & \mid \texttt{def}\ \textit{identifier}\ \texttt{(}\ \textit{parameter}\ \texttt{)}\ \texttt{:}\ \varphi\end{aligned}$$

and a check as follows:

$$\begin{aligned}\textit{check} \coloneqq\; & \texttt{check}\ \textit{identifier}\ \texttt{:}\ \varphi \\ & \mid \texttt{check}\ \textit{identifier}\ \texttt{:}\ \texttt{forall}\ \texttt{(}\ \textit{parameter}\ \texttt{:}\ a\ \texttt{...}\ b\ \texttt{)}\ \varphi\end{aligned}$$

where φ is a formula.

There are three types of properties that can be expressed, depending on the type of atomic propositions used.

MTL. The first are MTL properties, where the atomic properties refer to attributes of events (including the start and end events corresponding to claims). The set AP of atomic properties is equal to the set $\mathbf{M}$ of attribute mappings. These properties are checked on the set of events E of a trace. An atomic proposition $p \in AP$ holds on an event (t_i, m_i) if and only if $p \subseteq m_i$. That is, p holds on (t_i, m_i) if and only if for every attribute $a \in \mathbf{A}$ that is defined in p, $p(a) = m_i(a)$.

STL. The second type are STL properties, where the atomic properties refer to the values of signals and are of the form $f \geq x$ or $f \leq x$ for a signal f and a threshold value $x \in \mathbb{R}$. A signal is either identified by their attributes or generated by TRACE4CPS. STL formulas are defined over the entire (continuous) interval in which the signals they refer to are defined, using the continous semantics of STL.

Mix-TL. The third type are mix-TL properties: properties that combine MTL formulas (over events) and STL formulas (over signals). In this case, the signals are discretized by sampling the signal at the timestamps of the relevant events, plus at a fixed number of points between the events, and the result is checked using semantics similar to that of MTL.

In this thesis, we focus on MTL properties and consider properties using continuous signals out of scope.

The properties written in ETL using the syntax of Definition 3.2 are translated to MTL. The data structure in TRACE4CPS for storing and checking MTL properties follows the following grammar:

Definition 3.3 (MTL formulas in TRACE4CPS)

$$\varphi \coloneqq \mathsf{true} \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \rightarrow \varphi_2 \mid \varphi_1 \, \mathsf{U}_I \, \varphi_2$$

The events are encoded as a sequence $\rho = (t_0, m_0), (t_1, m_1), \ldots$ where $t_i \in \mathbb{R}$ is the timestamp and $m_i \in \mathbf{M}$ is the attribute mapping of event i. ρ is sorted according to ascending timestamps. The ordering of two events with the same timestamp is arbitrary.

The weak and strong semantics for MTL as defined in Definition 2.11 are used to obtain a verdict for a property φ on a finite trace ρ as follows:

Definition 3.4 (Informative prefix verdicts for MTL) [Hendriks et al. 2023]

If $\rho \models^+ \varphi$, then ρ is an *informative good prefix* for φ, so the verdict is GOOD. If $\rho \not\models^- \varphi$, then ρ is an *informative bad prefix* for φ, so the verdict is BAD. Otherwise, ρ is *non-informative*, so the verdict is NON-INFORMATIVE.

⋆ ⋆ ⋆

An example of a property specified in ETL is the following property, which concerns the image processing pipeline trace of Figure 3.1. It specifies that a single job may not take longer than 40 milliseconds.

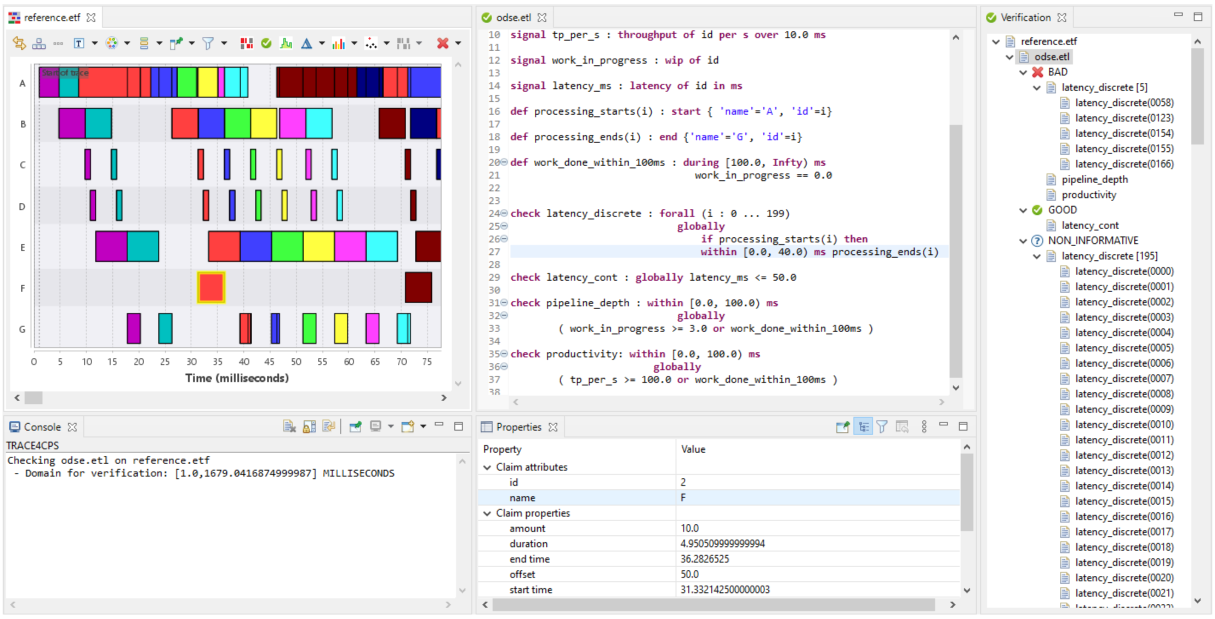
Example 3.5 (Image processing latency)

```
def processing_starts(i): start {'name'='A', 'id'=i}

def processing_ends(i): end {'name'='G', 'id'=i}

check latency_discrete: forall (i: 0 ... 199)
                            globally
                                if processing_starts(i) then
                                within [0.0, 40.0) ms processing_ends(i)
```

It is translated internally by TRACE4CPS into a set of MTL properties, with i from 0 to 199:

$$\mathsf{G}\left((\mathit{start}\ \{\mathit{name} \mapsto A, \mathit{id} \mapsto i\}) \rightarrow \mathsf{F}_{[0,40)}(\mathit{end}\ \{\mathit{name} \mapsto G, \mathit{id} \mapsto i\})\right)$$

Figure 3.2 shows the interface of TRACE4CPS after an ETL file has been verified on a trace. The Verification panel (on the right in Figure 3.2) shows the verdicts of the properties, and the quantified property `latency_discrete` is expanded to show the verdict for each value of i.



```
signal tp_per_s : throughput of id per s over 10.0 ms

signal work_in_progress : wip of id

signal latency_ms : latency of id in ms

def processing_starts(i) : start { 'name'='A', 'id'=i}

def processing_ends(i) : end {'name'='G', 'id'=i}

def work_done_within_100ms : during [100.0, Infty) ms
                              work_in_progress == 0.0


check latency_discrete : forall (i : 0 ... 199)
                        globally
                           if processing_starts(i) then
                           within [0.0, 40.0) ms processing_ends(i)

check latency_cont : globally latency_ms <= 50.0

check pipeline_depth : within [0.0, 100.0) ms
                        globally
      ( work_in_progress >= 3.0 or work_done_within_100ms )

check productivity: within [0.0, 100.0) ms
                     globally
      ( tp_per_s >= 100.0 or work_done_within_100ms )
```

```
TRACE4CPS
Checking odse.etl on reference.etf
 - Domain for verification: [1.0,1679.0416874999987] MILLISECONDS
```

Property	Value
Claim attributes	
id	2
name	F
Claim properties	
amount	10.0
duration	4.950509999999994
end time	36.2826525
offset	50.0
start time	31.332142500000003

```
reference.etf
  odse.etl
    BAD
      latency_discrete [5]
        latency_discrete(0058)
        latency_discrete(0123)
        latency_discrete(0154)
        latency_discrete(0155)
        latency_discrete(0166)
      pipeline_depth
      productivity
    GOOD
      latency_cont
    NON_INFORMATIVE
      latency_discrete [195]
        latency_discrete(0000)
        latency_discrete(0001)
        latency_discrete(0002)
        latency_discrete(0003)
        latency_discrete(0004)
        latency_discrete(0005)
        latency_discrete(0006)
        latency_discrete(0007)
        latency_discrete(0008)
        latency_discrete(0009)
        latency_discrete(0010)
        latency_discrete(0011)
        latency_discrete(0012)
        latency_discrete(0013)
        latency_discrete(0014)
        latency_discrete(0015)
        latency_discrete(0016)
        latency_discrete(0017)
        latency_discrete(0018)
        latency_discrete(0019)
        latency_discrete(0020)
        latency_discrete(0021)
```

Figure 3.2: Interface of TRACE4CPS after an ETL file has been checked

The algorithm for checking an MTL formula to compute a verdict according to Definition 3.4 (developed by Hendriks et al. [2016a]) is explained in Section 4.2.

Visual explanations for the verdicts, based on the intermediate values calculated by the checking algorithm, are added to the trace visualization when the user double-clicks on a property in the Verification panel. These are presented in Section 5.1.

4 Four-valued verdicts

When a property is checked on a trace in TRACE4CPS, one of three possible verdicts are generated: TRUE, FALSE, or NON-INFORMATIVE, as described in Definition 3.4. In practice, because of how the concept of informativeness is defined (see Section 2.1.3), many properties and traces result in a NON-INFORMATIVE verdict. This verdict represents the fact that, based on the information contained in the trace, the property is not definitively satisfied or violated in all extension of the trace. However, the user is likely interested in the validity of the property in the trace *so far*, but this information can only be determined from the verdict indirectly, or not at all, depending on the property.

In Section 4.1, a four-valued semantics is formulated with verdicts that can take on of the values TRUE, STILL_TRUE, STILL_FALSE, and FALSE. Inspired by the four-valued logic of RV-LTL, it uses the finite semantics of MTL to make a previously NON-INFORMATIVE verdict more precise.

Section 4.2 summarizes work by Hendriks et al. [2016b] that forms the basis of the algorithm in TRACE4CPS that computes the three-valued verdicts. While this algorithm could be used to obtain a four-valued verdict by first checking the property in the three-valued semantics and subsequently checking it with an algorithm that uses the finite semantics, this would mean that the values of the subformulas on events in the trace would not match the four-valued semantics, which could lead to confusing results in the visual explanations. For this reason, and in the interest of efficiency, Section 4.3 presents an extended MTL checking algorithm that produces a verdict in the four-valued semantics directly.

In Section 4.4, insights from the creation of this algorithm are used to provide a more direct, inductive definition of the four-valued informative prefix semantics for both MTL and LTL.

4.1 Four-valued semantics

Definition 3.4, which defines the (three-valued) informative prefix semantics of MTL, can be reformulated as follows:

Definition 4.1 (Informative prefix verdicts) Let ρ be a finite timed sequence and φ an MTL formula.

$$[\rho \models_{\mathrm{IP3}} \varphi] = \begin{cases} \top & \text{if } \rho \models^{+} \varphi & (\textsc{true}) \\ \bot & \text{if } \rho \not\models^{-} \varphi & (\textsc{false}) \\ ? & \text{otherwise} & (\textsc{non-informative}) \end{cases}$$

To illustrate the limits of a three-valued temporal logic, consider the following two situations.

We have the property $\mathsf{G}(\neg \mathit{error})$, which says that the proposition *error* should never occur:

$$[\{\neg \mathit{error}\}, \{\neg \mathit{error}\}, \{\neg \mathit{error}\} \models_{\text{IP3}} \mathsf{G}(\neg \mathit{error})] = ?$$

and the property $\mathsf{F}\,\mathit{success}$, which says that eventually *success* should be reached:

$$[\{\neg \mathit{success}\}, \{\neg \mathit{success}\}, \{\neg \mathit{success}\} \models_{\text{IP3}} \mathsf{F}\,\mathit{success}] = ?$$

In the first situation, the verdict is NON-INFORMATIVE because no error has occurred but there could be a future state in which the error does occur. In the second situation, the verdict is NON-INFORMATIVE because succes has not been reached but there could be a future state in which success is reached.

Both situations have the same verdict, but to a user who wants to perform verification on these finite sequences, these situations are very different. In the first, nothing has "gone wrong" yet, otherwise the verdict would have been $\bot$, so an NON-INFORMATIVE verdict is good news. In the second, our desired situation has not been reached, otherwise the verdict would have been $\top$, so an NON-INFORMATIVE verdict is bad news.

Using the finite semantics (similar to Definition 2.3) here would result in a TRUE verdict for the first property and a FALSE verdict for the second. However, this could be seen as too presumptuous: a sequence that has not satisfied or violated a property *yet* is different from a sequence where the validity of a property has been conclusively proven.

The knowledge that the only two possible verdicts for the property $\mathsf{G}(\neg \mathit{error})$ are NON-INFORMATIVE and FALSE does give us more information. When we receive a NON-INFORMATIVE verdict, we can deduce that *error* has not happened yet. Similarly, because the only possible verdicts for the property $\mathsf{F}\,\mathit{success}$ are NON-INFORMATIVE and TRUE, if we receive a NON-INFORMATIVE verdict, we know that *success* has not happened yet. This level of insight in the semantics of the verification process, which may not be evident in more complex properties, is ideally not required of the user of a verification program.

There are also properties where the three-valued informative semantics does not give any useful information, in the case of *non-monitorable properties* (see Section 2.1.5). An example is the formula $\mathsf{G}(\mathit{req} \to \mathsf{F}\,\mathit{ack})$, which specifies that every request should eventually be acknowledged. This property evaluates to NON-INFORMATIVE for *every* possible finite sequence, and does not give us the ability to distinguish between a sequence where every request has been acknowledged and one that contains non-acknowledged requests [Bauer et al. 2008].

Because of the tendency of NON-INFORMATIVE verdicts to dominate in practice, and the lack of useful information they offer the user, an extension of the verification functionality of TRACE4-CPS to return verdicts from the set $\{\text{TRUE}, \text{STILL\_TRUE}, \text{STILL\_FALSE}, \text{FALSE}\}$ ($\{\top, \top^?, \bot^?, \bot\}$) is presented.

The verdicts TRUE and FALSE follow from the satisfaction in the strong semantics and the violation in the weak semantics respectively, as before. Inspired by RV-LTL [Bauer et al. 2008] (Section 2.1.5), NON-INFORMATIVE is split into STILL_TRUE and STILL_FALSE by using the validity according to the finite semantics of MTL (which is analogous to LTL_f).

Definition 4.2 (Four-valued informative prefix semantics using $\models^+$, $\models^-$, and $\models_f$) Let ρ be a finite timed sequence and φ an MTL formula.

$$[\rho \models_{\text{IP4}} \varphi] = \begin{cases} \top & \text{if } \rho \models^+ \varphi & (\textsc{true}) \\ \bot & \text{if } \rho \not\models^- \varphi & (\textsc{false}) \\ \top^? & \text{if } \rho \not\models^+ \varphi \text{ and } \rho \models_f \varphi & (\textsc{still\_true}) \\ \bot^? & \text{if } \rho \models^- \varphi \text{ and } \rho \not\models_f \varphi & (\textsc{still\_false}) \end{cases}$$

Note that $\rho \models_f \varphi$ implies $\rho \models^- \varphi$, and $\rho \not\models_f \varphi$ implies $\rho \not\models^+ \varphi$ (from Proposition 2.5), so Definition 4.2 covers all combinations of strong, weak and finite verdicts. This is made clear in the following overview:

		$\models_f$	$\not\models_f$
$\models^+$	$\models^-$	$\top$	—[1]
	$\not\models^-$	—[2]	—[1]
$\not\models^+$	$\models^-$	$\top^?$	$\bot^?$
	$\not\models^-$	—[2]	$\bot$

[1] impossible because $\rho^i \models^+ \varphi \Rightarrow \rho^i \models_f \varphi$.
[2] impossible because $\rho^i \models_f \varphi \Rightarrow \rho^i \models^- \varphi$

The intuition of the semantics of Definition 4.2 is as follows. ρ is assumed to be a truncated prefix of a longer sequence. The four-valued verdict is TRUE if φ was shown to be satisfied before the end of ρ, and it is FALSE if φ was violated before the end of ρ. If the end of ρ was reached before the truth value of φ could definitively be determined, the verdict is STILL_TRUE if φ is true *for now*, but it is not certain that it cannot be violated later, and STILL_FALSE if it is *not yet* true, but could possibly become true later.

The four-valued informative prefix semantics is similar, but not identical, to that of RV-LTL ($\models_{\text{RV}}$). Because the informative prefix semantics only evaluate to TRUE and FALSE for informative prefixes (see Section 2.1.3), rather than all good and bad prefixes (see Definition 2.6), there are cases where a STILL_TRUE verdict in $\models_{\text{IP4}}$ is too pessimistic or a STILL_FALSE verdict too optimistic. For example, the formula $\mathsf{G}(p \vee \neg p)$ evaluates to true on every finite sequence in RV-LTL, because there is no infinite sequence that violates it, and therefore it is satisfied in any extension of any finite sequence. In contrast, in the semantics given by Definition 4.2, $\mathsf{G}(p \vee \neg p)$ would be STILL_TRUE for any finite sequence, since the strong semantics evaluates to false if the truth value of a formula relies on any information beyond the end of the sequence (but it is true in the finite semantics for any finite sequence). Dually, the property $\mathsf{F}(p \wedge \neg p)$ is false for any finite sequence in RV-LTL, since there is no (finite or infinite) sequence that could satisfy it, but it is STILL_FALSE in $\models_{\text{IP4}}$.

This means that occasionally, a STILL_TRUE verdict is received even when there is no extension of the sequence that could violate it, or a STILL_FALSE verdict is received even though there is no extension that could satisfy it. In our experience, because the STILL_TRUE and STILL_FALSE are more precise than NON-INFORMATIVE, and because we expect formulas containing tautologies or unsatisfiable formulas to be quite rare in practice, this does not pose a problem.

4.2 TRACE4CPS checking algorithm

To see how the MTL checking algorithm in TRACE4CPS can be extended to four-valued verdicts, let us look at how the current algorithm works.

The definitions of the strong, weak and finite semantics used in the informative prefix algorithm in TRACE4CPS are as follows. (Colors are used to show the correpondences between conditions in Definition 4.3 and variables in Algorithm 4.4.)

Definition 4.3 (Strong, weak and finite semantics of MTL) [Hendriks et al. 2016b] Let $\rho = (s_0, t_0), \ldots, (s_n, t_n)$ be a finite timed sequence of length $n+1$, $p \in AP$ an atomic proposition, $I \subseteq [0, \infty)$ an interval of the form $[\inf(I), \sup(I))$[1], $\varphi, \varphi_1, \varphi_2$ MTL formulas, $0 \leq i \leq n$ a position in the sequence.

The strong semantics $\models^+$ is defined as

$$\begin{array}{lll}
\rho^i \models^+ \mathsf{true} & & \\
\rho^i \models^+ p & \text{iff} & p \in L(s_i) \\
\rho^i \models^+ \varphi_1 \wedge \varphi_2 & \text{iff} & \rho^i \models^+ \varphi_1 \text{ and } \rho^i \models^+ \varphi_2 \\
\rho^i \models^+ \neg\varphi & \text{iff} & \rho^i \not\models^- \varphi \\
\rho^i \models^+ \varphi_1 \,\mathsf{U}_I\, \varphi_2 & \text{iff} & \exists_{i \leq j \leq n} [\rho^i \models^+ \varphi_2 \text{ and } t_j - t_i \in I \text{ and } \forall_{i \leq k < j}\, \rho^k \models^+ \varphi_1]
\end{array}$$

The weak semantics $\models^-$ is defined as

$$\begin{array}{lll}
\rho^i \models^- \mathsf{true} & & \\
\rho^i \models^- p & \text{iff} & p \in L(s_i) \\
\rho^i \models^- \varphi_1 \wedge \varphi_2 & \text{iff} & \rho^i \models^- \varphi_1 \text{ and } \rho^i \models^- \varphi_2 \\
\rho^i \models^- \neg\varphi & \text{iff} & \rho^i \not\models^+ \varphi \\
\rho^i \models^- \varphi_1 \,\mathsf{U}_I\, \varphi_2 & \text{iff} & (\exists_{i \leq j \leq n}\ [\rho^j \models^- \varphi_2 \text{ and } t_j - t_i \in I \text{ and } \forall_{i \leq k < j}\, \rho^k \models^- \varphi_1]) \\
 & & \text{or } (t_n - t_i < \sup(I) \text{ and } \forall_{i \leq k \leq n}\, \rho^k \models^- \varphi_1)
\end{array}$$

The finite semantics $\models_f$ is defined as

$$\begin{array}{lll}
\rho^i \models_f \mathsf{true} & & \\
\rho^i \models_f p & \text{iff} & p \in L(s_i) \\
\rho^i \models_f \varphi_1 \wedge \varphi_2 & \text{iff} & \rho^i \models_f \varphi_1 \text{ and } \rho^i \models_f \varphi_2 \\
\rho^i \models_f \neg\varphi & \text{iff} & \rho^i \not\models_f \varphi \\
\rho^i \models_f \varphi_1 \,\mathsf{U}_I\, \varphi_2 & \text{iff} & \exists_{i \leq j \leq n}\ [\rho^i \models_f \varphi_2 \text{ and } t_j - t_i \in I \text{ and } \forall_{i \leq k < j}\, \rho^k \models \varphi_1]
\end{array}$$

Based on these definitions, Hendriks et al. [2016b] present an algorithm that computes a verdict according to the informative prefix semantics. It is repeated here and explained briefly. For a full justification and a proof of correctness of this algorithm, see [Hendriks et al. 2016b].

[1]Here, and in the rest of this chapter, the right endpoint of I is assumed to be open and finite. The semantics and the algorithms in this chapter can easily be extended to any open, closed or half-open intervals: if the right endpoint of I is closed, the expression $t_n - t_i < \sup(I)$ is replaced by $t_n - t_i \leq \sup(I)$, and if the upper bound of I is ∞, it is always true.

Algorithm 4.4 shows the three-valued informative prefix algorithm in pseudocode. To compute $[\rho \models_{\text{IP3}} \varphi]$, it is invoked as $\textsc{getOrCompute}(\rho, \varphi, 0)$.

It operates on verdicts from a lattice $\mathbb{B}_3 = \{\top, \bot, ?\}$, where $\bot \sqsubset ? \sqsubset \top$. $\top$ and $\bot$ are complementary to each other and $?$ is complementary to itself. The resulting three-valued logic behaves as expected: $\neg_3$ generalizes $\neg$:

$$\neg_3\top = \bot \qquad \neg_3\bot = \top \qquad \neg_3? = ?$$

and $\sqcap$ and $\sqcup$ generalize $\wedge$ and $\vee$:

$$\bot \sqcap ? = \bot \qquad \top \sqcap ? = ? \qquad \bot \sqcup ? = ? \qquad \top \sqcup ? = \top \qquad \text{etc.}$$

The values for each subformula and position in the sequence are memoized in a memoization table $v : \Phi \times \mathbb{N} \rightharpoonup \{\top, \bot, ?, \text{undefined}\}$ which is initialized to be undefined for every subformula and position, and which serves as a basis for the property explanation features described in Chapter 5.

The boolean variables $c_1, c_{11}, c_2, c_{22}, c_3$ correspond to the conditions for the *until* operator in the strong and weak semantics as described in Definition 4.3.

The condition for the strong semantics is computed as follows: $\rho^i \models^+ \varphi_1 \mathsf{U}_I \varphi_2$ holds if and only if $C_1(i, n)$ holds, where, given a trace $\rho = (s_0, t_0), \ldots, (s_n, t_n)$, C_1 is defined as follows:

$$C_1(i, m) = \exists_{i \leq j \leq m}[\rho^j \models^+ \varphi_2 \wedge t_j - t_i \in I \wedge C_{11}(i, j)]$$
$$C_{11}(i, j) = \forall_{i \leq k < j}\ [\rho^k \models^+ \varphi_1]$$

The conditions for the weak semantics are similar, except that their negation is calculated, since we are interested in the case where the formula is false in the weak semantics. The validity of $\varphi_1 \mathsf{U}_I \varphi_2$ in the weak semantics is a disjunction of two conditions. Consequently, $\rho^i \not\models^- \varphi_1 \mathsf{U}_I \varphi_2$ holds if and only if both $C_2(i, n)$ and $C_3(i, n)$ hold, which are defined as follows:

$$C_2(i, m) = \forall_{i \leq j \leq m}[\rho^j \not\models^- \varphi_2 \vee t_j - t_i \notin I \vee C_{22}(i, j)]$$
$$C_{22}(i, j) = \exists_{i \leq k < j}\ [\rho^k \not\models^- \varphi_1]$$
$$C_3(i, m) = t_n - t_i \geq \sup(I) \vee \exists_{i \leq j \leq m}[\rho^j \not\models^- \varphi_1]$$

The variables c_{11} and c_{22} are used in the calculation of the values of c_1 and c_2, but are updated *after* c_1 and c_2, so that the value of c_{11} used in c_1 only represents the states up to but not including the current position (just as the domain of the universal quantifier in C_{11} is not inclusive of j), and the same applies to c_{22} and c_2.

At each iteration of the algorithm, after line 29, $c_1 \Leftrightarrow C_1(i, j)$, $c_2 \Leftrightarrow C_2(i, j)$, and $c_3 \Leftrightarrow C_3(i, j)$, and after line 31, $c_{11} \Leftrightarrow C_{11}(i, j)$ and $c_{22} \Leftrightarrow C_{22}(i, j)$.

An important difference between the variables for the strong semantics and those for the weak semantics relies on the fact that the algorithm operates in the three-valued logic of $\mathbb{B}_3$. The variables c_1 and c_{11} compute the strong semantics because in lines 24 and 30, the values of $[\rho^j \models_{\text{IP3}} \varphi_1]$ and $[\rho^j \models_{\text{IP3}} \varphi_2]$ are compared only against the value $\top$ (of the three possible values of $\top$, $\bot$ and $?$). Likewise, c_2 and c_{22} compute (part of) the weak semantics because in lines 28 and 31, the values of $[\rho^j \models_{\text{IP3}} \varphi_1]$ and $[\rho^j \models_{\text{IP3}} \varphi_2]$ are compared against $\bot$. If either φ_1 or φ_2 contains temporal operators, $[\rho^j \models_{\text{IP3}} \varphi_1]$ or $[\rho^j \models_{\text{IP3}} \varphi_2]$ can evaluate evaluates to $?$. In this case, it is possible that neither c_1 nor $c_2 \wedge c_3$ holds, signifying that the formula $\varphi_1 \mathsf{U}_I \varphi_2$

is neither satisfied in the strong semantics nor violated in the weak semantics, resulting in the value ?.

As an optimization in COMPUTE$(\rho, \varphi_1 \mathsf{U}_I \varphi_2, i)$, if before the end of the loop the value of $[\rho^j \models_{\text{IP3}} \varphi_1 \mathsf{U}_I \varphi_2]$ can already be determined, it is returned early:

- At line 25, when c_1 becomes *true* at some position j, it will remain *true*. So, $\varphi_1 \mathsf{U}_I \varphi_2$ is shown to hold in the strong semantics, and $\top$ can be returned early.
- At line 32, when at the last iteration $(j = n)$, or φ_1 is $\bot$ at some state ($c_{22} = \mathit{true}$), or we have reached a state past the interval I $(t_j - t_i \geq \sup(I))$, if c_2 and c_3 are *true*, they will remain true. So, $\varphi_1 \mathsf{U}_I \varphi_2$ is shown not to hold in the weak semantics, and $\bot$ is returned.
- At line 34, if c_1 is false and either c_{11} is false or we have reached a state past the interval I, then c_1 will remain false. Furthermore, if c_2 is false then it will remain false. This implies that neither $\rho^i \models^+ \varphi_1 \mathsf{U}_I \varphi_2$ nor $\rho^i \not\models^- \varphi_1 \mathsf{U}_I \varphi_2$ will be satisfied, so ? is returned.
- As a consequence of the first two points, at line 36 after the last iteration, we know that neither $\rho^i \models^+ \varphi_1 \mathsf{U}_I \varphi_2$ nor $\rho^i \not\models^- \varphi_1 \mathsf{U}_I \varphi_2$ have been satisfied, so ? is returned.

Algorithm 4.4 (Three-valued informative prefix for MTL) [Hendriks et al. 2016b]
Let $\rho = (s_0, t_0), \ldots, (s_n, t_n)$ be a finite timed sequence of length $n+1$, $p \in AP$ an atomic proposition, $I \subseteq [0, \infty)$ an interval of the form $[\inf(I), \sup(I))$, φ an MTL formula, and i a position in the sequence such that $0 \le i \le n$.
COMPUTE(ρ, φ, i) computes $[\rho^i \models_{\mathrm{IP3}} \varphi]$. GETORCOMPUTE$(\rho, \varphi, i)$ is a wrapper around COMPUTE that performs memoization of the computed values using the memoization table $v : \Phi \times \mathbb{N} \rightharpoonup \{\top, \bot, ?, \text{undefined}\}$.

```
1:  $v \leftarrow$ undefined for all $\varphi, i$
2:  procedure GETORCOMPUTE$(\rho, \varphi, i)$
3:      $result \leftarrow v(\varphi, i)$
4:      if $result =$ undefined then
5:          $result \leftarrow$ COMPUTE$(\rho, \varphi, i)$
6:          $v \leftarrow result$
7:      return $result$
8:  procedure COMPUTE$(\rho, \mathsf{true}, i)$
9:      return $\top$
10: procedure COMPUTE$(\rho, p, i)$
11:     if $p \in L(s_i)$ then
12:         return $\top$
13:     else
14:         return $\bot$
15: procedure COMPUTE$(\rho, \neg\varphi, i)$
16:     return $\neg_3$GETORCOMPUTE$(\rho, \varphi, i)$
17: procedure COMPUTE$(\rho, \varphi_1 \wedge \varphi_2, i)$
18:     return GETORCOMPUTE$(\rho, \varphi_1, i) \sqcap$ GETORCOMPUTE$(\rho, \varphi_2, i)$
19: procedure COMPUTE$(\rho, \varphi_1 \mathsf{U}_I \varphi_2, i)$
20:     $c_1 \leftarrow false, c_{11} \leftarrow true$
21:     $c_2 \leftarrow true, c_{22} \leftarrow false, c_3 \leftarrow t_n - t_i \ge \sup(I)$
22:     for $j = i$ to $n$ do
23:         $r_2 \leftarrow$ GETORCOMPUTE$(\rho, \varphi_2, j)$
24:         $c_1 \leftarrow c_1 \vee (r_2 = \top \wedge t_j - t_i \in I \wedge c_{11})$
25:         if $c_1$ then
26:             return $\top$
27:         $r_1 \leftarrow$ GETORCOMPUTE$(\rho, \varphi_1, j)$
28:         $c_2 \leftarrow c_2 \vee (r_2 = \bot \vee t_j - t_i \notin I \vee c_{22})$
29:         $c_3 \leftarrow c_3 \vee (r_1 = \bot)$
30:         $c_{11} \leftarrow c_{11} \wedge (r_1 = \top)$
31:         $c_{22} \leftarrow c_{22} \vee (r_1 = \bot)$
32:         if $(j = n \vee c_{22} \vee t_j - t_i \ge \sup(I)) \wedge c_2 \wedge c_3$ then
33:             return $\bot$
34:         if $(\neg c_{11} \vee t_j - t_i \ge \sup(I)) \wedge \neg c_1 \wedge \neg c_2$ then
35:             return ?
36:     return ?
```

4.3 Algorithm for four-valued semantics

4.3.1 Extending the algorithm

In order to compute $[\rho \models_{\mathrm{IP4}} \varphi]$ so that it complies with Definition 4.2, Algorithm 4.4 is modified so that it operates on truth values from the lattice $\mathbb{B}_4 = \{\top, \top^?, \bot^?, \bot\}$, where $\bot \sqsubset \bot^? \sqsubset \top^? \sqsubset \top$. $\top$ and $\bot$ are complementary to each other and $\top^?$ and $\bot^?$ are complementary (so that $\neg_4\top = \bot$, $\neg_4\bot = \top$, $\neg_4\top^? = \bot^?$, and $\neg_4\bot^? = \top^?$). In the resulting four-valued logic, $\neg_4$ generalizes $\neg$ and $\sqcap$ and $\sqcup$ again generalize $\wedge$ and $\vee$ as expected.

Now, instead of returning ?, the algorithm should return either $\top^?$ or $\bot^?$, using the finite semantics to decide between the two. Apart from updating the code for negation and conjunction to operate on $\mathbb{B}_4$ instead of $\mathbb{B}_3$, only the case for the *until* operator needs to be changed, since that is the only place where a ? can originate. This is not surprising, because a formula consisting only of non-temporal operators needs to evaluate only a single state. Thus, it cannot depend on information beyond the end of the sequence, and will therefore always evaluate to either $\top$ or $\bot$.

MTL formulas with non-nested *until* It is instructive to first look at the subset of MTL formulas where temporal operators are not nested. That is, formulas that can contain any of the constructs in the syntax of MTL, but in which subformulas of an *until* operator consist only of the propositional connectives ($\wedge$, $\neg$, and true) and atomic propositions. In this subset, if we encounter a subformula of the form $\varphi_1 \,\mathsf{U}_I\, \varphi_2$, we know that for any index j, both $[\rho^j \models_{\mathrm{IP4}} \varphi_1]$ and $[\rho^j \models_{\mathrm{IP4}} \varphi_2]$ will be either $\top$ or $\bot$ for the reason given above.

Under this assumption, four possible cases in the evaluation of $\textsc{compute}(\rho, \varphi_1 \,\mathsf{U}_I\, \varphi_2, i)$ can be distinguished:

- There is some state at position j which lies in the interval I where φ_2 holds, and for all states from i until j (exclusive), φ_1 holds. The formula is true in the strong semantics, so $\top$ is returned.
- φ_1 is found not to hold at some state before a state where φ_2 holds has been found in the interval I. The formula is false in the weak semantics so $\bot$ is returned.
- φ_2 holds at no state in the interval I and the sequence is not truncated before I ends. The formula is false in the weak semantics, so $\bot$ is returned.
- I extends past the end of the sequence, no state where φ_2 holds has been found, but φ_1 holds at every state starting at i. In this case, the formula is false in the strong semantics and is true in the weak semantics. The formula is false in the finite semantics, since the finite semantics requires that φ_2 holds at some state in the prefix, so we return $\bot^?$.

Thus, a formula of the form $\varphi_1 \,\mathsf{U}_I\, \varphi_2$ without nested temporal operators that holds in the finite semantics will always hold in the strong semantics too. Because $\mathbb{B}_4$ is totally ordered, the only way to obtain a $\top^?$ from a formula without nested temporal operators is by negating a $\bot^?$. It follows that for a formulas without nested temporal operators to evaluate to $\top^?$, an *until* must occur inside a negation. An example of such a formula is $\mathsf{G}\,\mathsf{true} = \neg(\mathsf{true}\,\mathsf{U}\,\neg\mathsf{true})$, which evaluates to $\top^?$ on every finite sequence.

The algorithm is easily modified so that it computes $[\rho \models_{\mathrm{IP4}} \varphi]$ for an MTL formula φ without nesting of temporal operators by updating the operators $\neg_3$, $\sqcap$, and $\sqcup$ to use $\mathbb{B}_4$ ($\neg_4$, $\sqcap$, and $\sqcup$) and replacing every statement where ? is returned in the procedure $\textsc{compute}(\rho, \varphi_1 \,\mathsf{U}_I\, \varphi_2, i)$ with one that returns $\bot^?$ instead (at lines 34 and 36 of Algorithm 4.4).

All MTL formulas To compute $[\rho \models_{\text{IP4}} \varphi]$ for all MTL formulas, including those with arbitrary nesting of temporal operators, we can no longer assume that subformulas of an until operator evaluate to $\top$ or $\bot$: $[\rho \models_{\text{IP4}} \varphi_1 \mathsf{U}_I \varphi_2]$ can evaluate to any of $\top$, $\top^?$, $\bot^?$, or $\bot$.

The cases where $\top$ and $\bot$ are returned again stay the same. The cases in Algorithm 4.4 where ? is returned are updated so that either $\bot^?$ or $\top^?$ are returned, and to decide between $\top^?$ and $\bot^?$ we use the finite semantics. As with the other variables, we define two conditions C_4 and C_{44} that correspond to the conditions of the until operator in the finite semantics: $C_4(i,n) \iff \rho^i \models_f \varphi_1 \mathsf{U}_I \varphi_2$:

$$C_4(i,m) = \exists_{i \le j \le m} [\rho^j \models_f \varphi_2 \land t_j - t_i \in I \land C_{44}(i,j)]$$
$$C_{44}(i,j) = \forall_{i \le k < j} \; [\rho^k \models_f \varphi_1]$$

In the algorithm we introduce variables c_4 and c_{44} such that after every iteration, $c_4 = C_4(i,j)$ and $c_{44} = C_{44}(i,j)$. This results in the following algorithm (where the parts that have changed since Algorithm 4.4 are highlighted):

Algorithm 4.5 (Four-valued informative prefix for MTL, version 1)
Let $\rho = (s_0,t_0),\ldots,(s_n,t_n)$ be a finite timed sequence of length $n+1$, $I \subseteq [0,\infty)$ an interval of the form $[\inf(I), \sup(I))$, φ_1, φ_2 MTL formulas, and i a position in the sequence such that $0 \le i \le n$.

```
1:  procedure COMPUTE($\rho, \varphi_1 \mathsf{U}_I \varphi_2, i$)
2:      $c_1 \leftarrow false, c_{11} \leftarrow true$
3:      $c_2 \leftarrow true, c_{22} \leftarrow false, c_3 \leftarrow t_n - t_i \ge \sup(I)$
4:      $\boxed{c_4 \leftarrow false, c_{44} \leftarrow true}$
5:      for $j = i$ to $n$ do
6:          $r_2 \leftarrow$ GETORCOMPUTE($\rho, \varphi_2, j$)
7:          $c_1 \leftarrow c_1 \lor (r_2 = \top \land t_j - t_i \in I \land c_{11})$
8:          $\boxed{c_4 \leftarrow c_4 \lor (r_2 \in \{\top, \top^?\} \land t_j - t_i \in I \land c_{44})}$
9:          if $c_1$ then
10:             return $\top$
11:         $r_1 \leftarrow$ GETORCOMPUTE($\rho, \varphi_1, j$)
12:         $c_2 \leftarrow c_2 \lor (r_2 = \bot \lor t_j - t_i \notin I \lor c_{22})$
13:         $c_3 \leftarrow c_3 \lor (r_1 = \bot)$
14:         $c_{11} \leftarrow c_{11} \land (r_1 = \top)$
15:         $c_{22} \leftarrow c_{22} \lor (r_1 = \bot)$
16:         $\boxed{c_{44} \leftarrow c_{44} \land (r_1 \in \{\top, \top^?\})}$
17:         if $(j = n \lor c_{22} \lor t_j - t_i \ge \sup(I)) \land c_2 \land c_3$ then
18:             return $\bot$
19:         if $(\neg c_{11} \lor t_j - t_i \ge \sup(I)) \land \neg c_1 \land \neg c_2$ $\boxed{\land\, c_4}$ then
20:             $\boxed{\text{return } \top^?}$
21:     if $c_4$ then
22:         return $\top^?$
23:     else
24:         return $\bot^?$
```

At lines 8 and 16, if the value of r_2 or r_1 is $\top$ or $\top^?$, we know that in the finite semantics it would have been $\top$.

At line 19, if previously ? would have been returned, we additionally check whether c_4 is true.

If c_4 is true, we know that c_4 will not become false later. $\varphi_1 \mathsf{U}_I \varphi_2$ is satisfied in the finite semantics, so we can return $\top^?$. If c_4 is not true, we do not yet know if it will become true in a later iteration, so we simply continue.

At line 21, after the last iteration, we again know that $\varphi_1 \mathsf{U}_I \varphi_2$ is neither satisfied in the strong semantics nor violated in the weak semantics. We return $\top^?$ or $\bot^?$ depending on whether it is satisfied in the finite semantics (c_4 is true or false, respectively).

As we have shown, the informative prefix checking algorithm can be extended to the four-valued semantics in such a way that the structure of the code remains mostly the same. The largest change to the control flow of the algorithm is in the condition to return $\top^?$ early, which was made stricter, resulting in more iterations being made in cases where previously ? would be returned but where the formula is not (yet) satisfied in the finite semantics. However, this does not impact the computational complexity of the algorithm in the worst case.

4.3.2 Reformulating the algorithm

In Algorithm 4.5 COMPUTE$(\rho, \varphi_1 \mathsf{U}_I \varphi_2, i)$ keeps track of 7 variables: two for both the strong and the finite semantics, and three for the weak semantics. However, there is some redundancy between these variables. For example, in the cases where c_1 is true, c_4 is always true, because $\rho^i \models^+ \varphi_1 \mathsf{U}_I \varphi_2$ implies $\rho^i \models_f \varphi_1 \mathsf{U}_I \varphi_2$. The variables c_1 and c_4 are initialized identically, and are updated almost identically. The only difference is that c_1 becomes true if r_2 is $\top$, and c_4 when r_2 is either $\top$ or $\top^?$. The same applies to c_{11} and c_{44}.

We can replace the boolean variables c_1, c_{11}, c_2, c_{22}, c_3, c_4 and c_{44} by two variables that take values from $\mathbb{B}_4$, and update them by replacing the boolean operations $\wedge$ and $\vee$ by the partial order operations that generalize them, $\sqcap$ and $\sqcup$, without losing any information.

We define a variable c_{lr} that generalizes c_1, c_2 and c_4 and a variable c_l that generalizes c_{11}, c_{22} and c_{44}, and can also be used to replace c_3 (lr and l signifying that they keep track of the results of both the left and right subformulas and only the left subformula of U, respectively). Let us again define two conditions that correspond to these variables:

$$\begin{aligned} C_{lr}(i,m) &= \textstyle\bigsqcup_{i \le j \le m} \left([\rho^j \models_{\mathrm{IP4}} \varphi_2] \sqcap t_j - t_i \in I \sqcap C_l(i, j-1)\right)^{2} \\ C_l(i,m) &= \textstyle\bigsqcap_{i \le j \le m} [\rho^j \models_{\mathrm{IP4}} \varphi_1] \end{aligned}$$

Note that C_{lr} and C_l closely mirror the conditions C_1 and C_{11}, etc., but that C_l is phrased slightly differently by being inclusive of the position m, because it is also used to cover C_3.

Proposition 4.6 (Connection between C_{lr} and strong, finite, and weak semantics) Let $\rho = (s_0, t_0), \ldots, (s_n, t_n)$ be a finite timed sequence of length $n+1$, φ_1 and φ_2 MTL formulas, $I \subseteq [0, \infty)$ an interval of the form $[\inf(I), \sup(I))$, and i a position in the sequence such that $0 \le i \le n$.

$$\begin{aligned} \rho^i \models^+ \varphi_1 \mathsf{U}_I \varphi_2 &\iff C_{lr}(i,n) = \top \\ \rho^i \models_f \varphi_1 \mathsf{U}_I \varphi_2 &\iff C_{lr}(i,n) \sqsupseteq \top^? \\ \rho^i \models^- \varphi_1 \mathsf{U}_I \varphi_2 &\iff C_{lr}(i,n) \neq \bot \text{ or } (t_n - t_i < \sup(I) \text{ and } C_l(i,n) \neq \bot) \end{aligned}$$

[2] In this expression and those that follow, the result of the expression $t_j - t_i \in I$ is assumed to be in $\mathbb{B}_4$ by mapping *true* to $\top$ and *false* to $\bot$.

We initialize the variables with $c_{lr} \leftarrow \bot$ and $c_l \leftarrow \top$ (the identity elements of $\sqcup$ and $\sqcap$, respectively), and again update c_l after c_{lr} so that the value of c_l used in c_{lr} only represents the states up to but not including the current position. The value of c_{lr} is updated in every iteration by a join operation of c_{lr} with the information of the next state, so it is monotonically increasing from $\bot$ in the direction of $\top$. The value of c_l is updated by a meet operation, so it is monotonically decreasing from $\top$ in the direction of $\bot$.

We can recover the strong and finite conditions c_1 and c_4 by taking $c_1 \Leftrightarrow [c_{lr} = \top]$ and $c_4 \Leftrightarrow [c_{lr} \sqsupseteq \top^?]$ (c_{lr} is at least $\top^?$, equivalent to $c_{lr} \in \{\top, \top^?\}$). Similarly, we have $c_{11} \Leftrightarrow [c_l = \top]$ and $c_{44} \Leftrightarrow [c_l \sqsupseteq \top^?]$.

The weak conditions c_2, c_{22} and c_3 (which are the negations of the conditions of the weak semantics), can also be recovered: $c_2 \Leftrightarrow [c_{lr} = \bot]$ and $c_{22} \Leftrightarrow [c_l = \bot]$. In Algorithm 4.4, c_3 is initialized as $t_n - t_i \geq \sup(I)$ and updated every iteration using $c_3 \leftarrow c_3 \vee (r_1 = \bot)$. It can be equivalently written in terms of our new variables as $c_3 \Leftrightarrow (t_n - t_i \geq \sup(I) \vee c_l = \bot)$.

Rewriting Algorithm 4.5 in terms of the variables c_{lr} and c_l results in the following equivalent algorithm:

Algorithm 4.7 (Four-valued informative prefix for MTL, version 2)
Let $\rho = (s_0, t_0), \ldots, (s_n, t_n)$ be a finite timed sequence of length $n+1$, $I \subseteq [0, \infty)$ an interval of the form $[\inf(I), \sup(I))$, $\varphi, \varphi_1, \varphi_2$ MTL formulas, and i a position in the sequence such that $0 \leq i \leq n$.

```
1:  procedure COMPUTE($\rho, \varphi_1 \mathsf{U}_I \varphi_2, i$)
2:      $c_{lr} \leftarrow \bot, c_l \leftarrow \top$
3:      for $j = i$ to $n$ do
4:          $r_2 \leftarrow$ GETORCOMPUTE($\rho, \varphi_2, j$)
5:          $c_{lr} \leftarrow c_{lr} \sqcup (r_2 \sqcap t_j - t_i \in I \sqcap c_l)$
6:          if $c_{lr} = \top$ then
7:              return $\top$
8:          $r_1 \leftarrow$ GETORCOMPUTE($\rho, \varphi_1, j$)
9:          $c_l \leftarrow c_l \sqcap r_1$
10:         if $(j = n \vee c_l = \bot \vee t_j - t_i \geq \sup(I)) \wedge c_{lr} = \bot \wedge (t_n - t_i \geq \sup(I) \vee c_l = \bot)$ then
11:             return $\bot$
12:         if $(c_l \sqsubset \top \vee t_j - t_i \geq \sup(I)) \wedge c_{lr} = \top^?$ then
13:             return $\top^?$
14:     return $c_{lr} \sqcup \bot^?$
```

Lines 6 and 10 were obtained by straightforward replacements of the previous conditions with their equivalents. At line 12, the same applies, but the expression $c_{lr} \sqsubset \top \wedge c_{lr} \sqsupset \bot \wedge c_{lr} \sqsupseteq \top^?$ was simplified to $c_{lr} = \top^?$.

At line 6, if the value of c_{lr} is $\top$, we know that it will remain so, because c_{lr} is monotonically increasing. In this case, $\varphi_1 \mathsf{U}_I \varphi_2$ is satisfied in the strong semantics, so $\top$ is returned.

At line 10, if we are at the last iteration, $c_l = \bot$ or the timestamp of the current state is past the interval I, we know that the value of c_{lr} will not increase anymore, and because c_l is monotonically decreasing, it will never increase. In this case, if the negated condition of the weak semantics ($c_{lr} = \bot \wedge (t_n - t_i \geq \sup(I) \vee c_l = \bot)$) holds, it will remain so. Thus, $\varphi_1 \mathsf{U}_I \varphi_2$ is violated in the weak semantics, so $\bot$ is returned.

At line 12, if the value of c_l is smaller than $\top$, it will remain so, because c_l is monotonically decreasing. If this is the case, or it is the case that the timestamp of the current event is past the interval I, we know (based on line 5) that c_{lr} will not become $\top$ if it is not yet $\top$. If the value of c_{lr} is not $\bot$ and we know that it will not become $\top$, we know that $\varphi_1 \mathsf{U}_I \varphi_2$ is neither satisfied in the strong semantics nor violated in the weak semantics. Then, if the value of c_{lr} is $\top^?$, it will remain so, which means that $\varphi_1 \mathsf{U}_I \varphi_2$ it is satisfied in the finite semantics, and $\top^?$ is returned. If the value of c_{lr} is smaller than $\top^?$, it may still become $\top^?$ in a later iteration, so we continue.

At line 14, we return the value of c_{lr} joined with $\bot^?$ because of the following:

- If $c_{lr} = \bot$, then the first condition for the weak semantics of *until* is false, but the result of COMPUTE should only be $\bot$ if the second condition for the weak semantics of *until* is also false, in which case COMPUTE would have returned at line 11. Thus, $\varphi_1 \mathsf{U}_I \varphi_2$ is not violated in the weak semantics, but it is violated in the finite semantics, so $\bot^?$ is returned.
- If $c_{lr} = \bot^?$, $\varphi_1 \mathsf{U}_I \varphi_2$ holds in the weak semantics, but it does not hold in the finite semantics, so $\bot^?$ is returned.
- If $c_{lr} = \top^?$, $\varphi_1 \mathsf{U}_I \varphi_2$ holds in the weak semantics, does not hold in the strong semantics, and holds in the finite semantics, so $\top^?$ is returned.
- If c_{lr} had become $\top$ at any point, COMPUTE would have returned at line 7, so line 14 would not be reached.

By replacing all occurrences of $\top^?$ and $\bot^?$ in Algorithm 4.7 with $?$, an algorithm equivalent to Algorithm 4.4 can be obtained.

4.4 Four-valued semantics, redux

From Proposition 4.6 and Algorithm 4.7, a concise semantics for $[\rho \models_{\text{IP4}} \varphi]$ that is inductive on the structure of the subformulas can be extracted:

Definition 4.8 (Four-valued informative prefix semantics, inductively)
Let $\rho = (s_0, t_0), \ldots, (s_n, t_n)$ be a finite timed sequence of length $n+1$, $p \in AP$ an atomic proposition, $I \subseteq [0, \infty)$ an interval of the form $[\inf(I), \sup(I))$, and i a position in the sequence such that $0 \leq i \leq n$.

The truth value of an MTL formula φ with respect to ρ is an element of a lattice $\mathbb{B}_4 = \{\top, \top^?, \bot^?, \bot\}$ such that $\bot \sqsubset \bot^? \sqsubset \top^? \sqsubset \top$ and $\neg_4 \top = \bot$ and $\neg_4 \top^? = \bot^?$, and is defined as

$$
\begin{aligned}
[\rho^i \models_{\text{IP4}} \mathsf{true}] &= \top \\
[\rho^i \models_{\text{IP4}} p] &= \begin{cases} \top & \text{if } p \in L(s_0) \\ \bot & \text{otherwise} \end{cases} \\
[\rho^i \models_{\text{IP4}} \varphi_1 \land \varphi_2] &= [\rho^i \models_{\text{IP4}} \varphi_1] \sqcup [\rho^i \models_{\text{IP4}} \varphi_2] \\
[\rho^i \models_{\text{IP4}} \neg\varphi] &= \neg_4 [\rho^i \models_{\text{IP4}} \varphi] \\
[\rho^i \models_{\text{IP4}} \varphi_1 \mathsf{U}_I \varphi_2] &= \begin{cases} \bot & \text{if } C(i,n) = \bot \text{ and} \\ & (t_n - t_i \geq \sup(I) \text{ or } (\textstyle\bigsqcap_{i \leq k \leq n} [\rho^k \models_{\text{IP4}} \varphi_1]) = \bot) \\ C(i,n) \sqcup \bot^? & \text{otherwise} \end{cases}
\end{aligned}
$$

$$\text{where } C(i,n) = \textstyle\bigsqcup_{i \leq j \leq n} ([\rho^j \models_{\text{IP4}} \varphi_2] \sqcap t_j - t_i \in I \sqcap \bigsqcap_{i \leq k < j} [\rho^k \models_{\text{IP4}} \varphi_1])$$

The case for the *until* operator can also be expressed more concisely as:

$$
\begin{aligned}
[\rho^i \models_{\text{IP4}} \varphi_1 \mathsf{U}_I \varphi_2] = &\textstyle\bigsqcup_{i \leq j \leq n} ([\rho^j \models_{\text{IP4}} \varphi_2] \sqcap t_j - t_i \in I \sqcap \bigsqcap_{i \leq k < j} [\rho^k \models_{\text{IP4}} \varphi_1]) \\
&\sqcup (t_n - t_i < \sup(I) \sqcap (\textstyle\bigsqcap_{i \leq j \leq n} [\rho^j \models_{\text{IP4}} \varphi_1]) \sqcap \bot^?)
\end{aligned}
$$

With Definition 4.8 we have a semantics that is equivalent to Definition 4.2, but is expressed as a single truth function instead of depending on the satisfaction relations of the strong, weak and finite semantics.

A three-valued informative prefix equivalent to Definition 4.1 can be obtained from Definition 4.8 by replacing $\mathbb{B}_4$ with $\mathbb{B}_3$ and $\bot^?$ with ?.

A four-valued informative prefix semantics for LTL can be obtained from Definition 4.8 by assuming every $I = [0, \infty)$ and defining the truth value for the *next* operator as follows:

Definition 4.9 (Four-valued informative prefix semantics for LTL)
Let $\sigma = s_0, \dots, s_n$ be a finite sequence of length $n+1$, and $0 \le i \le n$ a position in the sequence.
The truth value of an LTL formula φ with respect to σ is an element of a lattice $\mathbb{B}_4 = \{\top, \top^?, \bot^?, \bot\}$ such that $\bot \sqsubset \bot^? \sqsubset \top^? \sqsubset \top$ and $\neg_4 \top = \bot$ and $\neg_4 \top^? = \bot^?$, and is defined as

$$
\begin{aligned}
[\sigma^i \models_{\mathrm{IP4}} \textsf{true}] &= \top \\
[\sigma^i \models_{\mathrm{IP4}} p] &= \begin{cases} \top & \text{if } p \in L(s_0) \\ \bot & \text{otherwise} \end{cases} \\
[\sigma^i \models_{\mathrm{IP4}} \varphi_1 \wedge \varphi_2] &= [\sigma^i \models_{\mathrm{IP4}} \varphi_1] \sqcup [\sigma^i \models_{\mathrm{IP4}} \varphi_2] \\
[\sigma^i \models_{\mathrm{IP4}} \neg\varphi] &= \neg_4 [\sigma^i \models_{\mathrm{IP4}} \varphi] \\
[\sigma^i \models_{\mathrm{IP4}} \mathsf{X}\,\varphi] &= \begin{cases} \bot^? & \text{if } i = n \\ [\sigma^{i+1} \models_{\mathrm{IP4}} \varphi] & \text{otherwise} \end{cases} \\
[\sigma^i \models_{\mathrm{IP4}} \varphi_1 \,\mathsf{U}\, \varphi_2] &= \begin{cases} \bot & \text{if } C(i,n) = \bot \text{ and } (\bigsqcap_{i \le k \le n} [\sigma^k \models_{\mathrm{IP4}} \varphi_1]) = \bot \\ C(i,n) \sqcup \bot^? & \text{otherwise} \end{cases}
\end{aligned}
$$

where $C(i,n) = \bigsqcup_{i \le j \le n} \left([\sigma^j \models_{\mathrm{IP4}} \varphi_2] \sqcap \bigsqcap_{i \le k < j} [\sigma^k \models_{\mathrm{IP4}} \varphi_1] \right)$

5 | Interactive visualization of verdicts

In this chapter, several improvements to the visual property explanations of TRACE4CPS are developed.

Section 5.1 describes the presentation of verdicts and the property explanation features of TRACE4CPS before the start of this research project.

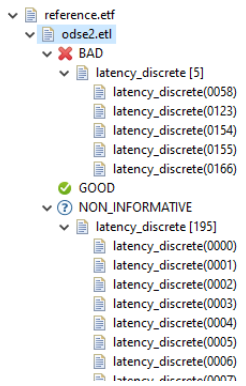
In Section 5.2, these are extended to allow the explanation of arbitrary subproperties.

In Section 5.3, a visualization of the time intervals of temporal operators that were visited during the property checking procedure is added.

5.1 TRACE4CPS property explanations

TRACE4CPS's Verification panel consists of a tree view with nodes that can be expanded and collapsed. After a set of properties is checked on a trace, the verdicts are shown in the Verification panel, grouped under three nodes for GOOD, BAD, and NON-INFORMATIVE.

Properties that consist of a quantification over the domain of a parameter (using `forall ...`) are grouped by the name of the top-level property. In the screenshot on the right, the property `latency_discrete` (from Section 3.3), which is quantified over the integers from 0 to 199, is FALSE for 5 values of the parameter, and NON-INFORMATIVE for the others.



```
reference.etf
  odse2.etl
    BAD
      latency_discrete [5]
        latency_discrete(0058)
        latency_discrete(0123)
        latency_discrete(0154)
        latency_discrete(0155)
        latency_discrete(0166)
    GOOD
    NON_INFORMATIVE
      latency_discrete [195]
        latency_discrete(0000)
        latency_discrete(0001)
        latency_discrete(0002)
        latency_discrete(0003)
        latency_discrete(0004)
        latency_discrete(0005)
        latency_discrete(0006)
        latency_discrete(0007)
```

To verify a property, the ETL formula is translated by TRACE4CPS into an MTL representation. As this MTL formula is checked using the algorithm described in Section 4.2, a memoization table is used to keep the computed values for each of the formula's subproperties on every event. At the end of the procedure, the table contains values for all subformula-event pairs that were visited in order to produce a verdict, and null values for those that were not visited.

When a property (or a property instantiated with a value in the case of a quantified property) is double-clicked, a property explanation is generated in the form of a set of events corresponding to the non-null values of the top-level formula and any named definitions in the formula in the memoization table. These events are added to the trace visualization in separate swimlanes, colored green, red and blue for TRUE, FALSE and NON-INFORMATIVE values, respectively. In Figure 5.1, a portion of the trace visualization is shown after the property `latency_discrete(0058)`, which has the verdict BAD, has been double-clicked.

Because the property explanation is only generated for named (sub)formulas (*checks* and *definitions*), it is up to the user to decide how the formula is factored out into definitions so that the property explanation is most useful.

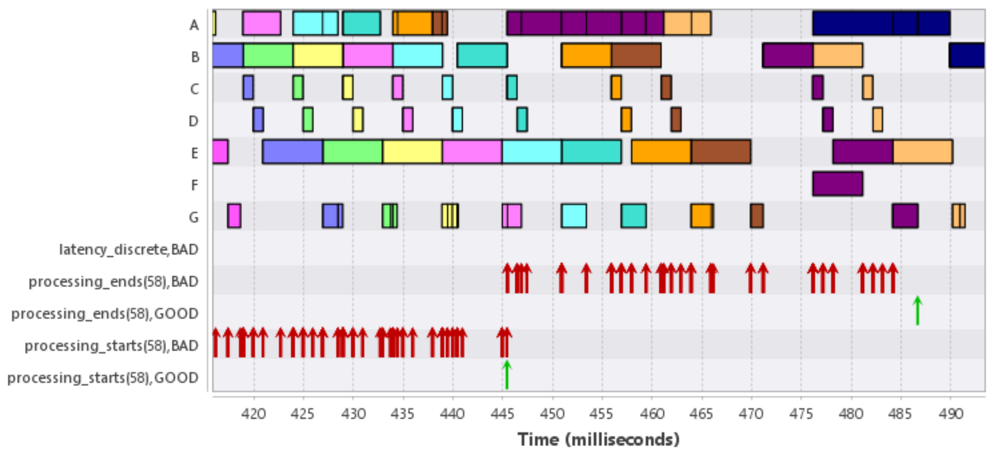


Figure 5.1: Property explanation for `latency_discrete(58)`

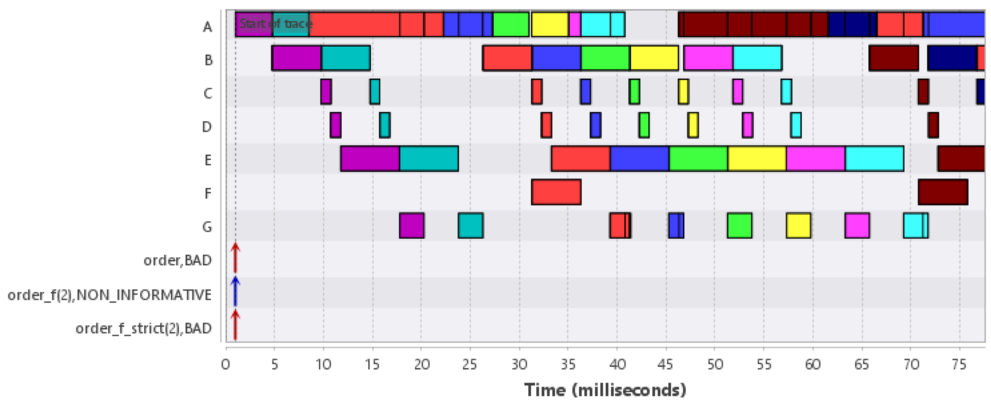


Figure 5.2: Property explanation for `order(2)`

Consider the property `order`, which specifies that in the image processing trace, task F may not start before task B has ended (`order_f`), and task F may not start before task C has ended (`order_f_strict`), and is defined as follows:

Example 5.1 (Ordering of image processing tasks)

```
def order_f(i): globally
  if start {'name'='F', 'id'=i} then
    until end {'name'='G', 'id'=i} we have that
      not end {'name'='B', 'id'=i}

def order_f_strict(i): globally
  if start {'name'='F', 'id'=i} then
    until end {'name'='G', 'id'=i} we have that
      not end {'name'='C', 'id'=i}

check order: forall (i: 0 ... 199) (order_f(i) and order_f_strict(i))
```

In Figure 5.2 the property explanation of the property `order(2)`, which has the verdict BAD, is shown. Because every named formula is a temporal formula that, when checked, is evaluated only on the first event of the trace, the explanation is not very helpful in this case.

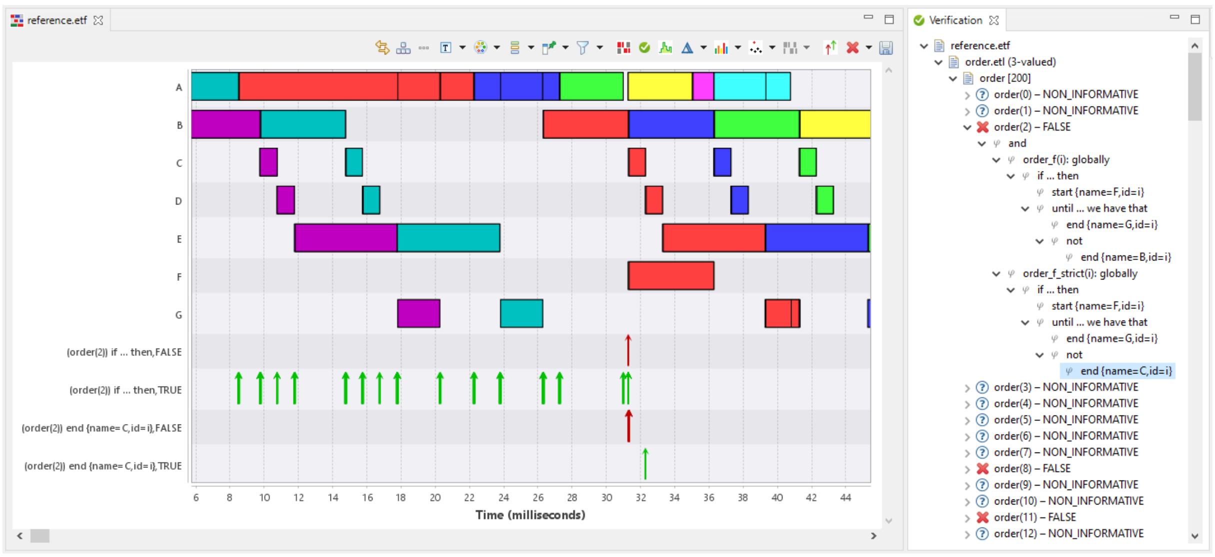
5.2 Explanations of subformulas

To increase the flexibility of property explanations, the ability to generate explanations for arbitrary subformulas of a property is added.

First, the Verification panel is reordered (shown on the right in Figure 5.3) so that, in the place of the GOOD, BAD, and NON-INFORMATIVE nodes, the properties in a file are shown in order, each with their own verdict. Additionally, the verdicts GOOD and BAD are renamed TRUE and FALSE to be in accordance with the verdicts of the four-valued semantics (TRUE, STILL_TRUE, STILL_FALSE, FALSE).

Double-clicking a property still generates an explanation of the top-level formula and the definitions it uses. However, the property can now be expanded to show the tree structure which corresponds to the abstract syntax tree of the formula,. Double-clicking a node in this formula tree generates an explanation of the sub-formula corresponding to that node. This is shown in Figure 5.3 for the node "`if ... then`" and the node "`end {name=C, id=i}`" of `order_f_strict`. The red arrow in the first of the four explanation swimlanes, together with the green arrow in the last swimlane, explain the reason that the property `order_f_strict` is violated: after the start of task F for job 2 (on which the `if ... then` subformula is evaluated), the end of task C occurs, indicating that task F has started before task C has finished executing.

The MTL representation of a formula (which is used to generate the memoization table on which the explanation is based) may be different from the ETL formula as specified by the user. For example, a formula of the form `globally ...` is represented in MTL as $\neg(\mathsf{true}\ \mathsf{U} \ldots)$. In order to generate the correct explanation for a node in the ETL representation of the formula, a mapping is introduced between ETL formulas and MTL formulas, which is generated during the translation process from ETL and MTL. The ETL subformulas are shown in the tree in the Verification panel, and the corresponding MTL subformula is looked up and used to generate the property explanation.


reference.etf
A
B
C
D
E
F
G
(order(2)) if ... then,FALSE
(order(2)) if ... then,TRUE
(order(2)) end {name=C,id=i},FALSE
(order(2)) end {name=C,id=i},TRUE
6
8
10
12
14
16
18
20
22
24
26
28
30
32
34
36
38
40
42
44
Time (milliseconds)
Verification

```
reference.etf
  order.etl (3-valued)
    order [200]
      order(0) – NON_INFORMATIVE
      order(1) – NON_INFORMATIVE
      order(2) – FALSE
        and
          order_f(i): globally
            if ... then
              start {name=F,id=i}
              until ... we have that
                end {name=G,id=i}
                not
                  end {name=B,id=i}
          order_f_strict(i): globally
            if ... then
              start {name=F,id=i}
              until ... we have that
                end {name=G,id=i}
                not
                  end {name=C,id=i}
      order(3) – NON_INFORMATIVE
      order(4) – NON_INFORMATIVE
      order(5) – NON_INFORMATIVE
      order(6) – NON_INFORMATIVE
      order(7) – NON_INFORMATIVE
      order(8) – FALSE
      order(9) – NON_INFORMATIVE
      order(10) – NON_INFORMATIVE
      order(11) – FALSE
      order(12) – NON_INFORMATIVE
```

Figure 5.3: New presentation of verification results of property `order`, with the trace visualization with explanations of two subformulas on the left and the Verification panel on the right

5.3 Explanations of real-time intervals

Besides the values of subformulas on events in the trace, the satisfaction of a property also depends on whether events occur in the intervals defined by the temporal operators with timing constraints. In ETL, these are represented by formulas of the form `during ...` (G_I), `within ...` (F_I), and `by ... and until then ...` (U_I).

An interval defined in a constrained temporal operator is evaluated relative to each event that a temporal operator is evaluated on. Thus, it can be difficult to see from the trace visualization which events occur within an interval.

To help the user understand the concrete timing constraints of properties, an additional explanation feature is added that visualizes the (relative) intervals that were encountered during the property checking procedure. This is done by adding virtual claims to the trace visualization, each corresponding to a concrete interval of the selected temporal operator. If a temporal operator is evaluated on multiple events, claims (colored gray) corresponding to the intervals relative to each of these events are shown in the same swimlane.

In Figure 5.4, a regular property explanation of the node `processing_ends(58)` and the new interval explanation of the node `within [0.0, 40.0) ms` are both shown in the trace visualization.

The combination of these two explanations clearly shows why `latency_discrete(58)` is FALSE. At time 486.67, task G ends, which is indicated by the green arrow in the first explanation swimlane. (Exact timestamps of events and claims are shown in a Properties panel when the user clicks on them.) However, the interval in the second explanation swimlane shows that the property requires that the end of task G occurs between time 445.43 and time 485.43. Thus, job 58 ends too late, and `latency_discrete(58)` is violated.

Finding the right combination of explanations (of subproperties and of temporal intervals) that illustrates the reason that a given verdict was reached is a matter of trying different explanations, and gradually adding more detail to the trace visualization as the user develops their understanding of the verdict. This process is supported by the ability to interactively add and remove explanations in the trace view, choosing from the nodes in the abstract syntax tree of the property.

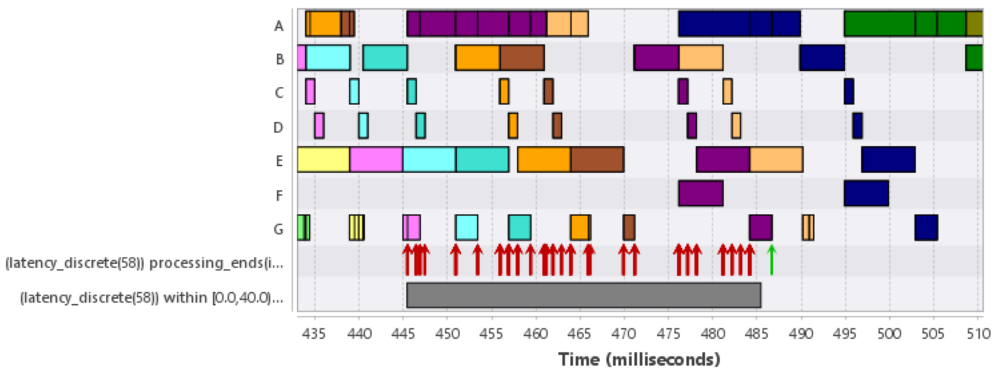


Figure 5.4: Explanation of the subformula `processing_ends(58)` and the interval of `within [0.0, 40.0) ms ...`

.

6 Causality for property explanation

In this chapter, the property explanation functionality of TRACE4CPS is extended by using an algorithm by Beer et al. to find the set of events and attributes that cause the violation of a property.

Section 6.1 summarizes [Beer et al. 2012], in particular the instance of HP causality and an approximation algorithm to compute the set of causes according to that definition.

In Section 6.2, the approximate cause algorithm is applied to properties and traces in TRACE-4CPS, modified to compute correct causes for STILL_FALSE properties and extended to MTL properties with timing constraints.

In Section 6.3, a new type of visual explanation is developed which uses the approximate cause algorithm to visualize the cause of a violation of an ETL property in TRACE4CPS.

6.1 HP causality for counterexample explanation

In model checking, a property is checked on a model of a system which represents all possible executions of the system. If the property is violated, a counterexample path is generated: a single execution that displays a violation of the property which can be used by the user to understand how the system violates the property.

Such a counterexample is often lengthy and determining the cause of the property violation is a time-consuming and error-prone process [Kaleeswaran et al. 2022].

6.1.1 Simplified binary causal models

An instance of HP causality (see Section 2.2.2) is used in [Beer et al. 2012] to identify the causes of a property violation in a counterexample for a LTL property. To make the connection between HP causality and the causes for LTL violations clearer, Beer et al. use a simplified definition of HP causality that is based on Definitions 2.12 to 2.15, but is restricted to binary causal models (i.e. models where the range of all variables is binary) and assumes there are no dependencies between the variables.

Definition 6.1 (Simplified binary model) [Beer et al. 2012]

A *binary causal model* M is a consists of a set $\mathcal{V}$ is a set of Boolean variables. A *context* $\vec{u}$ is a legal assignment for the variables in $\mathcal{V}$.

A primitive event has the form $X = x$, where $X \in \mathcal{V}$ is a variable, and x is either *true* or *false*. Given a Boolean function of primitive events η, $(M, \vec{u}) \models [\vec{Y} \leftarrow \vec{y}]\eta$ means that given context $\vec{u}$, η holds in the model $M_{\vec{Y} \leftarrow \vec{y}}$ in which the value of every variable Y is replaced with the corresponding value y.

The notation $\vec{X} \leftarrow \neg\vec{x}$ is used to say that the variables in $\vec{X}$ are set to their negated values, i.e. $X \leftarrow \neg x$ for every $X \in \vec{X}$ and x the value of X in $(M, \vec{u})$.

A *cause* in this simplified model is defined as follows:

> **Definition 6.2 (Cause for simplified binary models)** [Beer et al. 2012]
> $X = x$ is a *cause* of η in $(M, \vec{u})$ if the following conditions hold:
>
> AC1. $(M, \vec{u}) \models X = x$ and $(M, \vec{u}) \models \eta$
>
> AC2. There exists a subset $\vec{W}$ of $\mathcal{V}$ with $X \notin \vec{W}$ and $\vec{w}$ the values of $\vec{W}$ in $(M, \vec{u})$ such that the following conditions hold:
>
> (a) $(M, \vec{u}) \models [X \leftarrow \neg x, \vec{W} \leftarrow \neg\vec{w}]\neg\eta$. That is, changing the value of X from x to $\neg x$ and changing the values of all the variables in $\vec{W}$ together falsifies η in $(M, \vec{u})$.
>
> (b) For all $\vec{S} \subseteq \vec{W}$, with $\vec{s}$ denoting the values of $\vec{S}$ in $(M, \vec{u})$, we have $(M, \vec{u}) \models [\vec{S} \leftarrow \neg\vec{s}]\eta$. That is, changing the values of $\vec{W}$ or any subset of $\vec{W}$ without changing the value of X does not falsify η in $(M, \vec{u})$.

The notion of *criticality*, inspired by inspired by Definition 2.17, is defined as follows:

> **Definition 6.3 (Critical variable for simplified binary models)** [Beer et al. 2012]
>
> Let M be a model, $\vec{u}$ the current context, and η a Boolean formula. Let $(M, \vec{u}) \models \eta$, and X be a Boolean variable in M that has the value x in the context $\vec{u}$. $(X = x)$ is *critical* for η in $(M, \vec{u})$ if and only if $(M, \vec{u}) \models [X \leftarrow \neg x]\neg\eta$.
>
> That is, when changing the value of X to $\neg x$ falsifies η in $(M, \vec{u})$, $(X = x)$ is critical.

Intuitively, if there is a counterfactual dependence between $X = x$ and the truth value of η, we say that $X = x$ is critical for η. $X = x$ is a cause of η in $(M, \vec{u})$ if there exists a subset of variables such that changing their values makes $X = x$ critical for η.

6.1.2 Causality for LTL counterexamples

A *counterexample* to an LTL formula φ is a finite path $\sigma = s_0, \ldots s_n \in \Sigma^*$ such that $\sigma \not\models^- \varphi$, or an infinite path $\sigma = u \cdot v^\omega \in \Sigma^\omega$ consisting of a finite prefix u followed by repeated copies of a finite path v such that $\sigma \not\models^- \varphi$, where $\models^-$ is the weak satisfaction relation of [Eisner et al. 2003] (see Section 2.1.3).

In [Beer et al. 2012], counterexamples are generated by a model checker to illustrate the violation of an LTL property on model of a program. Programs are modeled as transition systems, and the states $s \in \Sigma$ in the counterexample path correspond to the states in the model. The states contain values for a set V of Boolean variables, and the set AP of atomic propositions in the counterexample path has a one-to-one correspondence to the set V of variables: given an atomic proposition p, $\sigma^i \models p$ if and only if the variable v corresponding to p is *true* in the state s_i of the model.

To find the *causes* of the failure of φ on σ, where a cause is a value of a variable v in a state s along the path σ, Beer et al. [2012] define an instance of HP causality in which a state-value pair $\langle s, v\rangle$ is a potential cause for the verdict $\sigma \not\models^- \varphi$ representing the violation of the property φ on the path σ.

The following notations are used in the definitions of criticality and causality. Given a pair $\langle s, v\rangle$ in σ, $\langle \hat{s}, v\rangle$ denotes the pair that is derived from $\langle s, v\rangle$ by switching the value of v in s from true to false or vice versa. Given a set A of pairs, $\hat{A}$ denotes the set that is obtained from A by replacing each $\langle s, v\rangle \in A$ with $\langle \hat{s}, v\rangle$. Given a path σ, a state s on σ, and a variable v, $\sigma^{\langle \hat{s}, v\rangle}$ denotes the path derived from σ by switching the value of v in s on σ. Similarly, given a set A of pairs, $\sigma^{\hat{A}}$ denotes the path obtained by replacing A with $\hat{A}$.

First, the definition of criticality described in Definition 6.3 is adapted to the domain of LTL counterexamples as follows:

Definition 6.4 (Critical value for LTL) [Beer et al. 2012]

A pair $\langle s, v\rangle$ is *critical* for the failure of φ on σ if $\sigma \not\models^{-} \varphi$, but $\sigma^{\langle \hat{s}, v\rangle} \models^{-} \varphi$. That is, $\langle s, v\rangle$ is critical if switching the value of v in s makes φ hold weakly on σ.

Note that in the place of a causal formula η is the *violation* of temporal formula in the weak semantics of LTL.

A cause is defined in terms of criticality:

Definition 6.5 (Cause for LTL) [Beer et al. 2012]

Let $\sigma \not\models^{-} \varphi$. A pair $\langle s, v\rangle$ in σ is a *cause for the failure of* φ *on* σ if there exists a set of pairs A such that $\langle s, v\rangle \notin A$, and the following hold:

(a) $\langle s, v\rangle$ is *critical* for $\sigma^{\hat{A}} \not\models^{-} \varphi$, and

(b) For all $S \subseteq A$, we have $\sigma^{\hat{S}} \not\models^{-} \varphi$.

In other words, a pair $\langle s, v\rangle$ is a cause if it *can be made critical* by switching the value of a set of pairs A, with the condition that no switching of A or any subset of A can itself cause φ to hold on σ. The set A is called a *witness set* for $\langle s, v\rangle$, σ, and φ, and there can be multiple witness sets for the same pair, formula and path.

To help find the set of causal state-value pairs, we will later make use of the following definition:

Definition 6.6 (Negation normal form for LTL)

An LTL formula φ is said to be in *negation normal form* (NNF) if it only uses the Boolean operators $\vee$, $\wedge$, the temporal operators G, U, X, and negations are only applied to atomic propositions.

Every LTL formula can be written as an equivalent formula in NNF. A *literal* is an LTL formula that consists either only of an atomic proposition, or the negation of an atomic proposition.

A consequence of the above definition of a cause is that given a formula φ in NNF and an atomic proposition p, if $\langle s, p\rangle$ is a cause for $\sigma \not\models^{-} \varphi$ then there exists a literal l in φ (where l is either p or $\neg p$) such that $s \not\models l$. That is, a pair $\langle s, p\rangle$ can be a cause only if the corresponding literal has the value *false* in s. If $\sigma \not\models^{-} \varphi$ and a literal l has the value *true* in some state, switching its value to *false* cannot make φ hold on σ.

6.1.3 An approximation algorithm for causality

Beer et al. [2012] conjecture that computing causality in their framework is Σ_2^P-hard, and hence intractable for all but short sequences and simple formulas. To remedy this, they present an approximation algorithm, Algorithm 6.7, which produces an approximation of the causal set according to the definition above.

The algorithm is based on the observation made earlier that given a formula in negation normal form, only a false-valued literal can cause the formula to be violated.

Because each subformula ψ of φ is evaluated at most once at each state s_i of the counterexample σ, the computational complexity of the procedure C is linear in n and in $|\varphi|$.

The algorithm doesn't necessarily visit all pairs $\langle s, v\rangle$ in σ: it computes the causes of the *first failure* of φ on σ [Beer et al. 2009]. For example, for the formula $\mathsf{G}\,a$ on the path s_0, s_1, s_2 labeled $\{a\}, \{\neg a\}, \{\neg a\}$, the algorithm would return the pair $\langle s_1, a\rangle$ only, even though according to Definition 6.5, the pair $\langle s_2, a\rangle$ should be a cause as well (since it becomes critical if the value of a is switched in s_1).

However, note that a single failure can have many causes that lead up to it. For example, for the (single) failure of the formula $a\,\mathsf{U}\,b$ on the path s_0, s_1, s_2 labeled $\{a\}, \{a\}, \{a\}$, the three causes $\{\langle s_0, b\rangle, \langle s_1, b\rangle, \langle s_2, b\rangle\}$ are generated.

When restricted to the first failure of φ on σ, the set C computed by Algorithm 6.7 *over-approximates* the set of causes according to Definition 6.5. For example, consider the unsatisfiable property $\varphi = p \wedge \neg p$ and a path $\sigma = s_0$ with $L(s_0) = \varnothing$. The property is false ($\sigma \not\models^- \varphi$), and $\langle s_0, p\rangle$ is given as a cause in C, but if the value of the proposition p were flipped in s_0, the property would still be false, so it is not a cause according to Definition 6.5.

The algorithm was implemented in a tool called RuleBase PE (see Section 2.3), IBM's formal verification platform, and user feedback indicated that the visualization of the causal set was seen as helpful and important.

Algorithm 6.7 (An approximation of the causal set) [Beer et al. 2012]

Let φ be a formula given in NNF, and let $\sigma = s_0, s_1, \ldots, s_n$ be a non-empty counterexample of length $n+1$ for φ. The procedure C produces a set $C(\sigma^i, \psi)$, an approximation of the set of causes for the failure of a (sub)formula ψ (that is in NNF) on a suffix of σ that starts at s_i. It is invoked as $C(\sigma^0, \varphi)$ to produce the approximate set of causes for the failure of φ on σ.

First, the auxiliary function $val(\sigma^i, \psi)$ is defined, which evaluates subformulas of ψ on the given path. It returns 0 if the subformula fails on the path and 1 otherwise. The value of *val* is computed as follows:

$$\begin{aligned}
val(\sigma^i, \mathsf{true}) &= 1\\
val(\sigma^i, \neg\mathsf{true}) &= 0\\
val(\sigma^i, \varphi) &= \begin{cases} 1 & \text{if } C(\sigma^i, \varphi) = \varnothing \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

To compute the value of C, at each step of the algorithm, when the value of a subformula ψ on a suffix σ^j is determined, the value is saved for future use in either C or *val* so that each subformula is evaluated at most once in each state. Apart from this detail, the value of C is computed as follows:

$$\begin{aligned}
C(\sigma^i, \mathsf{true}) &= \varnothing\\
C(\sigma^i, \neg\mathsf{true}) &= \varnothing\\
C(\sigma^i, p) &= \begin{cases} \{\langle s_i, p\rangle\} & \text{if } L(\langle s_i, p\rangle) = 0 \\ \varnothing & \text{otherwise} \end{cases}\\
C(\sigma^i, \neg p) &= \begin{cases} \{\langle s_i, p\rangle\} & \text{if } L(\langle s_i, p\rangle) = 1 \\ \varnothing & \text{otherwise} \end{cases}\\
C(\sigma^i, \varphi_1 \wedge \varphi_2) &= C(\sigma^i, \varphi_1) \cup C(\sigma^i, \varphi_2)\\
C(\sigma^i, \varphi_1 \vee \varphi_2) &= \begin{cases} C(\sigma^i, \varphi_1) \cup C(\sigma^i, \varphi_2) & \text{if } val(\sigma^i, \varphi_1) = 0 \text{ and } val(\sigma^i, \varphi_2) = 0 \\ \varnothing & \text{otherwise} \end{cases}\\
C(\sigma^i, \mathsf{X}\,\varphi) &= \begin{cases} C(\sigma^{i+1}, \varphi) & \text{if } i < n \\ \varnothing & \text{otherwise} \end{cases}\\
C(\sigma^i, \mathsf{G}\,\varphi) &= \begin{cases} C(\sigma^i, \varphi) & \text{if } val(\sigma^i, \varphi) = 0 \\ C(\sigma^{i+1}, \mathsf{G}\,\varphi) & \text{if } val(\sigma^i, \varphi) = 1 \text{ and } i < n \\ & \text{and } val(\sigma^i, \mathsf{X}\,\mathsf{G}\,\varphi) = 0 \\ \varnothing & \text{otherwise} \end{cases}\\
C(\sigma^i, \varphi_1 \,\mathsf{U}\, \varphi_2) &= \begin{cases} C(\sigma^i, \varphi_2) \cup C(\sigma^i, \varphi_1) & \text{if } val(\sigma^i, \varphi_1) = 0 \text{ and } val(\sigma^i, \varphi_2) = 0 \\ C(\sigma^i, \varphi_2) & \text{if } val(\sigma^i, \varphi_1) = 1 \text{ and } val(\sigma^i, \varphi_2) = 0 \\ & \text{and } i = n \\ C(\sigma^i, \varphi_2) \cup C(\sigma^{i+1}, \varphi_1 \,\mathsf{U}\, \varphi_2) & \text{if } val(\sigma^i, \varphi_1) = 1 \text{ and } val(\sigma^i, \varphi_2) = 0 \\ & \text{and } i < n \text{ and } val(\sigma^i, \mathsf{X}[\varphi_1 \,\mathsf{U}\, \varphi_2]) = 0 \\ \varnothing & \text{otherwise} \end{cases}
\end{aligned}$$

6.2 Causality for trace explanation in TRACE4CPS

The approach of Beer et al. described in Section 6.1 concerns the explanation of counterexamples generated by model checkers. However, it can also be used in the context of runtime verification in TRACE4CPS to obtain concise explanations of violations of ETL properties on execution traces, which are often longer and contain more irrelevant information than model checker-generated counterexample paths.

In Section 6.2.1, the approximate causal set algorithm is implemented in TRACE4CPS for properties with no timing constraints (i.e. those that can be expressed in LTL).

The algorithm gives expected causes for many, but not all, properties with a STILL_FALSE verdict. In Section 6.2.2, the algorithm is modified slightly to operate on the finite semantics instead of the weak semantics of LTL in order to generate correct results for properties that are STILL_FALSE.

Finally, in Section 6.2.3, the explanation functionality is extended to work on MTL properties with real-time constraints. This was done by adding the cause generation to the (finite) MTL checking algorithm.

6.2.1 Causes for LTL formulas with FALSE verdicts

In TRACE4CPS, both events in a trace and atomic propositions in an MTL property are *attribute mappings*, and an atomic proposition $p \in \mathbf{M}$ holds on an event $s \in \mathbf{M}$ if for every attribute $a \in \mathbf{A}$ that is specified in s, the value $s(a) \in \mathbf{V}$ equals the value $p(a)$ of that attribute in p. Equivalently, $s \models p \iff p \subseteq s$.

As a consequence, given an atomic proposition $p = \{a_0 \mapsto v_0, a_1 \mapsto v_1, \dots\}$ with $a_0, a_1, \dots \in \mathbf{A}$ and $v_0, v_1, \dots \in \mathbf{V}$, we can say that $s \models p$ is equivalent to $s \models \{a_0 \mapsto v_0\} \wedge \{a_1 \mapsto v_1\} \wedge \dots$.

This gives us the flexibility to decide whether a cause generated by the causal set algorithm corresponds to the atomic propositions specified in the formula (combined with an event) or to the values of the individual attributes of an event. Whether decomposing the propositions in a formula into a conjunction of individual attribute-value pairs makes the resulting explanation clearer or not depends on the trace and property, so the decision to generate an explanation on the level of attribute-value pairs or attribute mappings is left to the user.

In the initial implementation of the causal set algorithm, only properties without timing constraints (LTL properties, or MTL formulas where the intervals of all temporal operators are $[0, \infty)$) are supported, and the (four-valued) verdict of the property φ on the trace σ is assumed to be FALSE, so we can assume that both $\sigma \not\models^{-} \varphi$ and $\sigma \not\models_f \varphi$.

The syntax of LTL properties on which this initial implementation of the algorithm operates follows the following grammar:

Definition 6.8 (Syntax of LTL in negation normal form with X, U and G)

$$\varphi := \mathsf{true} \mid \neg\mathsf{true} \mid p \mid \neg p \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \mathsf{X}\,\varphi \mid \mathsf{G}\,\varphi \mid \varphi_1 \,\mathsf{U}\, \varphi_2$$

The (untimed) *next* operator $\mathsf{X}\,\varphi$ was not originally a part of the syntax of properties in TRACE-4CPS (Definition 3.3), but was included in both the syntax of ETL properties and that of the properties that the causal set algorithm operates on because it is used in algorithm C and in some example properties from [Beer et al. 2009] and [Beer et al. 2012].

The *globally* operator $\mathsf{G}\,\varphi$ is present in the syntax of ETL properties (as `globally` ...), but translated to formulas of the form $\neg(\mathsf{true}\,\mathsf{U}\,\varphi)$ internally in TRACE4CPS. It was included as a separate form for the causal set algorithm.

A transformation procedure from untimed MTL formulas to formulas according to Definition 6.8 was created by using the following equivalences as transformation rules:

Proposition 6.9 (Equivalences for LTL NNF transformation)

$$\begin{aligned} \neg\neg a &\equiv a & \neg\,\mathsf{X}\,a &\equiv \mathsf{X}\,\neg a && (1)\\ \neg(a \vee b) &\equiv a \wedge b & \neg(a\,\mathsf{U}\,b) &\equiv (\neg b\,\mathsf{U}\,(\neg a \wedge \neg b)) \vee \mathsf{G}\,\neg b && (2)\\ \neg(a \wedge b) &\equiv a \vee b & \neg\,\mathsf{G}\,a &\equiv \mathsf{true}\,\mathsf{U}\,\neg a && (3)\\ a \rightarrow b &\equiv \neg a \vee b \end{aligned}$$

The equivalence for X (1) is valid in the semantics of LTL for infinite sequences, but not for finite sequences (see Section 2.1.2). However, the formulas for which we are trying to find the causes are assumed to be violated in the weak semantics. Because in the weak semantics, every proposition on a state past the end of the sequence is true, the next operator X is equivalent to the weak next operator $\overline{\mathsf{X}}$. Therefore, in the formulas and traces on which the algorithm operates, equivalence (1) is valid.

The equivalence labeled (2) was derived as follows. There are two ways for a formula of the form $a\,\mathsf{U}\,b$ to be violated: either b never holds ($\mathsf{G}\,\neg b$), or there is some state before the first occurrence of b where neither a nor b holds ($\neg b\,\mathsf{U}\,(\neg a \wedge \neg b)$).

Equivalence (3) is included for the sake of completeness, but not actually used in the transformation: the formulas used by TRACE4CPS do not contain G directly, but $\mathsf{G}\,\varphi$ is encoded as $\neg(\mathsf{true}\,\mathsf{U}\,\neg\varphi)$, which is covered by equivalence (2), and the instances of G generated by the NNF transformation are already in NNF.

The following two equivalences can optionally be included in the transformation:

$$\neg(\mathsf{true}\,\mathsf{U}\,a) \equiv \mathsf{G}(\neg a) \quad (4)$$
$$\neg(\neg\neg a\,\mathsf{U}\,b) \equiv \neg(a\,\mathsf{U}\,b) \quad (5)$$

These are not strictly necessary to obtain an NNF formula, since they are already covered by equivalence (2), but by including equivalence (4) as a rule (with higher precedence than (2)), the generated formulas are shorter, and equivalence (5) (also with higher precedence than (2)) makes sure that equivalence (4) works even when the true part of the formula is obscured by double negations.

This results in the NNF transformation algorithm shown in Algorithm 6.10.

The approximate cause algorithm described in Section 6.1.3 was then implemented to generate the approximate set of causes for a violation in the weak semantics of the formulas described above on a trace. The C and *val* functions in Algorithm 6.7 were merged into a single C function which takes a boolean parameter *record_causes* so that causes are only generated when necessary. The resulting algorithm is described in Algorithm 6.11.

The algorithm memoizes the values of the subformulas, but not the causal sets. This is a trade-off: it has to compute and keep around fewer unneeded causal sets, but means that in the case of $\vee$, G and U it may evaluate a subformula on a state multiple times. This approach was chosen so that the algorithm is easier to integrate with the MTL checking algorithm later.

Algorithm 6.10 (NNF transformation for LTL with U and G)

$$
\begin{aligned}
&\mathit{nnf}(\mathsf{true}) &&= \mathsf{true} \\
&\mathit{nnf}(\neg\mathsf{true}) &&= \neg\mathsf{true} \\
&\mathit{nnf}(\neg\neg\varphi) &&= \mathit{nnf}(\varphi) \\
&\mathit{nnf}(\varphi_1 \lor \varphi_2) &&= \mathit{nnf}(\varphi_1) \lor \mathit{nnf}(\varphi_2) \\
&\mathit{nnf}(\neg(\varphi_1 \lor \varphi_2)) &&= \mathit{nnf}(\neg\varphi_1) \land \mathit{nnf}(\neg\varphi_2) \\
&\mathit{nnf}(\varphi_1 \land \varphi_2) &&= \mathit{nnf}(\varphi_1) \land \mathit{nnf}(\varphi_2) \\
&\mathit{nnf}(\neg(\varphi_1 \land \varphi_2)) &&= \mathit{nnf}(\neg\varphi_1) \lor \mathit{nnf}(\neg\varphi_2) \\
&\mathit{nnf}(\varphi_1 \to \varphi_2) &&= \mathit{nnf}(\neg\varphi_1) \lor \mathit{nnf}(\varphi_2) \\
&\mathit{nnf}(\neg(\varphi_1 \to \varphi_2)) &&= \mathit{nnf}(\varphi_1) \land \mathit{nnf}(\neg\varphi_2) \\
&\mathit{nnf}(\mathsf{X}\,\varphi_1) &&= \mathsf{X}\,\mathit{nnf}(\varphi_1) \\
&\mathit{nnf}(\neg\,\mathsf{X}\,\varphi_1) &&= \mathsf{X}\,\mathit{nnf}(\neg\varphi_1) && (1) \\
&\mathit{nnf}(\varphi_1 \,\mathsf{U}\, \varphi_2) &&= \mathit{nnf}(\varphi_1) \,\mathsf{U}\, \mathit{nnf}(\varphi_2) \\
&\mathit{nnf}(\neg(\mathsf{true} \,\mathsf{U}\, \varphi)) &&= \mathsf{G}(\mathit{nnf}(\neg\varphi)) && (4) \\
&\mathit{nnf}(\neg(\neg\neg\varphi_1 \,\mathsf{U}\, \varphi_2)) &&= \mathit{nnf}(\neg(\varphi_1 \,\mathsf{U}\, \varphi_2)) && (5) \\
&\mathit{nnf}(\neg(\varphi_1 \,\mathsf{U}\, \varphi_2)) &&= [\mathit{nnf}(\neg\varphi_2) \,\mathsf{U}\, (\mathit{nnf}(\neg\varphi_1) \land \mathit{nnf}(\neg\varphi_2))] \lor \mathsf{G}\,\mathit{nnf}(\neg\varphi_2) && (2)
\end{aligned}
$$

Depending on the user's preference to view causes as attribute mappings (m) or single attributes (a), the atomic propositions may be decomposed into conjunctions of atomic propositions consisting of a single attribute-value pair, as described before:

$$
\begin{aligned}
&\mathit{nnf}^m(p) &&= p \\
&\mathit{nnf}^m(\neg p) &&= \neg p
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{nnf}^a(\{a_0 \mapsto v_0, a_1 \mapsto v_1, \dots\}) &&= \{a_0 \mapsto v_0\} \land \{a_1 \mapsto v_1\} \land \dots \\
&\mathit{nnf}^a(\neg\{a_0 \mapsto v_0, a_1 \mapsto v_1, \dots\}) &&= \neg\{a_0 \mapsto v_0\} \lor \neg\{a_1 \mapsto v_1\} \lor \dots
\end{aligned}
$$

Algorithm 6.11 (Approximate causes for LTL with X, U and G)

Let $\sigma = s_0, \dots, s_n$ be a finite sequence of length $n+1$ and φ an LTL formula in negation normal form. The memoization table *val_table* contains for every pair i, φ one of $\{\mathit{true}, \mathit{false}\}$ if φ was evaluated on σ^i, or ? otherwise. If *record_causes* = *true*, C will return the validity of φ on σ^i and add the causes of a failure of φ on σ^i to *causes*. If *record_causes* = *false*, C will return the validity without adding to *causes*.

The algorithm is invoked as $\mathrm{C}(\sigma^0, \varphi, \mathit{true})$, and when it returns, *causes* contains the approximate set of causes for the failure of φ on σ.

```
1:  $causes \leftarrow \emptyset$
2:  $val\_table(i, \varphi) \leftarrow$ ? for all $i, \varphi$
3:  procedure $\mathrm{C}(\sigma^i, \varphi, record\_causes)$
4:      $v \leftarrow val\_table(i, \varphi)$
5:      if $v = ?$ or $record\_causes$ then
6:          if $\varphi = \mathsf{true}$ then
7:              $v \leftarrow true$
8:          else if $\varphi = \neg\mathsf{true}$ then
9:              $v \leftarrow false$
10:         else if $\varphi = p$ then
11:             if $p \notin s_i$ then
12:                 if $record\_causes$ then
13:                     $causes \leftarrow causes \cup \{\langle s_i, p \rangle\}$
14:                 $v \leftarrow false$
15:             else
16:                 $v \leftarrow true$
17:         else if $\varphi = \neg p$ then
18:             if $p \in s_i$ then
19:                 if $record\_causes$ then
20:                     $causes \leftarrow causes \cup \{\langle s_i, p \rangle\}$
21:                 $v \leftarrow false$
22:             else
23:                 $v \leftarrow true$
24:         else if $\varphi = \varphi_1 \wedge \varphi_2$ then
25:             $v_1 \leftarrow \mathrm{C}(\sigma^i, \varphi_1, record\_causes)$
26:             $v_2 \leftarrow \mathrm{C}(\sigma^i, \varphi_2, record\_causes)$
27:             $v \leftarrow v_1 \wedge v_2$
28:         else if $\varphi = \varphi_1 \vee \varphi_2$ then
29:             if $\neg\mathrm{C}(\sigma^i, \varphi_1, false)$ and $\neg\mathrm{C}(\sigma^i, \varphi_2, false)$ then
30:                 if $record\_causes$ then
31:                     $\mathrm{C}(\sigma^i, \varphi_1, true)$
32:                     $\mathrm{C}(\sigma^i, \varphi_2, true)$
33:                 $v \leftarrow false$
34:             else
35:                 $v \leftarrow true$
36:         else if $\varphi = \mathsf{X}\,\varphi_1$ then
37:             if $i < n$ then
38:                 $v \leftarrow \mathrm{C}(\sigma^{i+1}, \varphi_1, record\_causes)$
39:             else
40:                 $v \leftarrow true$
```

```
        else if $\varphi = \mathsf{G}\,\varphi_1$ then
            $v_1 \leftarrow \mathrm{C}(\sigma^i, \varphi_1, \mathit{false})$
            if $\neg v_1$ then
                if $\mathit{record\_causes}$ then
                    $\mathrm{C}(\sigma^i, \varphi_1, \mathit{true})$
                $v \leftarrow \mathit{false}$
            else if $v_1$ and $i < n$ and $\neg\mathrm{C}(\sigma^i, \mathsf{X}\,\varphi, \mathit{false})$ then
                if $\mathit{record\_causes}$ then
                    $\mathrm{C}(\sigma^{i+1}, \varphi, \mathit{true})$
                $v \leftarrow \mathit{false}$
            else
                $v \leftarrow \mathit{true}$
        else if $\varphi = \varphi_1 \,\mathsf{U}\, \varphi_2$ then
            $v_1 \leftarrow \mathrm{C}(\sigma^i, \varphi_1, \mathit{false})$
            $v_2 \leftarrow \mathrm{C}(\sigma^i, \varphi_2, \mathit{false})$
            if $\neg v_1$ and $\neg v_2$ then
                if $\mathit{record\_causes}$ then
                    $\mathrm{C}(\sigma^i, \varphi_2, \mathit{true})$
                    $\mathrm{C}(\sigma^i, \varphi_1, \mathit{true})$
                $v \leftarrow \mathit{false}$
            else if $v_1$ and $\neg v_2$ and $i = n$ then
                if $\mathit{record\_causes}$ then
                    $\mathrm{C}(\sigma^i, \varphi_2, \mathit{true})$
                $v \leftarrow \mathit{false}$
            else if $v_1$ and $\neg v_2$ and $i < n$ and $\neg\mathrm{C}(\sigma^i, \mathsf{X}\,\varphi, \mathit{false})$ then
                if $\mathit{record\_causes}$ then
                    $\mathrm{C}(\sigma^i, \varphi_2, \mathit{true})$
                    $\mathrm{C}(\sigma^{i+1}, \varphi, \mathit{true})$
                $v \leftarrow \mathit{false}$
            else
                $v \leftarrow \mathit{true}$
        $\mathit{val\_table}(i, \varphi) \leftarrow v$
    return $v$
```

⋆ ⋆ ⋆

A modification to Algorithm 6.11 was made in the cases for G and U in Algorithm 6.7. In the second condition for a *globally* formula and the third condition of an *until* formula, a formula $\mathsf{X}\,\mathsf{G}\,\varphi$ and a formula $\mathsf{X}[\varphi_1 \,\mathsf{U}\, \varphi_2]$ are constructed, respectively, in order to recurse to the next state to check whether a temporal formula is violated later in the trace.

Looking at Algorithm 6.7, because these expressions are only evaluated when $i < n$, we know that the expression $C(\sigma^i, \mathsf{X}\,\varphi)$ will evaluate to $C(\sigma^{i+1}, \varphi)$. Thus, the expression $\mathit{val}(\sigma^i, \mathsf{X}\,\mathsf{G}\,\varphi) = 0$ can safely be replaced with $\mathit{val}(\sigma^{i+1}, \mathsf{G}\,\varphi) = 0$, and $\mathit{val}(\sigma^i, \mathsf{X}[\varphi_1 \,\mathsf{U}\, \varphi_2]) = 0$ with $\mathit{val}(\sigma^{i+1}, \varphi_1 \,\mathsf{U}\, \varphi_2) = 0$.

The updated cases for *globally* and *finally* formulas in Algorithm 6.7 and Algorithm 6.11 are shown in Algorithm 6.12, and in pseudocode in Algorithm 6.14 (with the modified expressions highlighted using boxes).

Algorithm 6.12 (Approximate causes for LTL, modified)

The cases for *globally* and *until* are modified as follows:

$$C(\sigma^i, \mathsf{G}\,\varphi) = \begin{cases} C(\sigma^i, \varphi) & \text{if } \mathit{val}(\sigma^i, \varphi) = 0 \\ C(\sigma^{i+1}, \mathsf{G}\,\varphi) & \text{if } \mathit{val}(\sigma^i, \varphi) = 1 \text{ and } i < n \\ & \text{and } \boxed{\mathit{val}(\sigma^{i+1}, \mathsf{G}\,\varphi) = 0} \\ \varnothing & \text{otherwise} \end{cases}$$

$$C(\sigma^i, \varphi_1 \,\mathsf{U}\, \varphi_2) = \begin{cases} C(\sigma^i, \varphi_2) \cup C(\sigma^i, \varphi_1) & \text{if } \mathit{val}(\sigma^i, \varphi_1) = 0 \text{ and } \mathit{val}(\sigma^i, \varphi_2) = 0 \\ C(\sigma^i, \varphi_2) & \text{if } \mathit{val}(\sigma^i, \varphi_1) = 1 \text{ and } \mathit{val}(\sigma^i, \varphi_2) = 0 \\ & \text{and } i = n \\ C(\sigma^i, \varphi_2) \cup C(\sigma^{i+1}, \varphi_1 \,\mathsf{U}\, \varphi_2) & \text{if } \mathit{val}(\sigma^i, \varphi_1) = 1 \text{ and } \mathit{val}(\sigma^i, \varphi_2) = 0 \\ & \text{and } i < n \text{ and } \boxed{\mathit{val}(\sigma^{i+1}, \varphi_1 \,\mathsf{U}\, \varphi_2) = 0} \\ \varnothing & \text{otherwise} \end{cases}$$

Property-based testing was used to verify that this results in identical causes being generated for all LTL properties with relatively high confidence (see Section 7.4).

Not all properties and traces with a FALSE verdict have a cause. The simplest example is the property $\neg\mathsf{true}$. It is false on any trace, but because there are no atomic propositions whose value can be flipped in any state in the trace, the set of causes according to Definition 6.5 is empty, and the approximation algorithm finds no causes either.

However, disregarding those properties that have no causes for violations because of the use of true, the following proposition was determined to hold with relatively high confidence using property-based testing.

Proposition 6.13 (Existence of causes for FALSE, LTL)

Let φ be an LTL property that does not contain true except as the left side of an U (so that F and G are allowed), and σ a finite sequence. If $\sigma \not\models^- \varphi$, then the approximate causal set found by Algorithms 6.11 and 6.14 is non-empty.

That is, for such a property, if the verdict is FALSE Algorithms 6.11 and 6.14 always find at least one causal state-value pair.

Furthermore, for all properties that have either a STILL_TRUE or TRUE verdict, the approximate causal set generated by Algorithms 6.11 and 6.14 is empty.

Algorithm 6.14 (Approximate causes for LTL, modified)

```
if $\varphi = \mathsf{G}\,\varphi_1$ then
    $v_1 \leftarrow \mathrm{C}(\sigma^i, \varphi_1, \mathit{false})$
    if $\neg v_1$ then
        if $\mathit{record\_causes}$ then
            $\mathrm{C}(\sigma^i, \varphi_1, \mathit{true})$
        $v \leftarrow \mathit{false}$
    else if $v_1$ and $i < n$ and $\boxed{\neg\mathrm{C}(\sigma^{i+1}, \varphi, \mathit{false})}$ then
        if $\mathit{record\_causes}$ then
            $\mathrm{C}(\sigma^{i+1}, \varphi, \mathit{true})$
        $v \leftarrow \mathit{false}$
    else
        $v \leftarrow \mathit{true}$
else if $\varphi = \varphi_1\,\mathsf{U}\,\varphi_2$ then
    $v_1 \leftarrow \mathrm{C}(\sigma^i, \varphi_1, \mathit{false})$
    $v_2 \leftarrow \mathrm{C}(\sigma^i, \varphi_2, \mathit{false})$
    if $\neg v_1$ and $\neg v_2$ then
        if $\mathit{record\_causes}$ then
            $\mathrm{C}(\sigma^i, \varphi_2, \mathit{true})$
            $\mathrm{C}(\sigma^i, \varphi_1, \mathit{true})$
        $v \leftarrow \mathit{false}$
    else if $v_1$ and $\neg v_2$ and $i = n$ then
        if $\mathit{record\_causes}$ then
            $\mathrm{C}(\sigma^i, \varphi_2, \mathit{true})$
        $v \leftarrow \mathit{false}$
    else if $v_1$ and $\neg v_2$ and $i < n$ and $\boxed{\neg\mathrm{C}(\sigma^{i+1}, \varphi, \mathit{false})}$ then
        if $\mathit{record\_causes}$ then
            $\mathrm{C}(\sigma^i, \varphi_2, \mathit{true})$
            $\mathrm{C}(\sigma^{i+1}, \varphi, \mathit{true})$
        $v \leftarrow \mathit{false}$
    else
        $v \leftarrow \mathit{true}$
```

6.2.2 Causes for LTL formulas with STILL_FALSE verdicts

In [Beer et al. 2012], a counterexample to an LTL formula φ is a finite path $\sigma = s_0, \dots s_n$ such that $\sigma \not\models^{-} \varphi$, or an infinite path $\sigma = u \cdot v^{\omega}$ consisting of a finite prefix u followed by repeated copies of a finite path v such that $\sigma \not\models^{-} \varphi$. The weak semantics are used so that if φ is violated, assuming σ is a finite path, causes of the violation can always be found at some point in the sequence. In contrast, if the finite semantics were used, the cause of the violation of φ on a finite path σ such that $\sigma \not\models_f \varphi$ may be that the sequence is not long enough to satisfy φ, which cannot be expressed as a causal state-value pair.

For example, let $\varphi = \mathsf{X}\, a$ and $\sigma = s_0 = \{a\}$. Clearly, σ does not satisfy φ in the finite semantics, because $\mathsf{X}\, a$ requires that there is a next state for a to hold. However, there is no pair $\langle s_i, v\rangle$ such that changing the labeling of s_i (with any other set of states) will make σ satisfy φ, so the set of causes for the violation of φ is empty.

Disregarding properties where the cause lies outside the trace or that are unsatisfiable, the causal set algorithm as defined in Section 6.2.1 generates causes for all properties that are violated in the weak semantics (and therefore have a FALSE verdict). Furthermore, the cases for G and U operators in Algorithm 6.11 (Algorithm 6.14) are such that causes are also generated for most properties that are not violated in the weak semantics but are violated in the finite semantics (and therefore have a STILL_FALSE verdict). For example, given the property $\mathsf{F}\, a$ on a trace s_0, s_1, s_2 where each state is labeled $L(s) = \varnothing$, the algorithm correctly identifies that switching the value of a in any of s_1, s_2 and s_3 would satisfy the property.

Nevertheless, there are properties that have a STILL_FALSE verdict for which no causes are found, yet there are state-value pairs that when flipped would cause the property to be STILL_TRUE or TRUE. For example, consider the property $\mathsf{F}\,\mathsf{X}\, \neg a = \mathsf{true}\, \mathsf{U}\, \mathsf{X}\, \neg a$ on the following sequence:

$$\sigma = s_0, s_1 = \{a\}, \{a\}$$

The verdict (in the four-valued semantics) is STILL_FALSE, since it is true in the weak semantics but false in the finite semantics. The property is not violated in the weak semantics, and no causes are given by Algorithm 6.11, but if a were false in s_1, the verdict would have been TRUE.

To find causes for properties such as this, we first amend Definitions 6.4 and 6.5 to use the finite semantics instead of the weak semantics:

Definition 6.15 (Critical value for LTL$_f$) [Beer et al. 2012]

A pair $\langle s, v\rangle$ is *critical* for the failure of φ on σ if $\sigma \not\models_f \varphi$, but $\sigma^{\langle \hat{s}, v\rangle} \models_f \varphi$. That is, $\langle s, v\rangle$ is critical if switching the value of v in s makes φ hold on σ in the finite semantics.

Definition 6.16 (Cause for LTL$_f$) [Beer et al. 2012]

Let $\sigma \not\models_f \varphi$. A pair $\langle s, v\rangle$ in σ is a *cause for the failure of* φ *on* σ if there exists a set of pairs A such that $\langle s, v\rangle \notin A$, and the following hold:

(a) $\langle s, v\rangle$ is *critical* for $\sigma^{\hat{A}} \not\models_f \varphi$, and

(b) For all $S \subseteq A$, we have $\sigma^{\hat{S}} \not\models_f \varphi$

To modify the algorithm so that it approximates Definition 6.16 instead of Definition 6.5, recall from Section 2.1.2 that in the finite semantics the strong next operator X and weak next operator $\overline{\mathsf{X}}$ are distinguished. The weak next operator is usually defined in terms of the strong next operator ($\overline{\mathsf{X}}\,\varphi \overset{\text{def}}{=} \neg\,\mathsf{X}\,\neg\varphi$), but because this representation is not in negation normal form, we need to add it as a core operator to the syntax of properties.

This results in the following syntax for LTL$_f$ formulas in negation normal form:

Definition 6.17 (Syntax of LTL$_f$ in negation normal form with X, $\overline{\mathsf{X}}$, U and G)

$$\varphi \coloneqq \mathsf{true} \mid \neg\mathsf{true} \mid p \mid \neg p \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \mathsf{X}\,\varphi \mid \overline{\mathsf{X}}\,\varphi \mid \mathsf{G}\,\varphi \mid \varphi_1\,\mathsf{U}\,\varphi_2$$

The semantics for the weak next operator on which we base our algorithm holds no surprises:

$$\sigma^i \models_f \overline{\mathsf{X}}\,\varphi \quad \text{iff} \quad i = n \text{ or } \sigma^{i+1} \models_f \varphi$$

For LTL$_f$ formulas, equivalence (1) of Proposition 6.9 is replaced by the following:

$$\neg\,\mathsf{X}\,a \equiv \overline{\mathsf{X}}\,\neg a$$
$$\neg\,\overline{\mathsf{X}}\,a \equiv \mathsf{X}\,\neg a$$

Because formulas used by TRACE4CPS do not contain $\overline{\mathsf{X}}$ directly, and instances of $\overline{\mathsf{X}}$ generated by the NNF transformation are already in NNF, only the first needs to be used in the NNF transformation algorithm. Thus, in the NNF transformation for LTL$_f$ formulas, rule (1) is replaced by the following rule:

Algorithm 6.18 (NNF transformation for LTL$_f$ with U and G)

$$nnf(\neg\,\mathsf{X}\,\varphi_1) = \overline{\mathsf{X}}\,nnf(\neg\varphi_1)$$

The rest is the same as Algorithm 6.10.

Algorithm 6.7 is extended so that the strong next operator receives a special case in the *val* function, and a case for the weak next operator is added to C:

Algorithm 6.19 (Approximate causes for LTL$_f$)
The *val* function receives one special case for the *strong next* operator:

$$\begin{aligned}
val(\sigma^i, \mathsf{true}) &= 1\\
val(\sigma^i, \neg\mathsf{true}) &= 0\\
val(\sigma^i, \mathsf{X}\,\varphi) &= \begin{cases} 1 & \text{if } i < n \text{ and } C(\sigma^i, \mathsf{X}\,\varphi) = \varnothing \\ 0 & \text{otherwise} \end{cases}\\
val(\sigma^i, \varphi) &= \begin{cases} 1 & \text{if } C(\sigma^i, \varphi) = \varnothing \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

In C, the case for $\overline{\mathsf{X}}$ is identical to that of X:

$$C(\sigma^i, \overline{\mathsf{X}}\,\varphi) = \begin{cases} C(\sigma^{i+1}, \varphi) & \text{if } i < n \\ \varnothing & \text{otherwise} \end{cases}$$

The cases for both the strong and weak next operators were implemented in Algorithm 6.11 (and Algorithm 6.14) as follows (with modifications highlighted):

Algorithm 6.20 (Approximate causes for LTL$_f$)

```
1: if $\boxed{\varphi = \overline{\mathsf{X}}\,\varphi_1}$ then
2:     if $i < n$ then
3:         $v \leftarrow \mathrm{C}(\sigma^{i+1}, \varphi_1, \mathit{record\_causes})$
4:     else
5:         $v \leftarrow \mathit{true}$
6: else if $\varphi = \mathsf{X}\,\varphi_1$ then
7:     if $i < n$ then
8:         $v \leftarrow \mathrm{C}(\sigma^{i+1}, \varphi_1, \mathit{record\_causes})$
9:     else
10:        $\boxed{v \leftarrow \mathit{false}}$
```

With these modifications, if a *next* operator is evaluated on the last state in a sequence, resulting in a STILL_FALSE verdict, causes (other than the non-existent successor state of the last state, which cannot be a cause) are correctly generated.

Let us revisit the example shown earlier of the property $\mathsf{F}\,\mathsf{X}\,\neg a = \mathsf{true}\,\mathsf{U}\,\mathsf{X}\,\neg a$ on the following sequence:

$$\sigma = s_0, s_1 = \{a\}, \{a\}$$

The verdict (in the four-valued semantics) is STILL_FALSE, since it is true in the weak semantics but false in the finite semantics.

When using Algorithm 6.11, no causes were found, but with the modifications in Algorithm 6.20, $\langle s_1, a\rangle$ is correctly identified as a cause in the finite semantics.

To see why the distinction between X and $\overline{\mathsf{X}}$ is important in the generation of causes, consider the property $\mathsf{F}\,\neg\,\mathsf{X}\,a$ on the σ given above. The verdict is STILL_TRUE: the property is true in the finite semantics, since $\neg\,\mathsf{X}\,a$ holds on state s_1. When Definition 6.17 is used to transform the property to negation normal form, the resulting property is $\mathsf{true}\,\mathsf{U}\,\overline{\mathsf{X}}\,\neg a$. Algorithm 6.20 then correctly finds that the set of causes is empty.

If the NNF transformation of Algorithm 6.10 were used, which does not distinguish between X and $\overline{\mathsf{X}}$, the resulting transformed formula is $\mathsf{true}\,\mathsf{U}\,\mathsf{X}\,\neg a$. With this transformed formula Algorithm 6.20 would erroneously find $\langle s_1, a\rangle$ as a cause. This is because there is no state that has a successor state where $\neg a$ holds, so the algorithm determines that if a were true in s_1, the property $\mathsf{true}\,\mathsf{U}\,\mathsf{X}\,\neg a$ would have been true.

One might say that spurious causes for properties that are not FALSE or STILL_FALSE are not a serious problem, but if this property were part of a more complex property that is violated on a trace for a different reason, the spurious causes could be misleading.

Unfortunately, there are cases where Algorithm 6.20 seemingly under-approximates the set of causes compared to Algorithm 6.11, because of the combination of the fact that causes can only involve states in the trace, and that the algorithm returns on the first failure. For example, consider the property $\mathsf{G}((\mathsf{X}\,\mathsf{X}\,b) \wedge a)$ on the sequence $\sigma = s_0, s_1$ labeled $\{a\}, \{b\}$. The verdict is FALSE. Algorithm 6.11 (for the weak semantics) finds the cause $\langle s_1, a\rangle$. The subformula $\mathsf{X}\,\mathsf{X}\,b$ depends on a non-existent state s_2 beyond the end of the trace when evaluated on both s_0 and s_1, so in the weak semantics, it is satisfied in both states. a is violated in s_1, so when the algorithm evaluates s_1, the cause $\langle s_1, a\rangle$ is generated.

When Algorithm 6.20 (for the finite semantics) is used, however, no causes are generated. In the finite semantics, $\mathsf{X}\,\mathsf{X}\,b$ is not satisfied in either s_0 or s_1, so it causes the formula to be violated in state s_0. However, there is no state s_2 to be a cause so no causes are generated at this point. Because the algorithm returns on the first failure of the formula, the formula $(\mathsf{X}\,\mathsf{X}\,b) \land a$ is not evaluated on state s_1, so no further causes are generated.

Although it is unfortunate that in cases such as this, a cause in the weak semantics is no longer found in the finite semantics, it is strictly speaking not incorrect, because the violation of a in s_1 is not the first failure of the formula. This also means that for Algorithm 6.20, Proposition 6.13 no longer holds in cases where a *next* operator causes the formula to be violated in the finite semantics and the formula is also violated in the weak semantics by a different cause. In such a case, the causes that are generated are less helpful, but we expect this to be quite rare.

Possible solutions for this are discussed in Section 7.3.

6.2.3 Causes for MTL formulas

The definition of causes for LTL formulas of Definitions 6.5 and 6.16 are defined for LTL formulas, but can be applied to MTL formulas simply by using the (weak or finite) semantics of MTL.

By using state-value pairs for causes for the failure of an MTL formula, a cause is interpreted as a value in a state that when flipped (alongside a set of other values and states) causes the formula to be true as before. For an MTL formula, the set of states that can cause the violation of a formula depends on the intervals of the temporal operators and the timestamps of the states. However, similar to how the existence, non-existence, and relative orderings of states are not considered as possible causes in the paths of Definition 6.5, the timings of events are not considered as possible causes in this extension. Still, it is worthwhile to extend the approximation algorithm to timed sequences and MTL properties, as this would still greatly expand the number of properties in TRACE4CPS for which we can generate causes.

To update the algorithm to compute causes for MTL properties, several modifications must be made.

The first concerns the NNF transformation of MTL formulas. The equivalences in Proposition 6.9 hold for both MTL and LTL formulas. However, MTL adds the *constrained until* operator U_I, which equivalence (2) from Proposition 6.9 does not easily extend to:

$$\neg(a \,\mathsf{U}_I\, b) \not\equiv (\neg b \,\mathsf{U}_I\, (\neg a \wedge \neg b)) \vee \mathsf{G}_I \neg b$$

To see why, consider the following timed sequence, property and a hypothetical NNF transformation nnf' if the equivalence above were assumed to hold:

$$\rho = (\{b\}, 0), (\varnothing, 2), (\{b\}, 3)$$
$$\varphi = \neg(a \,\mathsf{U}_{[1,4]}\, b)$$
$$\varphi' = nnf'(\varphi) = (\neg b \,\mathsf{U}_{[1,4]}\, (\neg a \wedge \neg b)) \vee \mathsf{G}_{[1,4]} \neg b$$

The formula φ is true on ρ, but φ' is false. The second part of φ' ($\mathsf{G}_{[1,4]} \neg b$) is false as expected, because b does occur at time 3. The first part of φ' ($\neg b \,\mathsf{U}_{[1,4]}\, (\neg a \wedge \neg b)$) is false, because b does occur before $(\neg a \wedge \neg b)$ occurs. The problem is that $a \,\mathsf{U}_I\, b$ specifies that b should occur within the interval I, but a should hold on all states before b holds, even those before the interval I.

Several other ways of rewriting a negated timed until-formula into an NNF formula using U_I and G_I were tried, but each turned out not to result in an equivalent formula.

To remedy this, we use an alternative negation normal form by introducing the *release* operator R_I as an element of the core syntax of MTL, which (as you may recall from Section 2.1.1) is usually defined in terms of the *until* operator as its dual [Ouaknine and Worrell 2008]:

$$\varphi_1 \,\mathsf{R}_I\, \varphi_2 \overset{\text{def}}{=} \neg(\neg\varphi_1 \,\mathsf{U}_I\, \neg\varphi_2)$$

The timed release operator $\varphi_1 \,\mathsf{R}_I\, \varphi_2$ specifies that either φ_2 holds until the end of the interval I, φ_2 holds up to and including a state within the interval where φ_1 holds (so at that point both φ_1 and φ_2 hold), or there are no states in the interval I.

To allow R_I as a core element of the syntax of MTL, we must define its semantics explicitly, rather than in terms of another operator. It is given below for the finite semantics, formulated in two equivalent ways: one that is slightly more intuitive, from [Ouaknine and Worrell 2006], and the other such that it mirrors the semantics of U_I more closely to clearly see the duality. The semantics of U_I is repeated for easy comparison.

Definition 6.21 (Finite semantics of U and R in MTL) [Ouaknine and Worrell 2006]

Let $\rho = (s_0, t_0), \dots, (s_n, t_n)$ be a finite timed sequence, φ_1 and φ_2 formulas in MTL, and $I \subseteq [0, \infty)$ an interval.

$$
\begin{aligned}
\sigma^i \models_f \varphi_1 \,\mathsf{R}_I\, \varphi_2 \quad &\text{iff} \quad \forall_{i \leq j \leq n,\, t_j - t_i \in I}\, [\sigma^j \models \varphi_2 \text{ or } \exists_{i \leq k < j}\, \sigma^k \models \varphi_1] \\
&\Leftrightarrow \quad \forall_{i \leq j \leq n} [\sigma^j \models \varphi_2 \ \text{ or } \ t_j - t_i \notin I \ \text{ or } \ \exists_{i \leq k < j}\, \sigma^k \models \varphi_1] \\
\sigma^i \models_f \varphi_1 \,\mathsf{U}_I\, \varphi_2 \quad &\text{iff} \quad \exists_{i \leq j \leq n} [\sigma^j \models \varphi_2 \text{ and } t_j - t_i \in I \text{ and } \forall_{i \leq k < j}\, \sigma^k \models \varphi_1]
\end{aligned}
$$

Now, we can redefine negation normal form as follows:

Definition 6.22 (Negation normal form for MTL with X)
An MTL formula φ is said to be in *negation normal form* (NNF) if it only uses the Boolean operators $\vee$, $\wedge$, the temporal operators U_I, R_I, X, and $\overline{\mathsf{X}}$, and negations are only applied to atomic propositions.

This allows us to rewrite an MTL formula into negation normal form using the following equivalences

$$
\begin{aligned}
\neg(a \,\mathsf{U}_I\, b) &\equiv \neg a \,\mathsf{R}_I\, \neg b \\
\neg(a \,\mathsf{R}_I\, b) &\equiv \neg a \,\mathsf{U}_I\, \neg b
\end{aligned} \tag{6}
$$

Rules (2), (4), and (5) in Algorithm 6.10 were replaced by transformation rule (6):

Algorithm 6.23 (NNF transformation for MTL with U and R)

$$
nnf(\neg(\varphi_1 \,\mathsf{U}_I\, \varphi_2)) = nnf(\neg\varphi_1) \,\mathsf{R}_I\, nnf(\neg\varphi_2) \tag{6}
$$

The rest is the same as in Algorithm 6.18.

A rule for $nnf(\neg(\varphi_1 \,\mathsf{R}_I\, \varphi_2))$ is not necessary since formulas containing R are only created by the NNF transformation algorithm, and are already in NNF.

To verify that the introduction of R and the removal of G does not impact the results of the cause computation, an untimed version of the modified NNF transformation was applied to LTL formulas, and the C algorithm for LTL formulas was adapted by removing the case for G and adding the following case for untimed R to Algorithm 6.12:

Algorithm 6.24 (Approximate causes for LTL$_f$ with U and R)

$$
C(\sigma^i, \varphi_1 \,\mathsf{R}\, \varphi_2) = \begin{cases}
C(\sigma^i, \varphi_2) \cup C(\sigma^i, \varphi_1) & \text{if } val(\sigma^i, \varphi_1) = 0 \text{ and } val(\sigma^i, \varphi_2) = 0 \\
C(\sigma^i, \varphi_2) & \text{if } val(\sigma^i, \varphi_1) = 1 \text{ and } val(\sigma^i, \varphi_2) = 0 \\
C(\sigma^i, \varphi_2) \cup C(\sigma^{i+1}, \varphi_1 \,\mathsf{R}\, \varphi_2) & \text{if } val(\sigma^i, \varphi_1) = 0 \text{ and } val(\sigma^i, \varphi_2) = 1 \\
& \text{and } i < n \text{ and } val(\sigma^{i+1}, \varphi_1 \,\mathsf{R}\, \varphi_2) = 0 \\
\varnothing & \text{otherwise}
\end{cases}
$$

It can be explained as follows. If a (sub)formula $\varphi = \varphi_1 \,\mathsf{R}\, \varphi_2$ is explored on σ^i, there was no earlier state in which it was already found to be true. It can be false for any the following three reasons:

- φ_1 is false and φ_2 is false, in which case we add the causes of both φ_1 and φ_2 being false to the causes for the failure of φ.
- Only φ_2 is false, which violates φ because φ_2 is required to hold *up until and including* the state where φ_1 is true, so we add the causes of φ_2 being false to the causes for the failure φ.
- Only φ_1 is false, and φ is determined to be false at a later state. In this case, φ has not failed yet, but we know that it will, so we add the causes of φ_1 being false and the causes of the later violation of φ to the causes for the failure of φ.

In the case where both φ_1 and φ_2 hold, and the case where only φ_2 holds but we are at the last state in the sequence, φ is not violated so no causes are generated.

This was implemented in the code by removing the case for G and adding the following case to Algorithm 6.11:

Algorithm 6.25 (Approximate causes for LTL$_f$ with U and R)

```
1:  if $\varphi = \varphi_1 \mathsf{R} \varphi_2$ then
2:      $v_1 \leftarrow \mathrm{C}(\sigma^i, \varphi_1, \mathit{false})$
3:      $v_2 \leftarrow \mathrm{C}(\sigma^i, \varphi_2, \mathit{false})$
4:      if $\neg v_1$ and $\neg v_2$ then
5:          if $\mathit{record\_causes}$ then
6:              $\mathrm{C}(\sigma^i, \varphi_2, \mathit{true})$
7:              $\mathrm{C}(\sigma^i, \varphi_1, \mathit{true})$
8:          $v \leftarrow \mathit{false}$
9:      else if $v_1$ and $\neg v_2$ then
10:         if $\mathit{record\_causes}$ then
11:             $\mathrm{C}(\sigma^i, \varphi_2, \mathit{true})$
12:         $v \leftarrow \mathit{false}$
13:     else if $\neg v_1$ and $v_2$ and $i < n$ and $\neg \mathrm{C}(\sigma^{i+1}, \varphi, \mathit{false})$ then
14:         if $\mathit{record\_causes}$ then
15:             $\mathrm{C}(\sigma^i, \varphi_1, \mathit{true})$
16:             $\mathrm{C}(\sigma^{i+1}, \varphi, \mathit{true})$
17:         $v \leftarrow \mathit{false}$
18:     else
19:         $v \leftarrow \mathit{true}$
```

Comparing the results of Algorithm 6.25 (with the NNF transformation using R of Algorithm 6.23) of to those of Algorithm 6.20 (with the NNF transformation using G of Algorithm 6.18) by examining the causal sets generated for the example LTL properties and sequences from [Beer et al. 2009] and [Beer et al. 2012], and by using property-based testing (see Section 7.4), it was determined that they are equivalent as far as we could tell. This offered confidence that this modification would not cause incorrect results.

⋆ ⋆ ⋆

To generate causes for MTL properties with constrained temporal operators (U_I and R_I), rather than extending C with cases for these operators, an alternative approach was taken.

To avoid duplicating functionality between the MTL checking algorithm in TRACE4CPS and the C algorithm, and to investigate whether a closer integration between the causality analysis and the other functionality of TRACE4CPS is possible, the causal set generation is added to the MTL checking algorithm that computes verdicts in the finite semantics.

First, the *release* operator is added to the internal syntax of MTL properties, and added to the MTL checking algorithm so that it follows Definition 6.21. Next, the *next* and *weak next* are added to both the syntax and to the checking algorithm to help in testing whether the LTL causality algorithm and the MTL causality algorithm are equivalent for LTL properties containing next operators later.

Finally, the cause generation is added to the algorithm, so that similar to Algorithms 6.11, 6.14 and 6.20, when the procedure for computing the verdict of a formula φ on a trace σ returns, *causes* contains the approximate causal set for the first failure of φ on σ (in the finite semantics).

The resulting algorithm is shown in Algorithm 6.26.

The procedures for *until* and *release* are split into an outer and inner procedure: the outer procedure calles the inner with *record_causes* = *false* to check whether it is violated, and if so, calls it again with *record_causes* = *true* to generate the causes. This corresponds to the behavior of Algorithms 6.11, 6.14 and 6.20 and is necessary because when COMPUTE_CAUSE is called for an *until* or *release* formula, it is not yet known if the formula is violated later in the trace (and thus whether causes should be generated).

⋆ ⋆ ⋆

To see how Algorithm 6.26 analyzes the causes of a property with real-time constraints, we look at some examples.

First, consider the property $\varphi = \mathsf{F}_{[0,2]}\, b$, which specifies that b should occur within two time units from the first event of the trace, and the following timed trace:

$$\sigma = s_0, s_1, s_2 = (\{a\}, 0), (\{a\}, 1), (\{a\}, 3)$$

The verdict is FALSE, because there is no state in which b holds in the interval $[0, 2]$ starting from s_0. (The trace is longer than the interval, so it is also violated in the weak semantics.) Algorithm 6.26 finds a single cause $\langle s_1, b\rangle$: if b were true in s_1, the formula would be TRUE. If the F in φ were unconstrained, $\langle s_2, b\rangle$ could also be considered a cause, but because s_2 lies outside the interval $[0, 2]$, flipping the value of b in s_2 would not satisfy φ so it is correctly considered not to be a cause.

When we consider properties with constrained temporal operators, there are more cases in which Proposition 6.13 does not hold. For example, consider a property $\varphi = \mathsf{F}_{[1,2]}\, b$ and the following timed trace:

$$\sigma = s_0, s_1 = (\{a\}, 0), (\{b\}, 3)$$

The verdict is FALSE, because while b holds in state s_1, it is not in the interval $[1, 2]$. (The trace is longer than the interval, so it is also violated in the weak semantics.) In this example, Algorithm 6.26 finds no causes. For σ to satisfy φ, there would have to be a state in the interval $[1, 2]$ that satisfies b, but because there is no state in the interval $[1, 2]$, there is no value that could be flipped so that φ is satisfied. The fact that Definitions 6.5 and 6.16 do not directly capture the notion that the timings of the states in σ are what causes the violation of φ is discussed further in Section 7.3.

Nevertheless, we expect that for many MTL properties and traces, the causes that are generated are still helpful in finding the actual reason for a violation.

Algorithm 6.26 (MTL checking algorithm with cause generation)
Let $\rho = (s_0, t_0), \ldots, (s_n, t_n)$ be a finite timed sequence of length $n+1$, $p \in AP$ an atomic proposition, $I \subseteq [0, \infty)$ an interval of the form $[\inf(I), \sup(I))$, φ an MTL formula, and i a position in the sequence such that $0 \leq i \leq n$.

MTL properties are assumed to have been transformed to negation normal form using Algorithm 6.23. Calling COMPUTE_CAUSE($\rho, \varphi, 0, \mathit{true}$) will compute $[\rho \models_f \varphi]$ and add the causes of the failure of φ on ρ to *causes*. The return value of each call to COMPUTE_CAUSE is memoized in a memoization table like in Algorithm 4.4, the details of which are elided for clarity of presentation.

```
1:  $causes \leftarrow \varnothing$
2:  procedure COMPUTE_CAUSE($\rho$, true, $i$, $record\_causes$)
3:      return $\top$
4:  procedure COMPUTE_CAUSE($\rho$, $\neg$true, $i$, $record\_causes$)
5:      return $\bot$
6:  procedure COMPUTE_CAUSE($\rho, p, i, record\_causes$)
7:      if $p \in L(s_i)$ then
8:          return $\top$
9:      else
10:         if $record\_causes$ then
11:             $causes \leftarrow causes \cup \langle s_i, p \rangle$
12:         return $\bot$
13: procedure COMPUTE_CAUSE($\rho, \neg p, i, record\_causes$)
14:     if $p \in L(s_i)$ then
15:         if $record\_causes$ then
16:             $causes \leftarrow causes \cup \langle s_i, p \rangle$
17:         return $\bot$
18:     else
19:         return $\top$
20: procedure COMPUTE_CAUSE($\rho, \varphi_1 \wedge \varphi_2, i, record\_causes$)
21:     $v_1 \leftarrow$ COMPUTE_CAUSE($\rho, \varphi_1, i, record\_causes$)
22:     $v_2 \leftarrow$ COMPUTE_CAUSE($\rho, \varphi_2, i, record\_causes$)
23:     return $v_1 \wedge v_2$
24: procedure COMPUTE_CAUSE($\rho, \varphi_1 \vee \varphi_2, i, record\_causes$)
25:     $v_1 \leftarrow$ COMPUTE_CAUSE($\rho, \varphi_1, i, false$)
26:     $v_2 \leftarrow$ COMPUTE_CAUSE($\rho, \varphi_2, i, false$)
27:     if $record\_causes$ and $v_1 = \bot$ and $v_2 = \bot$ then
28:         COMPUTE_CAUSE($\rho, \varphi_1, i, true$)
29:         COMPUTE_CAUSE($\rho, \varphi_2, i, true$)
30:     return $v_1 \vee v_2$
31: procedure COMPUTE_CAUSE($\rho$, X $\varphi, i, record\_causes$)
32:     if $i = n$ then
33:         return $\bot$
34:     else
35:         return COMPUTE_CAUSE($\rho, \varphi, i+1, record\_causes$)
```

```
procedure COMPUTE_CAUSE($\rho, \overline{\mathsf{X}}\,\varphi, i, \mathit{record\_causes}$)
    if $i = n$ then
        return $\top$
    else
        return COMPUTE_CAUSE($\rho, \varphi, i+1, \mathit{record\_causes}$)
procedure COMPUTE_CAUSE($\rho, \varphi_1 \mathsf{U}_I \varphi_2, i, \mathit{record\_causes}$)
    $v \leftarrow$ COMPUTE_CAUSE_INNER($\rho, \varphi_1 \mathsf{U}_I \varphi_2, i, \mathit{false}$)
    if $\mathit{record\_causes}$ and $v = \bot$ then
        COMPUTE_CAUSE_INNER($\rho, \varphi_1 \mathsf{U}_I \varphi_2, i, \mathit{true}$)
    return $v$
procedure COMPUTE_CAUSE_INNER($\rho, \varphi_1 \mathsf{U}_I \varphi_2, i, \mathit{record\_causes}$)
    $c_1 \leftarrow \mathit{false}, c_{11} \leftarrow \mathit{true}$
    for $j = i$ to $n$ do
        $r_2 \leftarrow$ COMPUTE_CAUSE($\rho, \varphi_2, j, \mathit{false}$)
        $c_1 \leftarrow c_1 \vee (r_2 = \top \wedge t_j - t_i \in I \wedge c_{11})$
        if $c_1$ then
            return $\top$
        if $\mathit{record\_causes}$ and $r_2 = \bot$ and $t_j - t_i \in I$ then
            COMPUTE_CAUSE($\rho, \varphi_2, j, \mathit{true}$)
        $r_1 \leftarrow$ COMPUTE_CAUSE($\rho, \varphi_1, j, \mathit{false}$)
        $c_{11} \leftarrow c_{11} \wedge (r_1 = \top)$
        if $\mathit{record\_causes}$ and $r_1 = \bot$ and $t_j - t_i < \sup(I)$ then
            COMPUTE_CAUSE($\rho, \varphi_1, j, \mathit{true}$)
        if $(\neg c_{11} \vee j = n \vee t_j - t_i \geq \sup(I)) \wedge \neg c_1$ then
            return $\bot$
procedure COMPUTE_CAUSE($\rho, \varphi_1 \mathsf{R}_I \varphi_2, i, \mathit{record\_causes}$)
    $v \leftarrow$ COMPUTE_CAUSE_INNER($\rho, \varphi_1 \mathsf{R}_I \varphi_2, i, \mathit{false}$)
    if $\mathit{record\_causes}$ and $v = \bot$ then
        COMPUTE_CAUSE_INNER($\rho, \varphi_1 \mathsf{R}_I \varphi_2, i, \mathit{true}$)
    return $v$
procedure COMPUTE_CAUSE_INNER($\rho, \varphi_1 \mathsf{R}_I \varphi_2, i, \mathit{record\_causes}$)
    $c_1 \leftarrow \mathit{true}, c_{11} \leftarrow \mathit{false}$
    for $j = i$ to $n$ do
        $r_1 \leftarrow$ COMPUTE_CAUSE($\rho, \varphi_1, j, \mathit{false}$)
        $r_2 \leftarrow$ COMPUTE_CAUSE($\rho, \varphi_2, j, \mathit{false}$)
        if $\mathit{record\_causes}$ and $r_2 = \bot$ and $t_j - t_i \in I$ then
            COMPUTE_CAUSE($\rho, \varphi_2, j, \mathit{true}$)
            if $r_1 = \bot$ then
                COMPUTE_CAUSE($\rho, \varphi_1, j, \mathit{true}$)
        if $\mathit{record\_causes}$ and $j < n$ and $t_j - t_i \in I$ and $r_1 = \bot$ then
            COMPUTE_CAUSE($\rho, \varphi_1, j, \mathit{true}$)
        $c_1 \leftarrow c_1 \wedge (r_2 = \top \vee t_j - t_i \notin I \vee c_{11})$
        if $\neg c_1$ then
            return $\bot$
        $c_{11} \leftarrow c_{11} \vee (r_1 = \top)$
        if $(c_{11} \vee j = n \vee t_j - t_i \geq \sup(I)) \wedge c_1$ then
            return $\top$
```

6.3 Visual cause explanations

The approximate causal sets generated by Algorithm 6.26 are used to create a new visual explanation, to add to those described in Chapter 5.

Recall that the definition of a cause of a property violation in Definition 6.16 can be informally described (in the context of TRACE4CPS) as follows. A cause is the value of an atomic proposition in an event (i.e. whether an event passes an attribute filter defined in the property, see Section 3.3), such that if the validity of the atomic proposition on the event were flipped (that is, it would pass the attribute filter if it currently does not, or not pass if it currently does), possibly together with a 'witness set' of additional events and atomic propositions, the property would become either TRUE or STILL_TRUE.

Because the approximate causal set does not explicitly construct the witness set, only the causes themselves can be visualized. Furthermore, because the approximate causal set only explains the first failure of the property on the trace, it may be that a cause later in the trace is not explained (which may have the consequence that the explanation is actually clearer, but less complete).

When the user requests a cause explanation of a property with a FALSE or STILL_FALSE verdict, a set of events is added to the trace visualization, each corresponding to the combination of an event and an attribute filter. Optionally, the attribute filters can be decomposed so that a cause refers to single attribute key-value pair. The explanation events are grouped into swimlanes according to the attribute filter. The claims that correspond to the causal events can optionally be highlighted in a blue color.

In Figure 6.1, the visualization of the cause for the violation of `latency_discrete(58)` (defined in Example 3.5) has been added to the trace. It explains the cause of the violation of the requirement that a job takes at most 40 milliseconds as follows:

- The green arrow in the lower swimlane indicates that the event corresponding to the start of job 58 at time 445.43, which satisfies *start* $\{name \mapsto A, id \mapsto 58\}$, is a cause. Informally, if job 58 had not started, it could not have exceeded the latency requirement.
- The red arrows in the other swimlane indicate that all the events in the interval of 40 milliseconds, which do not satisfy *end* $\{name \mapsto G, id \mapsto 58\}$, are causes, because if any of them had been the end of job 58, it would have met the latency requirement.

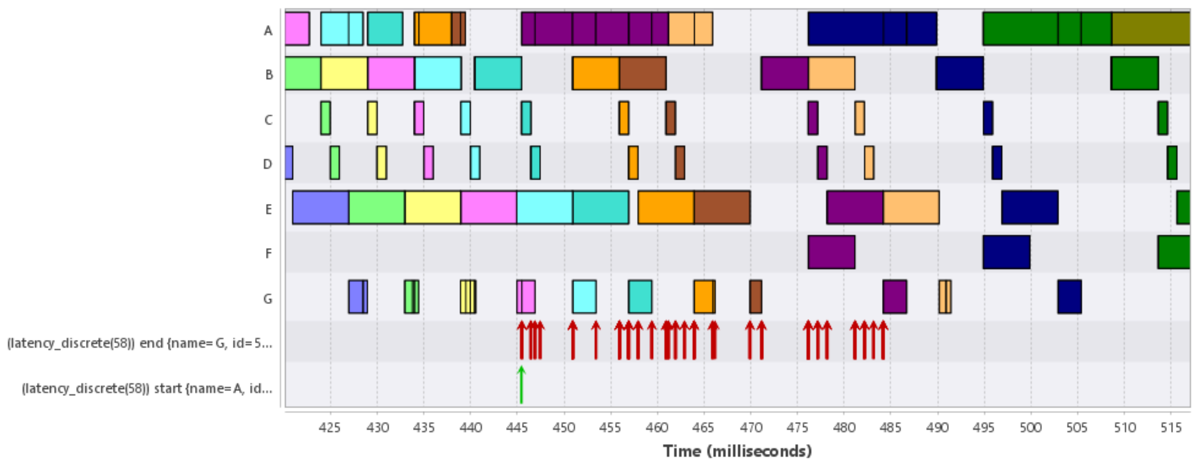


Figure 6.1: Cause explanation for `latency_discrete(58)`

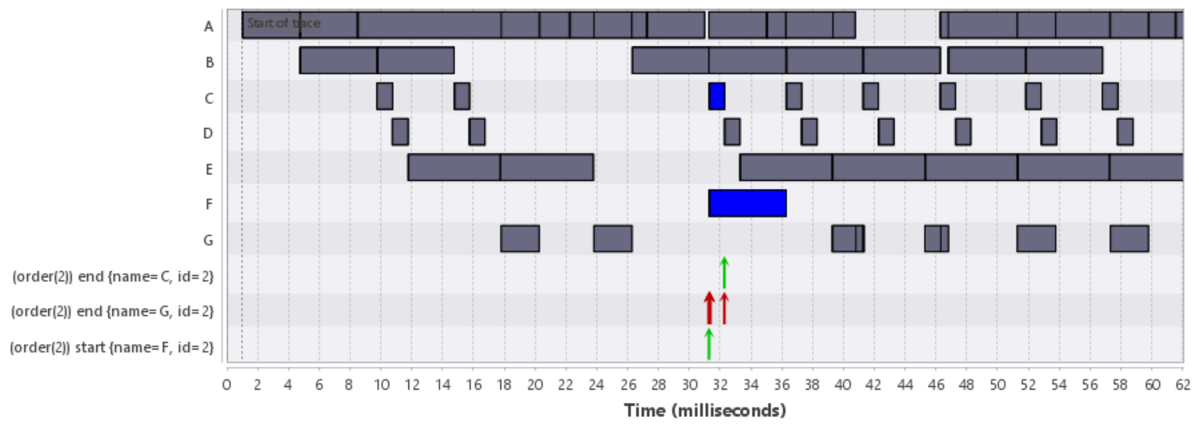


Figure 6.2: Cause explanation for `order(2)`

Note that the event at time 486.67, which corresponds to the end of task G of job 58, is not indicated as a cause. This is because there is no atomic proposition in the formula such that its validity on this event would impact the verdict, since it does not occur in the interval defined by the **`within`** operator.

Compared to the old property explanation in Figure 5.1, the cause explanation does not contain the spurious events leading up to the start of job 58 (between times 0 and 445.43) which do not contribute to the violation. Thus, it more directly and clearly indicates the point in the trace where the formula is violated.

It is quite similar to the property explanation in Figure 5.4, which was obtained through a process of trying different visualizations on different subformulas of the property. The clarity of the explanations are quite close, and choosing one over the other may come down to personal preference. However, the fact that the cause explanation was generated in a single click by the user makes the explanation much more accessible, since it requires less insight into which subformulas are likely to be informative.

The actual reason that the trace violates the specification – the fact that the end of job 58 occurs too late – is not captured by the notion of causality that we use here. This is discussed further in Section 7.3.

In Figure 6.2, the cause explanation of `order(2)` is shown, from the property `order` of Example 5.1, with highlighting of the causal claims. It indicates the start of task F and the end of task C as causes for the violation of the property.

Compared to the old property explanation in Figure 5.2 it is much more useful, since it indicates the actual events that are not ordered according to the specification, instead of simply pointing at the first event of the trace on which the *globally* operators of the sub-properties are evaluated.

Compared to the explanation shown in Figure 5.3, it is again more concise, as it does not show the superfluous events leading up to the start of task F, and is generated in a single click.

7 Discussion & future work

We have developed several new features for the presentation and explanation of runtime verification results in TRACE4CPS.

In Sections 7.1 to 7.3, the advantages and limitations of the new functionality are discussed, and possible directions for future research proposed.

In Section 7.4, the use of property-based testing to obtain a certain level of confidence in the correctness of the approaches is described.

7.1 Four-valued verdicts

In Chapter 4, the set of possible verdicts was expanded from three to four by computing a verdict according to a four-valued informative prefix semantics, implemented as an extension of the MTL checking algorithm of TRACE4CPS. The verdict is TRUE if the trace is an informative good prefix for the property, FALSE if the trace is an informative bad prefix. If the trace is not informative, the verdict is STILL_TRUE if it finitely satisfies the property, and STILL_FALSE if it does not.

In the case of the property `latency_discrete` from Example 3.5, in the three-valued semantics the verdict is NON-INFORMATIVE for every job that meets the maximum latency requirement. In the four-valued semantics, such a job is given the verdict STILL_TRUE, which is a more helpful result for the user. At the same time, the assumption that the trace might be extended in a way that may still satisfy or violate the property is represented by distinguishing STILL_TRUE from TRUE, which would not be possible if only the finite semantics were used to check this property.

Consider the property `reqack`, shown in Example 7.1, which specifies that every request event must eventually be followed by an acknowledgement event:

Example 7.1 (Request-acknowledgement property)

```
check reqack: globally
                if {'event'='req'} then
                  next finally {'event'='ack'}
```

When it is checked on any trace in the three-valued semantics, we always receive a NON-INFORMATIVE verdict, because any trace ending with an acknowledgement could be extended with an unacknowledged request, and any trace ending in an unacknowledged request could be extended with an acknowledgement. Using the four-valued semantics, a trace in which every request is acknowledged will result in a STILL_TRUE verdict, and a trace that contains an unacknowledged request will result in a STILL_FALSE verdict. Thus, even though we cannot be certain of the verdict in all possible extensions of the trace, the user receives useful information about the validity of the property on the trace so far.

Restricting ourselves to LTL formulas, the four-valued informative prefix semantics is less precise than RV-LTL (Section 2.1.5) in the sense that TRUE and FALSE are only returned for traces that are informative, rather than for all good and bad traces (traces that satisfy a property in all possible extensions, and traces that violate it in all possible extensions, see Definition 2.6). In the three-valued informative prefix semantics, where NON-INFORMATIVE verdicts already tend to be overabundant, this means that there are good and bad traces for which which we still receive a NON-INFORMATIVE verdict. (For example, the property $\mathsf{G}(p \land \neg p)$.) In the four-valued informative prefix semantics, a non-informative good trace will still result in a STILL_TRUE verdict rather than TRUE, and a non-informative bad trace in a STILL_FALSE verdict rather than FALSE, but we have found this level of imprecision to be less bothersome.

Nevertheless, if more precise verdicts are desired, it might be worth exploring whether it is possible to adapt the automaton-based monitoring approach for the three-valued semantics of LTL_3 and TLTL (timed LTL) in [Bauer et al. 2006] and for the four-valued semantics of RV-LTL in [Bauer et al. 2008] to (timed) MTL properties. In this case, the advantages of more precise verdicts would have to be weighed against the increase in computational complexity of this approach.

Alternatively, the approach for monitoring LTL specifications over finite traces of [Bartocci et al. 2018] could be considered, which is based on a counting semantics which produces one of five verdicts (true, presumably true, inconclusive, presumably false, false) representing a prediction of the satisfaction or violation of a formula given a prefix. Another option is the approach of [Morgenstern et al. 2012], which uses a set of semantics specialized to different classes of LTL formulas (safety, liveness, persistence and recurrence) and returns verdicts consisting of an n-tuple of 5-valued truth values (where n is dependent on the complexity of the formula), and has the property that it converges to the infinite path semantics of LTL. Both options apply to LTL properties, but it may be possible for them to be adapted to (timed) MTL properties.

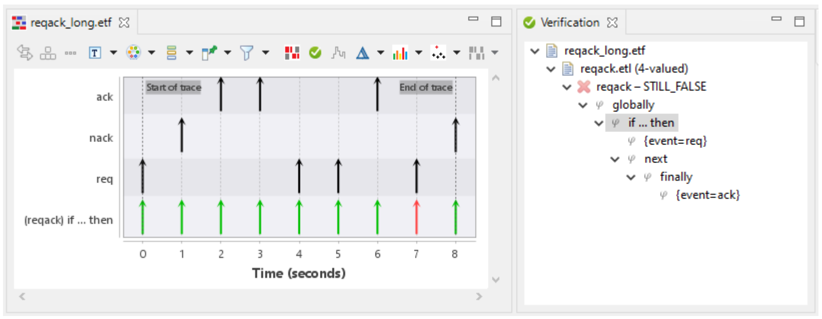


Figure 7.1: Visualization of a trace and the property `reqack`, showing the property tree, STILL_FALSE verdict and an explanation of the "`if ... then`" node

7.2 Interactive visualization of verdicts

In Chapter 5, the visual property explanation features of TRACE4CPS were extended with a number of new options. The addition of explanations of arbitrary subformulas, and of the visualization of temporal intervals that were evaluated while checking a timed property, introduces the possibility of an interactive process of exploring the verdict. While trying out different explanations of subformulas, the user is free to investigate the validity of subformulas on the events that led to the verdict, in the order that fits their current level of understanding.

Figure 7.1 shows a screenshot of the Verification panel and a property explanation of the property **`reqack`** of Example 7.1 after the changes made in Chapters 4 and 5. The verdict is STILL_FALSE instead of NON-INFORMATIVE, indicating that there is an unacknowledged request. A representation of the abstract syntax three of the formula is shown on the right, and each subformula can be double-clicked to generate an explanation of that subformula. The property explanation for the node "**`if ... then`**" is shown in the trace on the left, which clearly shows which request is left unacknowledged.

A similar visualization was possible in the unmodified version of TRACE4CPS, but the user would have had to factor out the "**`if ... then`**" subformula into a separate named definition. Especially for more complicated formulas, the ability to explore any subformula of the property without changing its ETL expression is more convenient, and allows to property to be expressed in a way that is most natural.

The tree representation of the formula, combined with the fact that multiple visualizations can be chosen for any node in this tree depending on the ETL operator it represents, forms a framework that supports a wealth of potential visualizations that can be added as part of the interactive explanation process. Due to time constraints and the promise of a more universal explanation mechanism in the form of the cause explanations of Chapter 6 (discussed in Section 7.3), the development of further visualizations is left for future work.

As an example, visualizations tailored to specific temporal operators may offer advantages, such as one for violations of (sub)formulas of the form $\mathsf{G}_I\,\varphi$ that shows a continuous indication of the validity of φ: green for the part of I in which φ was not yet violated, and red from that point it was violated. A similar visualization for the satisfaction of a formula $\mathsf{F}_I\,\varphi$ would be the reverse: red before φ was satisfied and green from that point on.

Visualizations tailored to specific combinations of temporal operators may offer further possibilities: for example, the violation of a (sub)formula of the form $\mathsf{G}_I(\varphi_1 \rightarrow \varphi_2)$ may be made clearer by ignoring those events in the interval I in which φ_1 is false, since the satisfaction of an implication in which the left side is false can be seen as trivial compared to an implication in which both the left and right side are true.

The development of a larger set of visualizations can be supported by user research in order to identify the (combinations of) visualizations that are most helpful in a set of example traces and properties, and possibly guide the development of a mechanism for automatically selecting the visualizations that are most likely to be helpful.

7.3 Cause explanations

In Chapter 6, the method of [Beer et al. 2012], extended to timed properties in the finite semantics, was used to develop a visual explanation of the cause of a property violation that is often clearer, more concise, and more accessible than the visualizations discussed in Section 7.2.

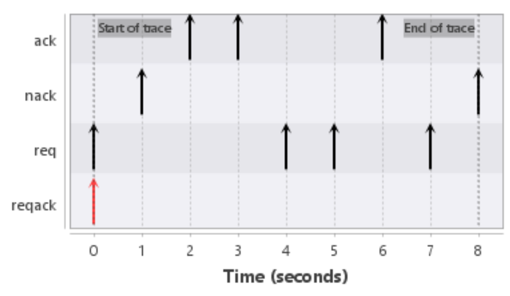
Figure 7.2a shows the explanation generated for the property `reqack` of Example 7.1 by the unmodified version of TRACE4CPS, and Figure 7.2b shows the cause explanation based on the approximate cause algorithm. Two causes are indicated: the unacknowledged "*req*" and an event that follows the request that is not an "*ack*". Compared to the old explanation, the cause explanation indicates the correct time at which the failure occurs (rather than the timestamp of the event on which the top-level *globally* formula was evaluated), and multiple possible causes are offered: if either the request at $t = 6$ had not happened, or an acknowledgement at $t = 7$ had happened, the property would have been satisfied.

7.3.1 Approximation

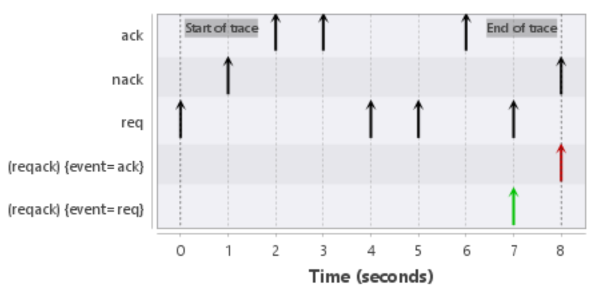
The formal, HP causality-based definition of a cause (Definition 6.5) on which the cause explanations are based can be informally described as follows. An event, in combination with an atomic proposition (i.e. an attribute filter) defined in the property, is a cause if changing the validity of the atomic proposition on that event (possibly together with another set of events and attributes) would satisfy the property.

The causes generated by the algorithm are an approximation in two senses. First, the algorithm only finds causes for the first failure of a formula on a trace. This means that "fixing" the trace (or system) using the causes that were given may not be enough to satisfy the formula. This means that the generated causes are not complete, but may in practice be clearer, since they are focused on a single failure.

However, it also means that when there are multiple failures of the formula in a trace, the first failure could potentially mask a more relevant failure later on. More specifically, in the finite semantics, the violation of a property in both the weak semantics and the finite semantics (resulting in a FALSE verdict) cannot be distinguished from a violation only in the finite semantics (resulting in a STILL_FALSE verdict). This could result in only a cause for the STILL_FALSE verdict being generated even when later in the trace, a cause for the FALSE verdict is present which might be more relevant to the user.



(a) Old explanation



(b) New cause explanation

Figure 7.2: Comparison between the default explanation of TRACE4CPS and the newly developed cause explanation on the `reqack` property

It is not clear whether this poses a problem in practical scenarios, but it might be useful to extend the algorithm so that causes for all failures of the formula can optionally be generated. Alternatively, the algorithm could be modified to operate in the four-valued semantics so that, while generating the cause for a FALSE verdict, if a violation leading to a STILL_FALSE verdict is encountered, it keeps looking for a violation leading to the FALSE verdict.

At the same time, the algorithm is an overapproximation of the set of causes for the first failure of a formula. In return, the algorithm has a much lower computational complexity than one that would compute causes according to the formal definition.

In Chapter 6, examples are given that display both the over- and underapproximation of the algorithm. It seems that in practice, the overapproximation is small, and the underapproximation is not necessarily a disadvantage, but this belief has not been tested on a large set of practical examples. To fully evaluate the cause explanations and the degree to which they are an approximation, it may again be insightful to perform user research with a large set of example properties and traces.

7.3.2 Causes for positive verdicts

Cause explanations can be generated for FALSE and STILL_FALSE verdicts. We expect that most instances in which explanations for verdicts are needed are for violations of a specification, because a violation indicates a potential bug in the system that generated the trace, in which case a clear understanding of the violation is helpful in locating and fixing the bug.

However, an explanation of the cause of the satisfaction of a property (i.e. where the verdict is TRUE or STILL_TRUE) can also be useful. For example, it may be that a formula is trivially satisfied because it does not correctly express the desired specification. A cause explanation may help the user gain confidence in the verdict if it confirms the user's intuition about the specified behavior.

Simply generating a cause explanation of the violation of the negation of the formula as an explanation of the satisfaction of the original formula may be useful as a basis for this functionality, but initial forays into this approach showed that it does not always generate useful results. The notions of *vacuity* [Beer et al. 1997] and *coverage* [Chockler et al. 2008] may be helpful in identifying formulas that are satisfied for unintended reasons.

7.3.3 Alternative notions of causality

The definition of causes used in our approach means that causes must refer to an event that occurs in the trace, and the values of the attributes of that event. The algorithm generates causes for most violated properties, but a property that is violated because a *next* operator is evaluted on the last event of the trace (so that the next event does not exist) may not have a cause that occurs in the trace. To indicate a cause in this situation requires either a notion of causality that can refer to required events beyond the end of the trace, or perhaps simply some indication that the cause is a requirement on an event beyond the end of the trace.

Similarly, the causes generated for timed properties, when the real reason for the violation is due to the timing of events, can sometimes be indirect or in some cases non-existent. This is because the definition of causes used for timed properties is a straightforward extension of the definition used by [Beer et al. 2012] to MTL formulas that, while it takes into account the real-time aspects of the semantics of MTL while evaluating the formula, does not directly capture the timing of events as possible causes, or consider events that occur outside the intervals of temporal operators.

In Section 6.2.3 we showed that the set of causes for $(\{a\}, 0), (\{b\}, 3) \not\models \mathsf{F}_{[1,2]}\, b$ is empty, because there are no states in the interval $[1, 2]$ whose values are directly responsible for the violation of the property.

In order to formulate the cause of the violation in cases such as the above, and to formulate more direct causes for formulas that are violated due to the timings of events, a different notion of causality might be considered that in its counterfactual criteria also allows the possibility of inserting a new state into the trace or changing the timings of the states (in addition to changing the values of attributes). In the example above, the timestamp (3) of the state $(\{b\}, 3)$ could then be considered a cause. However, the possibilities for changing the trace so that the property is satisfied would become infinite, so a finite representation of the infinite set of counterfactual possibilities must be used in order for the approach to be practical.

Alternatively, we could consider the interval $[1, 2]$ to have caused the failure of the property in the example above, since if the interval were larger, the state $(\{b\}, 3)$ could have satisfied the formula. This would shift the focus of causality from considering causes to be events and their attribute values in the trace to also considering the formula itself to be responsible for its violation.

More generally, the notion of causality used in this thesis captures the attribute values of states that exist in the trace, but cannot capture the non-occurrence of events, their relative ordering, or (directly) the timing of events. In the approach to causality checking in [Leitner-Fischer and Leue 2013; Beer et al. 2015], in which causes are found in programs modeled as transition systems, and which uses techniques similar to model checking, a more sophisticated notion of causality is used so that causes are expressed as formulas in a logic called event order logic which does directly capture the presence, non-occurrence and relative ordering of events. However, it is not clear if this approach can be applied to single traces.

7.3.4 Integration of explanation features

The use of negation normal form in the approximate cause algorithm leads to a simpler algorithm, but also means that in order to find the cause of the violation of an MTL formula which has been checked using the regular checking algorithm described in Chapter 4, it must first be transformed and then re-checked using the algorithm extended with cause generation. As a consequence, the formula that is used to generate the cause explanations does not map directly to the formula as it is shown in the Verification panel and used to generate the regular property explanations.

It may be beneficial for further integration of the causality analysis with the property checking and explanation features to develop an algorithm that does not require the transformation to negation normal form. To do this, some concepts that are presented in [Beer et al. 2009], which is an earlier version of the paper [Beer et al. 2012] that was used in Chapter 6 might be useful: the *polarity* of a subformula (which is positive if the subformula appears under an even number of negations and negative otherwise) and the notion of being *bottom-valued* (which is equivalent to false for a subformula with positive polarity and true for a subformula with negative polarity).

7.4 Verification using property-based testing

During the research, property-based testing was used to verify (but not prove) the correctness of the approaches developed in this thesis. Property-based testing is a software testing method in which a large number of tests is automatically generated. Test cases are randomly drawn from a distribution over the domain of inputs of the system under test, and the output of the system is checked using an automatically checkable specification [Claessen and Hughes 2000].

The Java package "junit-quickcheck" [Holser 2020] was used to define generators for MTL formulas and execution traces, and to generate the test cases in which several assertions were checked to verify that the developed algorithms fulfill several desired properties.

MTL formulas were generated according to the following grammar, where $r \in \mathbb{R}$ is a random number in the range $[0, 6]$:

$$\begin{aligned}
\varphi &\coloneqq \mathsf{true} \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \rightarrow \varphi_2 \mid \varphi_1 \,\mathsf{U}_I\, \varphi_2 \mid \mathsf{F}\,\varphi \mid \mathsf{G}\,\varphi \mid \mathsf{X}\,\varphi \\
p &\coloneqq \{e \mapsto 1\} \mid \{e \mapsto 2\} \mid \{e \mapsto 3\} \\
I &\coloneqq [i, i+j] \mid [i, i+j) \mid (i, i+j] \mid (i, i+j) \mid [0, \infty) \\
i, j &\coloneqq 0.0 \mid 0.5 \mid 1.0 \mid 1.5 \mid 2.0 \mid 2.5 \mid 5.0 \mid 6.0 \mid r
\end{aligned}$$

The reason that the numbers i, j used in the endpoints of intervals are chosen from both a set of constants and from the complete range of (double-precision floating point) real numbers is to increase the chance that the cases where the endpoints of an interval are exactly the timestamps of events in the trace are tested.

Formulas of the form $\varphi_1 \,\mathsf{R}_I\, \varphi_2$ are not generated directly (and there is no construct in ETL that directly maps to it), but they are indirectly created using the negation normal form transformation that uses R.

LTL formulas were generated by restricting the previous grammar so that all intervals are $[0, \infty)$.

The length of the generated formulas, defined as the number of nodes in the tree structure that describes the formula, could be controlled, and ranged from 3 to 20 in the executed tests.

Traces were generated according to the following grammar, with n ranging from 0 to 9:

$$\begin{aligned}
\rho &\coloneqq s_0, \ldots, s_n \\
s_i &\coloneqq (\{e \mapsto 1\}, i) \mid (\{e \mapsto 2\}, i)
\end{aligned}$$

Each test was executed using 2,000,000 generated test cases with the properties specified later in this section. Although a passed test does not constitute a proof that the system under test fulfills the specified property, it does give a relatively high level of confidence, under the assumption that the generated test cases are a representative sample for the entire domain of MTL properties and traces that TRACE4CPS supports.

The property-based testing approach resulted in several bugs being caught while formulating the algorithms described in this thesis, and the failing test cases turned out to be very useful in developing a better understanding of the problems the algorithms were meant to solve.

7.4.1 Four-valued semantics

The correctness of the algorithms developed in Section 4.3 were verified by testing the following property on every generated MTL formula φ and trace ρ:

Proposition 7.2 (Correctness of four-valued informative prefix algorithm)

$$\begin{aligned}
[\rho \models_{\mathrm{IP3}} \varphi] = \top & \implies [\rho \models_{\mathrm{IP4}} \varphi] = \top, \\
[\rho \models_{\mathrm{IP3}} \varphi] = \bot & \implies [\rho \models_{\mathrm{IP4}} \varphi] = \bot, \\
[\rho \models_{\mathrm{IP3}} \varphi] = \ ? \ \wedge \ \rho \models_f \varphi & \implies [\rho \models_{\mathrm{IP4}} \varphi] = \top^?, \\
[\rho \models_{\mathrm{IP3}} \varphi] = \ ? \ \wedge \ \rho \not\models_f \varphi & \implies [\rho \models_{\mathrm{IP4}} \varphi] = \bot^?
\end{aligned}$$

$[\rho \models_{\mathrm{IP3}} \varphi]$ was computed using Algorithm 4.4, which was already present in TRACE4CPS, and $[\rho \models_{\mathrm{IP4}} \varphi]$ was computed both using both Algorithm 4.5 and Algorithm 4.7, which both passed this test.

Because of the correctness proof of Algorithm 4.4 given in [Hendriks et al. 2016b], it can be assumed to conform to Definition 4.1. This means that Proposition 7.2 implies that the two tested algorithms conform to Definition 4.2.

7.4.2 Causality

Although no automatically checkable correctness criterion for the approximated cause algorithms of Section 6.2 could be formulated, some desired properties of the algorithms could still be verified.

Let $C_{\mathrm{original}}(\sigma, \varphi)$ be the set of causes returned by Algorithm 6.11 (which corresponds to the C algorithm of [Beer et al. 2012]), $C_{\mathrm{modified}}(\sigma, \varphi)$ those by Algorithm 6.14 (which uses a different expression to recurse into the next states for the temporal operators), $C_{\mathrm{finite}}(\sigma, \varphi)$ those by Algorithm 6.20 (which finds causes in the finite semantics instead of the weak semantics), C_{R} those by Algorithm 6.25 (which uses R instead of G in the negation normal form), and C_{MTL} those by Algorithm 6.26 (which finds causes for (timed) MTL formulas).

Let φ be an LTL property generated as describe above, and let σ be a generated trace ρ in which the timestamps of events are ignored. Let nnf^-_{G} be the negation normal form transformation of Algorithm 6.10, nnf^f_{G} that of Algorithm 6.18 and nnf^f_{R} that of Algorithm 6.23.

The fact that the modifications made in Algorithm 6.14 do not impact the results of the algorithm was verified using the following:

Proposition 7.3 (Correctness of modification of C)

$$C_{\mathrm{original}}(\sigma, nnf^-_{\mathsf{G}}(\varphi)) = C_{\mathrm{modified}}(\sigma, nnf^-_{\mathsf{G}}(\varphi))$$

Proposition 6.13 was checked using the following:

Proposition 7.4 (Validity of Proposition 6.13)

$$\sigma \not\models^- \varphi \implies |C_{\mathrm{original}}(\sigma, nnf^-_{\mathsf{G}}(\varphi))| = |C_{\mathrm{modified}}(\sigma, nnf^-_{\mathsf{G}}(\varphi))| > 0$$

The same property for $\sigma \not\models_f \varphi$ and $C_{\text{finite}}(\sigma, \varphi)$ was determined not to hold by this method, which is explained in Section 6.2.2.

The validity of the negation normal form transformation using R instead of G for LTL formulas was verified using the following:

Proposition 7.5 (Correctness of $\boldsymbol{nnf}_{\mathsf{R}}$)

$$C_{\text{finite}}(nnf^f_{\mathsf{G}}(\sigma), \varphi) = C_{\mathsf{R}}(nnf^f_{\mathsf{R}}(\sigma), \varphi)$$

The correctness of Algorithm 6.26 for LTL formulas (which is a subset of all properties that it supports) was verified using the following:

Proposition 7.6 (Correctness of MTL cause algorithm (for LTL))

$$C_{\text{finite}}(\sigma, nnf^f_{\mathsf{G}}(\varphi)) = C_{\text{MTL}}(\sigma, nnf^f_{\mathsf{R}}(\varphi))$$

Finally, let ψ be an MTL formula generated as above. The correctness of the semantics of the *release* operator that was added to the MTL checking algorithm, and of the negation normal form transformation using R for MTL formulas, was tested using the following:

Proposition 7.7 (Correctness of R_I)

$$\rho \models_f \psi \iff \rho \models_f nnf^f_{\mathsf{R}}(\psi)$$

Unfortunately, the correctness of the extension of the approximated cause algorithm to the finite semantics and to timed properties could not easily be verified using property-based testing, but a number of test cases were checked manually to see that they are consistent with what was expected.

8 Conclusion

In Section 1.3, the aim of the research presented in this thesis was defined to be the exploration of several paths towards improving the presentation of the results of runtime verification in TRACE4CPS.

We posed the following main research question:

RQ0. How can the presentation of runtime verification results be improved to help the user understand a verdict on an execution trace in TRACE4CPS?

In order to answer this question, it was refined into three sub-questions.

RQ1. How can the user, upon receiving a NON-INFORMATIVE verdict, be given information about the validity of the property so far?

A four-valued informative prefix semantics was formulated as an extension of the three-valued informative prefix semantics for MTL on finite traces that splits the NON-INFORMATIVE verdict into two verdicts: STILL_FALSE and STILL_TRUE. In the case that a user checks a property on a trace where the satisfaction or violation of the property could not be definitively determined, they are now presented with a verdict that offers useful information regarding the validity of the property on the finite trace, without being overly optimistic or pessimistic about its validity on the unknown continuation of the trace.

The informative prefix algorithm with which MTL formulas are checked was extended to use the four-valued semantics, so that a four-valued verdict is computed in a single execution of the algorithm, and the property explanations based on the intermediate values produced by the algorithm use the correct values for subformulas of the property.

Using the observation that the four truth values form a lattice, the algorithm was expressed in a concise and elegant way in terms of partial order operations on this lattice, and can easily be implemented in any offline runtime verification application.

RQ2. How can all subformulas of an ETL property be explained without becoming overwhelming?

After the user checks a property on a trace, they are now presented with a tree representation of the formula in terms of the ETL syntax in which the property was specified. The user is then free to generate visual explanations of any subformula of the property (in addition to the existing explanation of only the top-level formula and any named definitions) that show the validity of the subformula for all events that it was checked on. Additionally, a type of explanation was introduced that visualizes the temporal intervals of timed operators in the formula as they were evaluated on the trace. This helps the user understand how the timing of events relates to the time constraints of the property.

These visualizations support an interactive process where the user gains understanding of the verdict through a series of explanations directed by the user's level of understanding. The user can pick one of multiple visualizations on any node in the tree representation of the ETL formula, which can serve as a basis for future development of new visualizations that can be combined into explanations tailored to specific (combinations of) ETL constructs.

RQ3. How can property explanations clearly explain the cause of the verdict?

An existing approach that uses an instance of HP causality to explain violations of a property on a trace was used to create cause explanations, a new type of property explanation that directly and concisely indicates possible causes of a FALSE or STILL_FALSE verdict. The algorithm was extended so that cause explanation can also be generated for timed properties (that can be expressed in MTL).

The cause explanations are a quick way to understand why a FALSE or STILL_FALSE was reached that is universal in that it is not dependent on the specific structure of the ETL formula. In cases where the explanation does not directly indicate the cause, such as those where the actual reason for the violation concerns the absence of events or where the timing of events is such that they occur outside the intervals specified in the property, they are often a helpful starting point for the investigation using the other property explanation features.

⋆ ⋆ ⋆

To answer the main research question, we note that the three areas in which contributions were made in this thesis work together to increase the understandability of verification results.

The four-valued verdicts help the user in determining to what extent the trace meets the specification, and the intermediate values produced by the extended verification algorithm (including the distinction between STILL_FALSE and STILL_TRUE) are used in the generation of the property explanations. The visual cause explanations provide a quick and precise way to focus the user's attention on the cause of a property violation. If necessary, this understanding can be refined interactively with the use of the other visual property explanations of any subformula and of the temporal intervals.

Ultimately, a deep understanding of the verdict can be used by the user to be more effective in fixing the defect in the system that has produced the failure, or to refine the specification as part of the verification process.

Our results, though presented as new functionality in TRACE4CPS, are intended to be useful for any application of runtime verification using temporal logic. Our hope is that improving the understandability of verdicts will help in lowering barriers that prevent more widespread use of formal verification to increase the safety and reliability of software systems.

Bibliography

Alur, Rajeev, Tomás Feder, and Thomas A. Henzinger (1996). "The benefits of relaxing punctuality". In: *Journal of the ACM* 43.1, pp. 116–146. DOI: 10.1145/227595.227602.

Baier, Christel, Clemens Dubslaff, Florian Funke, Simon Jantsch, Rupak Majumdar, Jakob Piribauer, and Robin Ziemek (2021). "From verification to causality-based explications". In: *Leibniz International Proceedings in Informatics, LIPIcs* 198, 1:1–1:20. DOI: 10.4230/LIPIcs.ICALP.2021.1.

Baier, Christel, Clemens Dubslaff, Florian Funke, Simon Jantsch, Jakob Piribauer, and Robin Ziemek (2022). "Operational Causality - Necessarily Sufficient and Sufficiently Necessary". In: *A Journey from Process Algebra via Timed Automata to Model Learning*. Vol. 13560 LNCS. Springer, pp. 27–45. DOI: 10.1007/978-3-031-15629-8_2.

Baier, Christel and Joost-Pieter Katoen (2008). *Principles Of Model Checking*. MIT Press. ISBN: 9780262026499. URL: https://mitpress.mit.edu/9780262026499/principles-of-model-checking/.

Ball, Thomas, Mayur Naik, and Sriram K. Rajamani (2003). "From symptom to cause: Localizing errors in counterexample traces". In: *ACM SIGPLAN Notices* 38.1, pp. 97–105. DOI: 10.1145/640128.604140.

Bartocci, Ezio, Roderick Bloem, Dejan Nickovic, and Franz Roeck (2018). "A Counting Semantics for Monitoring LTL Specifications over Finite Traces". In: *Computer Aided Verification. CAV 2018*. Vol. 10981 LNCS. Springer, pp. 547–564. DOI: 10.1007/978-3-319-96145-3_29.

Bauer, Andreas, Martin Leucker, and Christian Schallhart (2006). "Monitoring of Real-Time Properties". In: *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science*. Vol. 4337 LNCS. Springer, pp. 260–272. DOI: 10.1007/11944836_25.

Bauer, Andreas, Martin Leucker, and Christian Schallhart (2008). *The good, the bad, and the ugly-but how ugly is ugly?* Tech. rep. Institut für Informatik der Technische Universität München. URL: https://mediatum.ub.tum.de/1094622.

Beer, Adrian, Stephan Heidinger, Uwe Kühne, Florian Leitner-Fischer, and Stefan Leue (2015). "Symbolic causality checking using bounded model checking". In: *Model Checking Software. SPIN 2015*. Vol. 9232 LNCS. Springer, pp. 203–221. DOI: 10.1007/978-3-319-23404-5_14.

Beer, Ilan, Shoham Ben-David, Hana Chockler, Avigail Orni, and Richard Trefler (2009). "Explaining counterexamples using causality". In: *CAV 2009: Computer Aided Verification*. Vol. 5643 LNCS. Springer, pp. 94–108. DOI: 10.1007/978-3-642-02658-4_11.

Beer, Ilan, Shoham Ben-David, Hana Chockler, Avigail Orni, and Richard Trefler (2012). "Explaining counterexamples using causality". In: *Formal Methods in System Design* 40.1, pp. 20–40. DOI: 10.1007/s10703-011-0132-2.

Beer, Ilan, Shoham Ben-David, Cindy Eisner, and Avner Landver (1996). "RuleBase: An industry-oriented formal verification tool". In: *33rd Design Automation Conference Proceedings*, pp. 655–660. DOI: 10.1109/DAC.1996.545656.

Beer, Ilan, Shoham Ben-David, Cindy Eisner, and Yoav Rodeh (1997). "Efficient detection of vacuity in ACTL formulas". In: *Computer Aided Verification. CAV 1997*. Vol. 1254 LNCS. Springer, pp. 279–290. DOI: 10.1007/3-540-63166-6_28.

Caltais, Georgiana, Sophie Linnea Guetlein, and Stefan Leue (2019). “Causality for General LTL-definable Properties”. In: *Electronic Proceedings in Theoretical Computer Science* 286, pp. 1–15. DOI: 10.4204/EPTCS.286.1.

Chockler, Hana, Joseph Y. Halpern, and Orna Kupferman (2008). “What causes a system to satisfy a specification?” In: *ACM Transactions on Computational Logic* 9.3, pp. 1–26. DOI: 10.1145/1352582.1352588.

Claessen, Koen and John Hughes (2000). “QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs”. In: *ICFP ’00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming.* ACM, pp. 268–279. DOI: 10.1145/351240.351266.

Dou, Wei, Domenico Bianculli, and Lionel Briand (2018). “Model-Driven Trace Diagnostics for Pattern-based Temporal Specifications”. In: *21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018.* ACM, pp. 278–288. DOI: 10.1145/3239372.3239396.

Dubslaff, Clemens, Kallistos Weis, Christel Baier, and Sven Apel (2022). “Causality in Configurable Software Systems”. In: *ICSE ’22: Proceedings of the 44th International Conference on Software Engineering.* Vol. 2022-May. IEEE Computer Society, pp. 325–337. DOI: 10.1145/3510003.3510200.

Dwyer, Matthew B., George S. Avrunin, and James C. Corbett (1999). “Patterns in property specifications for finite-state verification”. In: *Proceedings of the 21st international conference on Software engineering.* ACM, pp. 411–420. DOI: 10.1145/302405.302672.

Eclipse TRACE4CPS™ (2022). URL: https://projects.eclipse.org/projects/technology.trace4cps (visited on 2023-01-25).

Eisner, Cindy, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout (2003). “Reasoning with temporal logic on truncated paths”. In: *Computer Aided Verification. CAV 2003.* Vol. 2725 LNCS. Springer, pp. 27–39. DOI: 10.1007/978-3-540-45069-6_3.

Eiter, Thomas and Thomas Lukasiewicz (2002). “Complexity results for structure-based causality”. In: *Artificial Intelligence* 142.1, pp. 53–89. DOI: 10.1016/S0004-3702(02)00271-0.

Furia, Carlo A. and Matteo Rossi (2006). “Integrating discrete- and continuous-time metric temporal logics through sampling”. In: *Formal Modeling and Analysis of Timed Systems. FORMATS 2006.* Vol. 4202 LNCS. Springer, pp. 215–229. DOI: 10.1007/11867340_16.

Groce, Alex (2004). “Error Explanation with Distance Metrics”. In: *Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2004.* Vol. 2988 LNCS. Springer, pp. 108–122. DOI: 10.1007/978-3-540-24730-2_8.

Hall, Ned (2004). “Two Concepts of Causation”. In: *Causation and Counterfactuals.* Ed. by John Collins, Ned Hall, and L.A. Paul. The MIT Press. Chap. 9, pp. 225–276. DOI: 10.7551/mitpress/1752.003.0010.

Halpern, Joseph Y. (2015). “A Modification of the Halpern-Pearl Definition of Causality”. In: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence*, pp. 3022–3033. URL: https://www.ijcai.org/Abstract/15/427.

Halpern, Joseph Y. and Judea Pearl (2001). “Causes and Explanations: A Structural-Model Approach: Part i: Causes”. In: *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence.* UAI’01, pp. 194–202. arXiv: 1301.2275.

Halpern, Joseph Y. and Judea Pearl (2005). “Causes and Explanations: A Structural-Model Approach. Part I: Causes”. In: *The British Journal for the Philosophy of Science* 56.4, pp. 843–887. DOI: 10.1093/bjps/axi147.

Harman, Mark and Robert Hierons (2001). “An overview of program slicing”. In: *Software Focus* 2.3, pp. 85–92. DOI: 10.1002/SWF.41.

Hendriks, Martijn, Marc Geilen, Amir R. B. Behrouzian, Twan Basten, Hadi Alizadeh, and Dip Goswami (2016a). “Checking Metric Temporal Logic with TRACE”. In: *2016 16th International Conference on Application of Concurrency to System Design (ACSD)*. IEEE, pp. 19–24. DOI: 10.1109/ACSD.2016.13.

Hendriks, Martijn, Marc Geilen, Amir R. B. Behrouzian, Twan Basten, Hadi Alizadeh, and Dip Goswami (2016b). *Checking Metric Temporal Logic with TRACE*. Tech. rep. Eindhoven University of Technology. URL: https://www.es.ele.tue.nl/esreports/esr-2016-01.pdf.

Hendriks, Martijn, Jacques Verriet, and Twan Basten (2023). “Visualization, Transformation and Analysis of Execution Traces with the Eclipse TRACE4CPS Trace Tool”. (Working paper).

Ho, Hsi-Ming, Joël Ouaknine, and James Worrell (2014). “Online Monitoring of Metric Temporal Logic”. In: *Runtime Verification. RV 2014*. Vol. 8734 LNCS. Springer, pp. 178–192. DOI: 10.1007/978-3-319-11164-3_15.

Holser, Paul (2020). *junit-quickcheck*. URL: https://pholser.github.io/junit-quickcheck/site/1.0/.

Horak, Tom, Norine Coenen, Niklas Metzger, Christopher Hahn, Tamara Flemisch, Julian Mendez, Dennis Dimov, Bernd Finkbeiner, and Raimund Dachselt (2022). “Visual Analysis of Hyperproperties for Understanding Model Checking Results”. In: *IEEE Transactions on Visualization and Computer Graphics* 28.1, pp. 357–367. DOI: 10.1109/TVCG.2021.3114866.

Horak, Tom, Norine Coenen, Niklas Metzger, Christopher Hahn, Tamara Flemischm, Julián Méndez, Dennis Dimov, Bernd Finkbeiner, and Raimund Dachselt (2023). *HyperVis*. URL: https://imld.de/en/research/research-projects/hypervis/ (visited on 2023-02-09).

Hume, David (1739). *A Treatise of Human Nature: Being an Attempt to Introduce the Experimental Method of Reasoning into Moral Subjects*. Sterling Publishing. ISBN: 9780760771723.

Kaleeswaran, Arut Prakash, Arne Nordmann, Thomas Vogel, and Lars Grunske (2022). “A systematic literature review on counterexample explanation”. In: *Information and Software Technology* 145, p. 106800. DOI: 10.1016/j.infsof.2021.106800.

Koymans, Ron (1990). “Specifying real-time properties with metric temporal logic”. In: *Real-Time Systems* 2.4, pp. 255–299. DOI: 10.1007/BF01995674.

Kupferman, Orna and Moshe Y. Vardi (2001). “Model Checking of Safety Properties”. In: *Formal Methods in System Design* 19.3, pp. 291–314. DOI: 10.1023/A:1011254632723.

Leitner-Fischer, Florian and Stefan Leue (2013). “Causality Checking for Complex System Models”. In: *VMCAI 2013: Verification, Model Checking, and Abstract Interpretation*. Vol. 7737 LNCS. Springer, pp. 248–267. DOI: 10.1007/978-3-642-35873-9_16.

Leitner-Fischer, Florian and Stefan Leue (2014). “SpinCause: A tool for causality checking”. In: *SPIN 2014: Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*. SPIN 2014. ACM, pp. 117–120. DOI: 10.1145/2632362.2632371.

Leucker, Martin and Christian Schallhart (2009). “A brief account of runtime verification”. In: *Journal of Logic and Algebraic Programming* 78.5, pp. 293–303. DOI: 10.1016/j.jlap.2008.08.004.

Lewis, David (1973). “Causation”. In: *The Journal of Philosophy* 70.17, p. 556. DOI: 10.2307/2025310.

Maler, Oded and Dejan Nickovic (2004). “Monitoring Temporal Properties of Continuous Signals”. In: *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, FORMATS/FTRTFT 2004*. Vol. 3253 LNCS. Springer, pp. 152–166. DOI: 10.1007/978-3-540-30206-3_12.

Manna, Zohar and Amir Pnueli (1992). *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer. DOI: 10.1007/978-1-4612-0931-7.

Markey, N. and P. Schnoebelen (2003). “Model Checking a Path”. In: *International Conference on Concurrency Theory*. Vol. 2761. Springer, pp. 251–265. DOI: 10.1007/978-3-540-45187-7_17.

Morgenstern, Andreas, Manuel Gesell, and Klaus Schneider (2012). “An Asymptotically Correct Finite Path Semantics for LTL”. In: *Logic for Programming, Artificial Intelligence, and Reasoning. LPAR 2012*. Vol. 7180 LNCS. Springer, pp. 304–319. DOI: 10.1007/978-3-642-28717-6_24.

Ouaknine, Joël and James Worrell (2006). “Safety metric temporal logic is fully decidable”. In: *Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2006*. Vol. 3920 LNCS. Springer, pp. 411–425. DOI: 10.1007/11691372_27.

Ouaknine, Joël and James Worrell (2008). “Some recent results in metric temporal logic”. In: *Formal Modeling and Analysis of Timed Systems. FORMATS 2008*. Vol. 5215 LNCS. Springer, pp. 1–13. DOI: 10.1007/978-3-540-85778-5_1.

Ovsiannikova, Polina, Igor Buzhinsky, Antti Pakonen, and Valeriy Vyatkin (2021). “Oeritte: User-Friendly Counterexample Explanation for Model Checking”. In: *IEEE Access* 9, pp. 61383–61397. DOI: 10.1109/ACCESS.2021.3073459.

Pakonen, Antti, Igor Buzhinsky, and Valeriy Vyatkin (2018). “Counterexample visualization and explanation for function block diagrams”. In: *Proceedings - IEEE 16th International Conference on Industrial Informatics, INDIN 2018*. IEEE, pp. 747–753. DOI: 10.1109/INDIN.2018.8472025.

Pnueli, Amir (1977). “The temporal logic of programs”. In: *18th Annual Symposium on Foundations of Computer Science (SFCS 1977)*. Vol. 1977-Octob. IEEE, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

Renieris, Manos and Steven P. Reiss (2003). “Fault localization with nearest neighbor queries”. In: *Proceedings - 18th IEEE International Conference on Automated Software Engineering, ASE 2003*, pp. 30–39. DOI: 10.1109/ASE.2003.1240292.

TRACE4CPS User Manual (2021). URL: https://www.eclipse.org/trace4cps/html/content/manual.html.

Wang, Chao, Zijiang Yang, Franjo Ivančić, and Aarti Gupta (2006). “Whodunit? Causal Analysis for Counterexamples”. In: *Automated Technology for Verification and Analysis. ATVA 2006*. Vol. 4218 LNCS. Springer, pp. 82–95. DOI: 10.1007/11901914_9.

Zeller, Andreas (2002). “Isolating cause-effect chains from computer programs”. In: *ACM SIGSOFT Software Engineering Notes* 27.6, pp. 1–10. DOI: 10.1145/605466.605468.