Towards Proved Formal Specification and Verification of STL Operators as Synchronous Observers

Céline Bellanger
ENAC, Université de Toulouse

Pierre-Loïc Garoche
ENAC, Université de Toulouse

Matthieu Martel
Université de Perpignan Via Domitia

Célia Picard
ENAC, Université de Toulouse

Signal Temporal Logic (STL) is a convenient formalism to express bounded horizon properties of autonomous critical systems. STL extends LTL to real-valued signals and associates a non-singleton bound interval to each temporal operators. In this work we provide a rigorous encoding of non-nested discrete-time STL formulas into Lustre synchronous observers.

Our encoding provides a three-valued online semantics for the observers and therefore enables both the verification of the property and the search of counter-examples. A key contribution of this work is an instrumented proof of the validity of the implementation. Each node is proved correct with respect to the original STL semantics. All the experiments are automated with the Kind2 model-checker and the Z3 SMT solver.

1 Introduction

In the context of autonomous critical systems, an undesirable behaviour can lead to significant material or human damage. Thus, the specification of properties and their formal verification play a paramount role in ensuring the safety, reliability and compliance of such systems.

Dynamical systems continuously respond to environmental changes. Signal Temporal Logic (STL) has emerged as a powerful formalism for expressing temporal properties within these systems [16]. The main particularity of STL language is the association of each temporal operator with a finite, non-singleton time interval, during which the operator is studied. Consider the temporal $\Diamond$ *(Eventually)* operator, which evaluates whether a property φ is satisfied or not at least once. A correct formalism for $\Diamond$ in STL is $\Diamond_{[a,b]}\varphi$, where a and b are times such that $a < b$. Most of the time, STL properties are assess offline: we execute the system from start to finish, and we observe after the end of the execution if the system behaviour and its outputs are compliant to specified requirements.

However, the complexity of certain autonomous dynamical systems may require runtime verification. This involves continuous assessment of the system's compliance to its specification throughout execution. Synchronous observers can be employed for this purpose. These specialized observers react when a property is satisfied or violated, providing instantaneous information about the system's state. This approach offers advantages such as consistent real-time information transmission and the ability to halt executions immediately upon property satisfaction or violation, without waiting for completion. Notably, this enables quicker reactions to external events, crucial for critical systems. For instance, let us admit that we wish to satisfy a property φ at least once during a time interval $[a,b]$. If the property is satisfied in the interval, then there is no need to wait for time b to affirm that the property is indeed verified.

This paper introduces preliminary works on the specification and verification of STL operators, using synchronous observers. The rest of the document focuses on discrete times, and non nested temporal

M. Farrell, M. Luckcuck, M. Schwammberger, and
M. Gleirscher (Eds): Fifth International Workshop on
Formal Methods for Autonomous Systems (FMAS 2023)
EPTCS 395, 2023, pp. 188–204, doi:10.4204/EPTCS.395.14

$$
\begin{aligned}
(\mathcal{X},t) \vDash \mu &\Leftrightarrow \mu(t) && (1)\\
(\mathcal{X},t) \vDash \neg\varphi &\Leftrightarrow \neg((\mathcal{X},t) \vDash \varphi) && (2)\\
(\mathcal{X},t) \vDash \varphi_1 \wedge \varphi_2 &\Leftrightarrow (\mathcal{X},t) \vDash \varphi_1 \wedge (\mathcal{X},t) \vDash \varphi_2 && (3)\\
(\mathcal{X},t) \vDash \varphi_1 \vee \varphi_2 &\Leftrightarrow (\mathcal{X},t) \vDash \varphi_1 \vee (\mathcal{X},t) \vDash \varphi_2 && (4)\\
(\mathcal{X},t) \vDash \varphi_1 \,\mathcal{U}_{[a,b]}\, \varphi_2 &\Leftrightarrow \exists t' \in t+[a,b] : (\mathcal{X},t') \vDash \varphi_2 \wedge \forall t'' \in [t,t'] : (\mathcal{X},t'') \vDash \varphi_1 && (5)\\
(\mathcal{X},t) \vDash \Diamond_{[a,b]}\, \varphi &\Leftrightarrow \exists t' \in t+[a,b] : (\mathcal{X},t') \vDash \varphi && (6)\\
(\mathcal{X},t) \vDash \Box_{[a,b]}\, \varphi &\Leftrightarrow \forall t' \in t+[a,b] : (\mathcal{X},t') \vDash \varphi && (7)
\end{aligned}
$$

Figure 1: STL offline semantics

operators. For example, STL properties like $\Box_{[a,b]}(\Diamond_{[c,d]}\varphi)$ with φ an atomic proposition and a, b, c and d distinct times such that $a < b$ and $c < d$, are excluded due to the nested $\Diamond$ operator.

Our main contribution concerns the formal verification of the correctness of STL operators. To this end, we provide a three-valued online STL semantics as well as the implementation of each STL operator in the synchronous language Lustre. The soundness of the implementation is expressed as a set of lemmas and automatically proved with the Kind2 model-checker.

Section 2 covers the preliminary concepts, including the Signal Temporal Logic, the synchronous language Lustre, the model checker Kind2, and an introduction to three-valued logic. We formalise an online semantics for STL operators in Section 3, and detail its Lustre implementation in Section 4. Finally, Section 5 describes the formal correction of the operators implementation.

2 Preliminaries

2.1 Signal Temporal Logic

Let $\mathbb{T}$ denote a set of discrete times such that $\mathbb{T} = \mathbb{N}$ and let $\mathcal{X}$ be a finite sets of signals. Let $a, b \in \mathbb{T}$ with $a < b$. Without loss of generality, we assume that all signals are defined as functions in $\mathbb{T} \to \mathbb{R}$ from time to real values. To simplify notations, we denote the set of time $[t+a, t+b]$ as $t+[a,b]$.

Definition 1 (STL formal grammar)**.** *Let μ be an atomic predicate whose value is determined by the sign of a function of an underlying signal $x \in \mathcal{X}$, i.e., $\mu(t) \equiv \mu(x(t)) > 0$. Let φ, ψ be STL formulas. STL formula φ is defined inductively as:*

$$\varphi ::= \mu \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \Box_{[a,b]}\,\psi \mid \Diamond_{[a,b]}\,\psi \mid \varphi\,\mathcal{U}_{[a,b]}\,\psi$$

Definition 2 (STL semantics)**.** *The semantics of a formula φ is defined at a time $t \in \mathbb{T}$ and for a set of signals $\mathcal{X}$ as $(\mathcal{X},t) \vDash \varphi$ as described in the Figure 1.*

μ is evaluated locally, at time t over the current values of the signals, Eq. (1)*. Equation* (2) *(Negation) is the logical negation of φ. Equation* (3) *(And) is the logical conjunction between φ_1 and φ_2. Equation* (4) *(Or) is the logical disjunction between φ_1 and φ_2.*

It is worth mentioning that, in STL, all temporal operators have to be associated to a bounded, non-singleton time interval. Equation (5) *(Until) describes a temporal operator that is satisfied if φ_1 holds from time t until φ_2 becomes True within the time horizon $t+[a,b]$. Equation* (6) *(Eventually) describes a temporal operator that is satisfied if φ is verified at least once within the time horizon $t+[a,b]$. Finally,*

Equation (7) *(Always or Globally) describes a temporal operator that is satisfied if* φ *is always verified within the time horizon* $t+[a,b]$. *Note that the usual definitions of* $\Diamond_{[a,b]}$ *and* $\Box_{[a,b]}$ *based on* $\mathcal{U}_{[a,b]}$ *still apply:*

$$\Diamond_{[a,b]}\varphi = True\ \mathcal{U}_{[a,b]}\ \varphi, \text{ and} \tag{8}$$

$$\Box_{[a,b]}\varphi = \neg(\Diamond_{[a,b]}\neg\varphi). \tag{9}$$

Remark 1. *While evaluation of predicates is performed at time* t *in* $(\mathcal{X},t)\models p \Leftrightarrow \mu(t)$, *all occurrences of time intervals* $[a,b]$ *in the definitions of* $\mathcal{U}_{[a,b]}$, $\Box_{[a,b]}$ *or* $\Diamond_{[a,b]}$ *are used to delay the current time* t: $t+[a,b]=[t+a,t+b]$. *These times* a *and* b *are then relative times while* t *acts more as an absolute time.*

2.2 Lustre

Lustre[5] is a synchronous language for modeling systems of synchronous reactive components. A Lustre program L is a finite collection of nodes $[N_0,N_1,\ldots,N_m]$. The nodes satisfy the grammar described in Table 1 in which td denotes type constructors, including enumerated types, and v either constants of enumerated types C or primitive constants such as integers i. Each node is declared by the grammar construct d of Table 1. A Lustre node N transforms infinite streams of *input* flows to streams of *output* flows, with possible local variables denoting *internal* flows. A notion of a symbolic "abstract" universal clock is used to model system progress. At each time step k, a node reads the value of each input stream and instantaneously computes and returns the value of each output stream. Note that all the equations of a node are computed at each time step. Therefore an if-then-else statement is purely functional and both of its branches are evaluated while only one of the computed value is returned.

td	::=	**type** bt | **type** t $= enum\ \{C_i,...\}$
bt	::=	$real \mid bool \mid int \mid enum\_ident$
d	::=	**node** $f\ (p)$ **returns** (p); **vars** p **let** D **tel**
p	::=	$x:bt;...;x:bt$
D	::=	$pat=e;D \mid pat=e;$
pat	::=	$x \mid (pat,...,pat)$
e	::=	$v \mid x \mid (e,...,e) \mid e \to e \mid op(e,...,e)$ | **if** e **then** e **else** e | **pre** e
v	::=	$C \mid i$

Table 1: A subset of Lustre syntax

Stateful constructs. Two important Lustre operators are the unary right-shift `pre` (for `pre`vious) operator and the binary initialization $\to$ (for followed-by) operator. Their semantics is as follows. For the operator *Pre:* at first step $k=0$, `pre` p is undefined, while for each step $k>0$ it returns the value of p at $k-1$. For the operator $\to$*:* At step $k=0$, $p \to q$ returns the value of p at $k=0$, while for $k>0$ it returns the value of q at k step.

For example, the Lustre equation $y=x_0 \to pre(u)$; will be defined for each time step k by:

$$y(k)=\begin{cases} x_0(0) & \text{if } k=0 \\ u(k-1) & \text{if } k>0 \end{cases}$$

2.3 Specifying and verifying assume-guarantee contracts with Kind2

The annotation language CoCoSpec [6] was proposed for Lustre models to lift the notion of Hoare triple [12] and Assume/Guarantee statements as dataflow contracts. A contract is associated to a node and has only access to the input/output streams of that node. The body of a contract may contain a set of `assume` (A) and `guarantee` (G) statements and `mode declarations`. Modes are named and consist of `require` (R) and `ensure` (E) statements. Assumes, guarantees, requires, and ensures

```
1  node timeab (const a,b: int) returns (time: bool);
2  (*@contract
3      var clk : int = 0 -> 1 + pre clk;
4      assume a >=0;
5      guarantee time = (clk >= a and clk <= b);
6  *)
```

Figure 2: Example of a Lustre contract implementation

are all Boolean expressions over streams. In particular, assumptions and requires are expressions over input streams, while guarantees and ensures are expressions over input/output streams. A synchronous observer corresponds to such a contract with only a guarantee statement. A node *satisfies* a contract $C = (A, G')$ if it satisfies $Historically(A) \Rightarrow G'$, where $G' = G \cup \{R_i \Rightarrow E_i\}$ and $Historically(A)$ when A is true at all time.

Contracts can also define local flows, acting as *ghost variables*. These potentially stateful flows can then be used in `guarantees` and `ensure` statements.

The following is an example of function `timeab` in Lustre using a local contract Figure 2. `timeab` is a Lustre node indicating whether the current time is inside a given time interval $[a,b]$. It takes as inputs the integers a and b, and returns a boolean value `time` that is `True` if the current time is inside $[a,b]$. First line of the contract (line 3) defines a local variable `clk` as an integer, which initially takes the value 0 and is then incrementing at each time. The *assume* at line 4 indicates to model checker that it has to prove the Lustre node only in the cases where the condition $a \geq 0$ is satisfied. If another Lustre node is using `timeab`, Kind-2 also checks that this node could not provide an input a which runs counter to this assumption. Finally, Kind-2 must guarantee the equality, line 5, for all the inputs respecting the previous assumption, whatever the current time is. This equality compares the `time` output to value of the specification clock in a valid interval $[a,b]$. `timeab` shall calculate the same output with an internal clock bounded at time b. So here, we verify that bounding the clock has no effect on the provided output.

The Kind-2 model-checker [7] implements various SMT-based model-checking algorithms such as k-induction [20] or IC3/PDR [4] and allows to verify contracts with respect to nodes.

2.4 Three-valued logic

We present here the interest of three-valued logic, and introduce Kleene's three-valued logic, which we use in the next section to formalise an online version of STL operators.

For most tools, when performing monitoring of STL predicates, for a given value of simulation data, the trajectory is typically finite. It is produced by a simulation engine and stored in a data file. It is then loaded by the monitoring tool and analyzed with respect to the STL specification. In this offline setting, the final outcome indicates whether or not the input signal satisfies the specification. It is a boolean output.

Temporal operators are used to evaluate properties that change over time. Most of the time in these situations, we need to wait to decide whether a temporal property is satisfied or violated. Based on this observation, how to evaluate a property before being able to conclude, i.e. before the beginning of the time interval of a STL operator? Should we suppose that the operator is `True` or `False` before being able to decide?

Let us consider a property $\Box_{[0,10]}P$, a set of signals $\mathcal{X}$ and an initial time t_0, e.g., $t_0 = 0$. We are interested in checking $(\mathcal{X}, t_0) \vDash \Box_{[0,10]}P$. Let us assume that we are given with a trace for $\mathcal{X}$ of length

A and B	B		
	F	**U**	**T**
F	**F**	**F**	**F**
U	**F**	**U**	**U**
T	**F**	**U**	**T**

(a) AND Operation

A or B	B		
	F	**U**	**T**
F	**F**	**U**	**T**
U	**U**	**U**	**T**
T	**T**	**T**	**T**

(b) OR Operation

A	$\neg A$
F	**T**
U	**U**
T	**F**

(c) Negation Operation

A ⇒ B	B		
	F	**U**	**T**
F	**T**	**T**	**T**
U	**U**	**U**	**T**
T	**F**	**U**	**T**

(d) Logical implication

Table 2: Truth Tables showing Kleene's 3-valued strong logic operations

$l < 10$, e.g., $l = 8$, where the predicate P is valid along the whole trace. What is the validity of such a predicate? On the one hand, it is always valid, but on the other hand, it has no real definition within the time $[l, 10]$. Indeed, P could be false at time $t = 9$ or $t = 10$ and the property would be violated.

Since STL semantics requires all temporal connectors to be associated with bounded intervals, any STL predicate has a bounded horizon limit, after which it is always possible to determine the validity of a formula. We can then use this limit to evaluate temporal operators from it, considering that the values returned before may be irrelevant. But, in some case, the validity of the predicate can already be determined. In the previous example, if P is not valid at time $t = 2$, we already know at this time that the operator will not be satisfied at the end of the time interval. Existing works regarding online semantics for STL [8] try to optimize the runtime evaluation of the predicate monitoring, detecting when one can conclude, positively or negatively.

Rather than optimizing execution time based on binary logic, Łukasiewicz proposed a three-valued logic [15]. This logic introduces a third truth-value, `Unknown` (**U**), describing values for which we are not yet able to conclude if the property is satisfied or violated. Later, Kleene proposes a strong logic of indeterminacy [14] similar to Łukasiewicz logic. The main difference lies in the returned value for implication. Kleene's approach states that $\mathbf{U} \Rightarrow \mathbf{U}$ is `Unknown` while Łukasiewicz considers that $\mathbf{U} \Rightarrow \mathbf{U}$ should be `True`. In our use of three-valued logic, we base ourselves on Kleene strong logic.

Table 2 presents the truth tables showing the logical operations AND, OR, the logical implication, as well as the negation for Kleene's strong logic.

3 Online semantics for STL

To evaluate STL properties online, we rely on Kleene strong three-valued logic introduced in Table 2. In this section, we first introduce a way to obtain a three-valued output as proposed by Kleene, from two-valued outputs. Then, we provide an online and three-valued semantics for STL properties.

Definition 3 (Positive, negative and indeterminacy logics)**.** *Each STL temporal operator can be expressed in a three-valued form. To implement it, we define three new concepts: 1. A Positive logic* **T** *returning* `True` *when the property is satisfied, and* `False` *when it is yet undetermined or negative. 2. A Negative logic* **F** *that acts like an alarm to underline a negative result, which means that a statement returns* `True` *when we are sure that the property is not satisfied, and it returns* `False` *otherwise (undetermined property or satisfied property situations). 3. An Indeterminacy logic* **U** *highlighting situations where it is not yet possible to conclude about the satisfaction or violation of the property.*

Definition 4. *Let φ, φ_1 and φ_2 be STL properties. We denote by $\mathbf{T}_\varphi^t$ (resp. $\mathbf{U}_\varphi^t$ and $\mathbf{F}_\varphi^t$) the evaluation of $(\mathcal{X}, t) \models \varphi$ according to the positive (resp. indeterminacy and negative) logic. We denote by $\mathbf{B}_\varphi$ the evaluation of φ according to the offline implementation introduced in Figure 1. Note that $\mathbf{B}_\varphi$ does not depend on time instant t.*

Property 1 (Complete and pairwise distinct)**.** *At any time instant t, exactly one of the three logic returns* `True` *for a given property.*

$$\mathbf{T}_\varphi^t \vee \mathbf{U}_\varphi^t \vee \mathbf{F}_\varphi^t \quad \textit{(completeness)} \tag{10}$$

$$\neg((\mathbf{T}_\varphi^t \wedge \mathbf{F}_\varphi^t) \vee (\mathbf{T}_\varphi^t \wedge \mathbf{U}_\varphi^t) \vee (\mathbf{U}_\varphi^t \wedge \mathbf{F}_\varphi^t)) \quad \textit{(disjointness)} \tag{11}$$

Remark 2 (Deduction of the output of the third logic)**.** *According to Property 1, we only need the output of a given property in two of these three logics to determine its output in the last one. For example, if a property* φ *returns* `False` *in positive and negative logic, it means that* φ *is still Unknown (*`True` *in the indeterminacy logic).*

Remark 3 (Property determination)**.** *There exists an instant* t_d *from which we cannot satisfy* $\mathbf{U}_\varphi^{t \geq t_d}$*. For a non-nested temporal operator evaluated on time interval* $[a,b]$*,* t_d *corresponds at the latest to* $t+b$*.*

$$\exists t_d \leq t+b : \forall t' \geq t_d, \neg \mathbf{U}_\varphi^{t'} \tag{12}$$

Property 2. *From a specific time instant* t_f*, the offline and online results are similar. Thus, the outputs of the offline* $\mathbf{B}_\varphi$ *and online* $\mathbf{T}_\varphi^{t_f}$ *versions are equivalent. In the same way, the negation of the offline operator is equivalent to the online negative version* $\mathbf{F}_\varphi^{t_f}$*. For a non-nested temporal operator evaluated on time interval* $[a,b]$*, this time instant corresponds at the latest to* $t+b$*:*

$$\tau \geq t+b \implies ((\mathbf{B}_\varphi \iff \mathbf{T}_\varphi^\tau) \wedge (\neg \mathbf{B}_\varphi \iff \mathbf{F}_\varphi^\tau)) \tag{13}$$

Property 3 (Immutability: Positive and negative logics are final)**.** *If a property is satisfied in the positive (resp. negative) logic, it will remain so in the future.*

$$\exists t \in \mathbb{T} : \mathbf{T}_\varphi^t \implies \forall t' \geq t, \mathbf{T}_\varphi^{t'} \tag{14}$$

$$\exists t \in \mathbb{T} : \mathbf{F}_\varphi^t \implies \forall t' \geq t, \mathbf{F}_\varphi^{t'} \tag{15}$$

From these three logics, we obtain easily a three-valued output. The property is: 1. `True` in three-valued logic if it is `True` in positive logic; 2. `False` in three-valued logic if it is `True` in negative logic; 3. `Unknown` in three-valued logic if it is `True` in indeterminacy logic.

Let us now characterize for each construct, the sufficient and necessary conditions to determine a positive, a negative, or a temporary indeterminate value.

In the case of a non-temporal property, the property validity can always be determined as either satisfied or violated. Let μ be an atomic proposition, ie. non temporal, and $t \in \mathbb{T}$:

$$\forall \mathcal{X}, \forall t, \quad (\mathcal{X},t) \vDash \mu \iff \mathbf{T}_\mu^t \quad (\mathcal{X},t) \vDash \neg\mu \iff \mathbf{F}_\mu^t \quad \mathbf{U}_\mu = \bot \tag{16}$$

If the property is a combination of multiple predicates based on logical operators ($\wedge, \vee, \implies, \neg\varphi, \ldots$), the validity is obtained using Kleene's three-valued strong logic presented in Table 2.

In the case of STL temporal operators as described in Figure 1, we define a three-valued semantics describing when each operator is `True`, `False` or `Unknown`. For positive and negative logics, we also provide an explicit version obtained by enumerating all the terms in the time horizon $t+a$ and $t+b$. The unknown explicit version for a property `P` can be obtained by combining the positive and negative explicit versions :

$$\mathbf{U}_P^\tau \text{ (explicit)} \iff \left(\neg \mathbf{T}_P^\tau \text{ (explicit)}\right) \wedge \left(\neg \mathbf{F}_P^\tau \text{ (explicit)}\right) \tag{17}$$

Let us describe the positive, negative and indeterminate versions of each STL operator:

$$\mathbf{T}_P^\tau \quad \tau \geq t+a \wedge \exists t' \in [t+a, min(\tau, t+b)] : (\mathcal{X}, t') \vDash \varphi \tag{18}$$

$$\mathbf{F}_P^\tau \quad \tau \geq t+b \wedge \forall t' \in [t+a, t+b], (\mathcal{X}, t') \vDash \neg\varphi \tag{19}$$

$$\mathbf{U}_P^\tau \quad (\tau < t+a) \vee (\tau < t+b \wedge \forall t' \in [t+a, \tau], (\mathcal{X}, t') \vDash \neg\varphi) \tag{20}$$

$$\mathbf{T}_P^\tau \text{ (explicit)} \quad \big((\mathcal{X}, t+a) \vDash \varphi\big) \vee \big((\mathcal{X}, t+a+1) \vDash \varphi\big) \vee \ldots \vee \big((\mathcal{X}, t+b-1) \vDash \varphi\big) \vee \big((\mathcal{X}, t+b) \vDash \varphi\big) \tag{21}$$

$$\mathbf{F}_P^\tau \text{ (explicit)} \quad \big((\mathcal{X}, t+a) \vDash \neg\varphi\big) \wedge \big((\mathcal{X}, t+a+1) \vDash \neg\varphi\big) \wedge \ldots \wedge \big((\mathcal{X}, t+b-1) \vDash \neg\varphi\big) \wedge \big((\mathcal{X}, t+b) \vDash \wedge\neg\varphi\big) \tag{22}$$

Figure 3: Three-valued semantics of Eventually operator: $P = \Diamond_{[a,b]}\varphi$

$$\mathbf{T}_P^\tau \quad \tau \geq t+b \wedge \forall t' \in [t+a, t+b], (\mathcal{X}, t') \vDash \varphi \tag{23}$$

$$\mathbf{F}_P^\tau \quad \tau \geq t+a \wedge \exists t' \in [t+a, min(\tau, t+b)] : (\mathcal{X}, t') \vDash \neg\varphi \tag{24}$$

$$\mathbf{U}_P^\tau \quad (\tau < t+a) \vee (\tau < t+b \wedge \forall t' \in [t+a, \tau], (\mathcal{X}, t') \vDash \varphi) \tag{25}$$

$$\mathbf{T}_P^\tau \text{ (explicit)} \quad \big((\mathcal{X}, t+a) \vDash \varphi\big) \wedge \big((\mathcal{X}, t+a+1) \vDash \varphi\big) \wedge \ldots \wedge \big((\mathcal{X}, t+b-1) \vDash \varphi\big) \wedge \big((\mathcal{X}, t+b) \vDash \varphi\big) \tag{26}$$

$$\mathbf{F}_P^\tau \text{ (explicit)} \quad \big((\mathcal{X}, t+a) \vDash \neg\varphi\big) \vee \big((\mathcal{X}, t+a+1) \vDash \neg\varphi\big) \vee \ldots \vee \big((\mathcal{X}, t+b-1) \vDash \neg\varphi\big) \vee \big((\mathcal{X}, t+b) \vDash \neg\varphi\big) \tag{27}$$

Figure 4: Three-valued semantics of Always operator: $P = \Box_{[a,b]}\varphi$

Eventually $\Diamond_{[a,b]}\varphi$ (Fig. 3) In the positive logic, Eq. (18), we need to wait for time $t+a$, the beginning of the time interval, to have a chance to conclude positively if a valid condition has been observed. From time $t+b$, if the condition was not yet valid, the positive eventually operator always returns `False`. For the negative logic, Eq. (19), invalidity requires to wait until the end of the time interval, otherwise one cannot conclude. Finally, the validity is unknown if we have not yet reached the end of the time interval but have not yet observed a suitable time, Eq. (20).

The explicit positive version Equation (21) is obtained by considering each instant between $t+a$ and $t+b$. One of these instants is supposed to satisfy the property. We use the disjunction between all the terms to check it. At the opposite, explicit negative version Equation (22) returns `True` if all the terms between $t+a$ and $t+b$ satisfy $\neg\varphi$. We therefore rely on the conjunction between all the terms.

Always $\Box_{[a,b]}\varphi$ (Fig. 4) In the positive logic, Eq. (23), similarly to the negative case of the eventually operator, one needs to wait until the end of the interval to claim validity. For the negative logic. Eq. (24), we detect invalidity as soon as we observe an invalid time, within the proper time interval. Unknown cases are either before the time interval or within it, if the property φ is valid, up to now, Eq. (25).

The Always explicit positive version Equation (26) returns `True` if each instant between $t+a$ and $t+b$ satisfies φ. Similarly to the explicit negative version of Eventually, we use the conjunction to verify this point. For the explicit negative version to return `True`, it suffices that at one instant between $t+a$ and $t+b$, the property φ is not satisfied. Thus, we check the disjunction of all the terms, searching if one of them violates φ.

$$\mathbf{T}_P^\tau \quad (\tau \geq t+a) \wedge (\ \exists t_1 \in [t+a, min(\tau, t+b)] : (\mathcal{X}, t_1) \vDash \varphi_2 \wedge \forall t_2 \in [t, t_1], (\mathcal{X}, t_2) \vDash \varphi_1\ ) \tag{28}$$

$$\begin{aligned}\mathbf{F}_P^\tau \quad & (\exists t_6 \in [t, min(\tau, t+a)] : (\mathcal{X}, t_6) \vDash \neg\varphi_1) \quad \vee \\ & (\tau \geq t+a \wedge \tau < t+b \wedge \exists t_7 \in [t+a, \tau] : (\mathcal{X}, t_7) \vDash \neg\varphi_1 \wedge \\ & \quad \neg(\exists t_8 \in [t+a, \tau] : (\mathcal{X}, t_8) \vDash \varphi_2 \wedge \forall t_9 \in [t, t_8], (\mathcal{X}, t_9) \vDash \varphi_1)) \quad \vee \\ & (\tau \geq t+b \wedge \neg(\exists t_{10} \in [t+a, t+b] : (\mathcal{X}, t_{10}) \vDash \varphi_2 \wedge \forall t_{11} \in [t, t_{10}], (\mathcal{X}, t_{11}) \vDash \varphi_1))\end{aligned} \tag{29}$$

$$\begin{aligned}\mathbf{U}_P^\tau \quad & (\tau < t+a \wedge \forall t_3 \in [t, \tau], (\mathcal{X}, t_3) \vDash \varphi_1) \quad \vee \\ & \big(\tau \geq t+a \wedge \tau < t+b \wedge \forall t_4 \in [t, \tau], (\mathcal{X}, t_4) \vDash \varphi_1 \quad \wedge \forall t_5 \in [t+a, \tau], (\mathcal{X}, t_5) \vDash \neg\varphi_2\big)\end{aligned} \tag{30}$$

$$\begin{aligned}\mathbf{T}_P^\tau \text{ (explicit)} \quad & \Big(\big(\bigwedge_{n=0}^{a}(\mathcal{X}, n) \vDash \varphi_1\big) \wedge \big((\mathcal{X}, t+a) \vDash \varphi_2\big)\Big) \vee \Big(\big(\bigwedge_{n=0}^{a+1}(\mathcal{X}, n) \vDash \varphi_1\big) \wedge \big((\mathcal{X}, t+a+1) \vDash \varphi_2\big)\Big) \vee \ldots \vee \\ & \Big(\big(\bigwedge_{n=0}^{b-1}(\mathcal{X}, n) \vDash \varphi_1\big) \wedge \big((\mathcal{X}, t+b-1) \vDash \varphi_2\big)\Big) \vee \Big(\big(\bigwedge_{n=0}^{b}(\mathcal{X}, n) \vDash \varphi_1\big) \wedge \big((\mathcal{X}, t+b) \vDash \varphi_2\big)\Big)\end{aligned} \tag{31}$$

$$\begin{aligned}\mathbf{F}_P^\tau \text{ (explicit)} \quad & \Big(\bigvee_{n=0}^{a}(\mathcal{X}, n) \vDash \neg\varphi_1\Big) \vee \Big(\bigvee_{n_1=a+1}^{b}\big((\mathcal{X}, n_1) \vDash \neg\varphi_1 \wedge \big(\bigwedge_{n_2=a}^{n_1}(\mathcal{X}, n_2 - 1) \vDash \neg\varphi_2\big)\big)\Big) \vee \\ & \Big(\bigwedge_{n=a}^{b}(\mathcal{X}, n) \vDash \neg\varphi_2\Big)\end{aligned} \tag{32}$$

Figure 5: Three-valued semantics of Until operator: $P = \varphi_1 \mathcal{U}_{[a,b]} \varphi_2$

Until $\varphi_1 \mathcal{U}_{[a,b]} \varphi_2$ (Fig. 5) Until operator is the most complex. We conclude positively when an event φ_2 occurred within the proper time interval, and until this moment φ_1 was always satisfied, Eq. (28). For the negative logic, there are multiple conditions that can lead to a violation of the property. First before the time interval, if φ_1 is not satisfied. Then inside the time interval, if $(\mathcal{X}, \tau) \vDash \neg\varphi_1$ before the moment when $(\mathcal{X}, \tau) \vDash \varphi_2$, or if it was false before. Finally from $t+b$, if φ_2 is never reached inside the time interval or if it was false before, Eq. (29). About indeterminacy, we cannot yet conclude on the validity of the formula, if, for the moment the formula is neither validated nor violated. A first condition is that φ_1 holds from time t until now. A second is that, at the current time τ, we have not reach yet $t+a$ or we always have $\neg\varphi_2$. These condition only apply before reaching the end of the time interval $t+b$, Eq. (30).

As the Until operator depends at the same time to the satisfaction of a property inside the time interval, and the satisfaction of another one before and inside the time interval, the explicit versions are less trivial to obtain than for others operators. For the explicit positive version to return `True`, we need to satisfy the Until property at least once between $t+a$ and $t+b$, so we proceed by disjunction. Each term of the disjunction is satisfied only if φ_1 is satisfied from time t until this time instant included (conjunction between all the terms between instant t and this time instant) and φ_2 is satisfied at this moment. Note that time t is represented by the 0 value in the Until temporal referential, in the same way that instants $t+a$ or $t+b$ correspond to time a or b inside Until. As there are several ways of violated the Until operator, explicit false version of Until is built differently as others explicit versions. Indeed, we have a disjunction between the three possibilities to not satisfy the Until operator, as described above. We verify if φ_1 is not satisfied before or at time $t+a$ by relying on the disjunction between all the φ_1 terms from t until $t+a$. Then, inside the time interval after time $t+a$, the property is violated with certainty if there exist a moment where φ_1 is not `True`, and until the previous time, φ_2 was never satisfied. Indeed, if φ_2

was satisfied previously, either the property is satisfied, which means that explicit negative version must return `False`; either the property was already violated before, so there exist another anterior time where φ_1 was not satisfied before φ_2 was satisfied. Finally, explicit negative version must return `True` if φ_2 is never satisfy inside the time interval, which is studied by examining the conjunction of all the $\neg\varphi_2$ terms between $t+a$ and $t+b$.

4 Operators implementation strategy

Based on this online semantics, we propose an implementation of *Eventually*, *Always* and *Until* in discrete time. We use the synchronous language Lustre. We recall that all the nodes are available at `https://garoche.net/publication/2023_fmas_submission/`.

Useful constructs for the implementation First, we define the basic nodes needed to implement the temporal operators. Node `min` returns the minimum value between two variables. Node `exist(time :bool; prop: bool)` returns `True` as soon as a property `prop` has been satisfied during the time interval represented by `time`. Node `forall_a(time: bool ; prop: bool)` returns `True` if a property `prop` has always been `True` during the time interval. All these nodes can easily be implemented in Lustre.

Regarding the implementation of the nodes detecting whether or not we are in the time slot $t+[a,b]$, and since we work with finite intervals, we can optimize our clock, preventing it from incrementing to infinity. We can limit the counter until value b, ensuring the absence of overflow. We implement the node `timeab`, that returns `True` if the current time instant is inside the time interval, based on this bounded internal counter. As counter stops at b, end of the time interval is intercepted looking at the counter previous value. If it was already b, we know that we exceeded the end of the time interval.

We are now able to implement our nodes for each version of each operator. In the case of the *Positive* and *Negative* versions, we want to stay as close as possible to the definition proposed in the Section 3. We take two liberties in order to optimise the memory management. First, for each operator, we define a bounded internal clock as described above. The same strategy as for the counter is used to determine the end of the time interval, comparing the previous value of the internal node counter with `b`. Secondly, we want to have a bounded number of memories, not dependent on the trace-length or on the length of the time interval. We proceed as described in the literature [9], by reusing the outputs obtained at the previous time instant to obtain the outputs at the current one. For example, here is the implementation of the Until False node: Figure 6. Others *Positive* and *Negative* versions are obtained based on the same principle. Last, we deduce the *Unknown* version from the *Positive* and *Negative* versions, as described in the Property 1.

Remark 4. *The case where both Positive and Negative versions of the operator are* `True` *at the same time is never supposed to happen and would result in an error. Indeed, it would mean that the property is both satisfied and violated, which is impossible. This result comes directly from Property 1.*

Note that these implementations can only represent non-nested STL operators. That is to say that we only consider $\Diamond_{[a,b]}\varphi$, $\Box_{[a,b]}\varphi$ and $\varphi_1\mathcal{U}_{[a,b]}\varphi_2$ with φ, φ_1 and φ_2 being non-temporal predicates.

5 Formal verification of STL operators

In this section, we demonstrate by model checking that the operators implementation described in Section 4 corresponds to the given specification, as presented in Section 3. We first introduce the formalizing

```
1  node until_false (a,b: int ; phi1, phi2: bool)
2      returns (result_until_false: bool);
3  var until_time: int;
4  let
5      -- internal clock
6      until_time = min(0 -> pre until_time + 1, b);
7
8      -- init t=0 : until is violated if phi1 is false
9      result_until_false = not phi1 ->
10
11     -- violated if phi1 false before a
12     ((until_time <= a) and (not phi1)) or
13
14     -- violated if we are in the time interval,...
15     (until_time > a and until_time <= b and
16       exist(timeab(a,b), not phi1)
17       -- and before this moment we never had
18       and not (exist(timeab(a,b),
19       -- phi2 is true and until this moment phi1 is true.
20       ((phi2) and forall_a(timeab(0,b),phi1))))) or
21
22     -- violated if there is no instant in the time interval
23     ((until_time >= b) and not (exist(timeab(a,b),
24     -- where phi2 is true and until this moment
25     -- phi1 is true
26       ((phi2) and forall_a(timeab(0,b),phi1))))) or
27
28     -- still violated if it was violated once in the past
29     pre result_until_false;
30 tel
```

Figure 6: Until Lustre node for the False version of the operator

```
1  node P_at_k (const k: int; clk:int; P:bool)
2      returns (ok: bool);
3  let
4    ok = if clk = k then P else (false -> pre ok);
5  tel
```

Figure 7: *P_at_k* Lustre node

of each STL operator proof node for positive, negative and three-valued versions. Then, we present the use of Kind2 to concretely verify these proof nodes.

5.1 Induction on time interval size

To demonstrate the correctness of the positive and negative versions of the operators, we compare the outputs of our implementation proposition for each operator and an explicit equivalent, as provided in Equations (21), (22), (26), (27), (31) and (32). We remind that the explicit version is obtained by enumerating all the terms in the time horizon $t+[a,b]$. This allows to check directly the value of each term of the operator, and hence, to be sure to understand the obtained output. We have to show that our implementation and the explicit one are equivalent for any time interval. We prove this property by strong structural induction on the time interval size.

We proceed as follows. In a first time, we demonstrate that a statement is true for the smallest possible STL time interval, cf. base case of Eq. (33). Then, we demonstrate that if the statement is true for a given time interval size, it is also true when we increase the size interval by 1, cf. Eq. (35). By verifying these two properties, we demonstrate the correctness of our operators for all intervals $[a,b]$ such that $a,b \in \mathbb{T} \wedge a < b$.

Base case: $[a,a+1]$ Let **Op** be a version of a temporal operator, and **Op_exp** its explicit representation as described in Section 3.

$$\forall a \in \mathbb{T}, \mathbf{Op}_{[a,a+1]}\varphi \iff \mathbf{Op\_exp}_{[a,a+1]}\varphi \tag{33}$$

For the base case proof, we create a new Lustre node `P_at_k`, cf. Fig. 7 that checks if a property is satisfied at a specific time or was satisfied before. This allows us to implement the explicit case. Let us take the example of the *Positive* version of the Eventually. For the $[a,a+1]$ time interval, its explicit version is Eq. (34) and its implementation corresponds to the lines 13 and 14 of the Figure 8

$$\Diamond_{[a,a+1]}\varphi \equiv ((\mathcal{X},a) \vDash \varphi) \vee ((\mathcal{X},a+1) \vDash \varphi) \tag{34}$$

Inductive case: $[a,b+1]$ Let **Op** be a version of a temporal operator and **Op_exp** its explicit representation as described in Section 3.

$$(\mathbf{Op}_{[a,b]}\varphi \iff \mathbf{Op\_exp}_{[a,b]}\varphi) \implies (\mathbf{Op}_{[a,b+1]}\varphi \iff \mathbf{Op\_exp}_{[a,b+1]}\varphi) \tag{35}$$

In our implementation, $\mathbf{Op\_exp}_{[a,b+1]}$ is obtained thanks to the previous value of $\mathbf{Op}_{[a,b]}$, that we assume equivalent to $\mathbf{Op\_exp}_{[a,b]}$. For example, inductive case of the explicit version of Eventually

```
1          returns (base_case,ind_case: bool);
2          (*@contract
3             assume a<b and a>=0;
4             guarantee base_case;
5             guarantee ind_case;
6          *)
7          var clk : int;
8              output_ev_true, output_ev_true_bp1: bool;
9          let
10           clk = 0 -> 1 + pre clk;
11           output_ev_true = eventually_true (a,b,phi);
12           output_ev_true_bp1 = eventually_true (a,b+1,phi);
13           base_case = (b=a+1) =>
14             (output_ev_true = P_at_k(a,clk,phi) or P_at_k(a+1,clk,phi));
15
16           ind_case =
17             (output_ev_true_bp1 = (output_ev_true or P_at_k(b+1,clk,phi)));
18         tel
```

Figure 8: Eventually True proof node

```
1          node always_3v(const a,b:int; phi: bool)
2            returns (output_ev_true, output_ev_false: bool);
3          (*@contract
4            assume a<b and a>=0;
5            -- their are mutually exclusive
6            guarantee not (output_ev_true and output_ev_false);
7          *)
8          let
9              output_ev_true = eventually_true(a,b,phi);
10             output_ev_false = eventually_false(a,b,phi);
11         tel
```

Figure 9: Three-valued Eventually node in Lustre

True operator is obtained as described in Eq. (36) and its implementation corresponds to lines 16 and 17 of Figure 8

$$\Diamond_{[a,b+1]}\varphi \equiv \left(\Diamond_{[a,b]}\varphi\right) \vee \left((\mathcal{X}, b+1) \vDash \varphi\right) \tag{36}$$

To concretely check these basic and inductive cases, we use the Kind2 model checker, cf Section 2.3. For each positive and negative version of STL operators, we express the base and inductive case as two properties, ie. two lemmas inside a contract to guarantee basic and inductive case, cf. Figure 8.

Finally, to obtain a three-valued output, we need to encode the result on two booleans. We combine the positive and negative outputs - previously verified - to determine the state of the operator. According to Property 1, Unknown is obtained if Positive and Negative outputs return `False` at the same time. As a complementary check, we ensure inside a contract that Positive and Negative versions are mutually exclusive as mentioned in the Remark 4. Figure 9 summarizes the implementation of this final node in Lustre.

Node name	# Property	Method	Proof time
timeab	assume	PDR	0.339s
	assume	induction	0.351s
	guarantee	PDR	2.618s
eventually_true	-	-	-
proof_ev_true	assume	PDR	0.653s
	assume	2-induction	0.713s
	guarantee	2-induction	26.778s
	guarantee	2-induction	29.631s
eventually_false	-	-	-
proof_ev_false	guarantee	PDR	37.215s
	guarantee	PDR	37.215s
eventually_3v	assume	PDR	0.677s
	assume	2-induction	0.688s
	guarantee	2-induction	0.688s
	guarantee	2-induction	13.216s
	guarantee	2-induction	18.855s

Table 3: Experiments for operator Eventually.

5.2 Using Kind2 as a theorem prover

Each of the three temporal operators is defined in a separate file. They all rely on basic nodes mentioned in Sect. 4, in which only `timeab` is fitted with a contract. The following tables summarize all contract elements automatically proved by Kind2 model-checker. We recall that Kind2 relies on different model-checking algorithms that are executed in parallel. The method that succeeds first interrupt the proof process. In the table, *PDR* stands for Property-Direct-Reachability [4] while *k-induction* specifies the number of steps of the k-induction process used to conclude. In both methods, Kind2 produces subproblems that are solved using Z3 [18].

As mentioned above, each operator `op` is defined using two underlying nodes `op_false` and `op_true` as well as a node `op_3v` that reconstruct the three-valued output. The nodes `op_false` and `op_true` are not directly associated to a contract but their soundness is expressed through the validity of another node: respectively `proof_op_false` and `proof_op_true`. These nodes are defining the base and inductive cases and associated to the main contract (cf. Fig. 8). Last, the final node `op_3v` is only fitted with an extra contract guarantying disjunctiveness of the output (cf. Fig. 9 encoding Eq. (11)).

Experiments were run with kind2 v2.0.0-7-gdcc7f6f on a 1,2 GHz Quad-Core Intel Core i7 with 16 GB of RAM. To build the table, each node is analyzed independently, but a quicker analysis of each file can be performed with all nodes analyzed at once. Note also results with the same execution time such as the elements of the node `eventually_3v`. Typically, in this case, they denote properties that were proved together k-inductive by the algorithm. We observe something similar with PDR for node `proof_ev_false`.

As a last remark, we have to say that, because of the parallel architecture of Kind2, it is difficult to obtain perfect reproductibility of the results. For example, one can observe that the runtime of the validity proof of the simple node `timeab_tmp` varies slightly between experiments while it is the exact same node. The difference can also appear in the number of unrolling of the k-induction engine.

Node name	# Property	Method	Proof time
`timeab`	assume	PDR	0.432s
	assume	2-induction	0.454s
	guarantee	PDR	2.855s
`until_true`	-	-	-
`proof_until_true`	assume	PDR	0.537s
	assume	2-induction	0.595s
	guarantee	2-induction	6.347s
	guarantee	2-induction	93.526s
	guarantee	2-induction	161.614s
`until_false`	-	-	-
`proof_until_false`	assume	induction	0.898s
	assume	induction	0.898s
	assume	induction	0.898s
	guarantee	2-induction	606.067s
	guarantee	PDR	1605.403s
`until_3v`	guarantee	PDR	34.530s

Table 4: Experiments for operator Until.

Node name	# Property	Method	Proof time
`timeab`	assume	PDR	0.400s
	assume	induction	0.425s
	guarantee	PDR	3.375s
`always_true`	-	-	-
`proof_alw_true`	guarantee	2-induction	76.633s
	guarantee	2-induction	96.686s
`always_false`	-	-	-
`proof_alw_false`	guarantee	2-induction	19.407s
	guarantee	PDR	27.540s
`always_3v`	guarantee	PDR	12.854s

Table 5: Experiments for operator Always.

6 Discussions and conclusion

Related Works. The use of three-valued logic has already been explored in the context of temporal logic, particularly in LTL, with the same division used in this paper: one value indicating the certainty of satisfaction of a property, another indicating the certainty of violation of a property, and a final value representing indeterminacy [3, 11].

Formal verification of STL properties has also been studied. Roehm et al. [19] propose to check STL properties on reach sequences, using hybrid model checking algorithms such as Cora [1] or SpaceEx [10]. A first step consists in the transformation of STL properties into their *reachset temporal logic* (RTL) equivalent. This transformation comes close to the explicit development of each operator that we described in Section 3, requiring potentially a large set of memories.

Moreover, several examples of algorithms and online implementation of STL properties have been produced, using a finite number of memories, cf [17, 9]. Thus, [9] proposes an algorithm for quantitative online STL implementation, and show on different examples the time-saving benefits of using their online method compared to the offline one. Balsini et al. [2] propose a qualitative online implementation of STL in Simulink, which nevertheless has some limitations. In particular, since three-valued logic is not used in this implementation, we cannot be sure whether a property has been satisfied or violated until the end of execution. These proposals go further than ours, allowing operators to be nested, sometime with some limitations like [2] that can only contain one operator inside another. However the soundness of the encoding is not formally proven.

Conclusion. In this paper, we propose an online discrete implementation of the STL semantics, in the continuity of Balsini's work [2]. Our contribution is twofold. First, we proposed an implementation based on Kleene's three-valued logic in order to be able to represent indeterminacy. Second, we formally demonstrated the soundness of our implementation, proving the validity of each operator with respect to its semantics.

Our approach was the following: we first defined the online STL semantics, and used it to build each STL operator as a synchronous observer in the Lustre language. Finally, we formally demonstrated the correctness of their implementation, using the Kind2 model-checker. We proceed by induction on the size of temporal intervals. We succeed to demonstrate all the proof objectives for each temporal operator implemented.

Future Work. They are mainly two directions to continue this work. A first one is to apply these operators to models and see how model-checkers such as Kind2 can verify properties or produce counter-examples. For example revisiting the use case of Roehm et al. [19]. The other direction is to extend the set of STL formulas that can be encoded in our framework. While Balsini et al. [2] proposed a similar encoding (but without proof) of nested operators with restricted form and up to two levels, we would like to lift the restrictions and deal with more general formulas. The notion of propagation delays introduced in Kempa et al. [13] could also lead to an efficient encoding with memories, also associated with proof of the implementation.

7 Acknowledgment

The authors would like to thank the Institute for Cybersecurity in Occitania (ICO) for partially funding this work.

References

[1] Matthias Althoff (2015): *An Introduction to CORA 2015*. In: *EPiC Series in Computing*, 34, EasyChair, pp. 120–151, doi:10.29007/zbkv. Available at `https://easychair.org/publications/paper/xMm`. ISSN: 2398-7340.

[2] Alessio Balsini, Marco Di Natale, Marco Celia & Vassilios Tsachouridis (2017): *Generation of simulink monitors for control applications from formal requirements*. In: *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*, IEEE, Toulouse, pp. 1–9, doi:10.1109/SIES.2017.7993389. Available at `https://ieeexplore.ieee.org/document/7993389/`.

[3] Andreas Bauer, Martin Leucker & Christian Schallhart (2006): *Monitoring of Real-Time Properties*. In S. Arun-Kumar & Naveen Garg, editors: *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 260–272, doi:10.1007/11944836_25.

[4] Aaron R. Bradley (2012): *IC3 and beyond: Incremental, Inductive Verification*. In P. Madhusudan & Sanjit A. Seshia, editors: *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, *Lecture Notes in Computer Science* 7358, Springer, p. 4, doi:10.1007/978-3-642-31424-7_4.

[5] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs & John Plaice (1987): *Lustre: A Declarative Language for Programming Synchronous Systems*. In: *POPL'87*, pp. 178–188, doi:10.1145/41625.41641.

[6] Adrien Champion, Arie Gurfinkel, Temesghen Kahsai & Cesare Tinelli (2016): *CoCoSpec: A Mode-Aware Contract Language for Reactive Systems*. In: *SEFM'16*, pp. 347–366, doi:10.1007/978-3-319-41591-8_24.

[7] Adrien Champion, Alain Mebsout, Christoph Sticksel & Cesare Tinelli (2016): *The Kind 2 Model Checker*. In Swarat Chaudhuri & Azadeh Farzan, editors: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, *Lecture Notes in Computer Science* 9780, Springer, pp. 510–517, doi:10.1007/978-3-319-41540-6_29.

[8] Jyotirmoy V. Deshmukh, Alexandre Donzé, Shromona Ghosh, Xiaoqing Jin, Garvit Juniwal & Sanjit A. Seshia (2017): *Robust online monitoring of signal temporal logic*. *Formal Methods in System Design* 51(1), pp. 5–30, doi:10.1007/s10703-017-0286-7.

[9] Alexandre Donzé & Oded Maler (2010): *Robust Satisfaction of Temporal Logic over Real-Valued Signals*. In Krishnendu Chatterjee & Thomas A. Henzinger, editors: *Formal Modeling and Analysis of Timed Systems*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 92–106, doi:10.1007/978-3-642-15297-9_9.

[10] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang & Oded Maler (2011): *SpaceEx: Scalable Verification of Hybrid Systems*. In Shaz Qadeer Ganesh Gopalakrishnan, editor: *Proc. 23rd International Conference on Computer Aided Verification (CAV)*, LNCS, Springer, pp. 379–395, doi:10.1007/978-3-642-22110-1_30.

[11] Hsi-Ming Ho, Joël Ouaknine & James Worrell (2014): *Online Monitoring of Metric Temporal Logic*. In Borzoo Bonakdarpour & Scott A. Smolka, editors: *Runtime Verification*, Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 178–192, doi:10.1007/978-3-319-11164-3_15.

[12] C. A. R. Hoare (1969): *An Axiomatic Basis for Computer Programming*. *Commun. ACM* 12(10), pp. 576–580, doi:10.1145/363235.363259.

[13] Brian Kempa, Pei Zhang, Phillip H. Jones, Joseph Zambreno & Kristin Yvonne Rozier (2020): *Embedding Online Runtime Verification for Fault Disambiguation on Robonaut2*. In Nathalie Bertrand & Nils Jansen, editors: *Formal Modeling and Analysis of Timed Systems - 18th International Conference, FORMATS 2020, Vienna, Austria, September 1-3, 2020, Proceedings*, *Lecture Notes in Computer Science* 12288, Springer, pp. 196–214, doi:10.1007/978-3-030-57628-8_12.

[14] Stephen Cole Kleene (1952): *Introduction to Metamathematics*. North-Holland, Amsterdam.

[15] J. Lukasiewicz (1970): *Selected Works*. Available at `https://www.scribd.com/document/359602256/J-Lukasiewicz-Selected-Works-L-Borkowski-Editor`.

[16] Oded Maler & Dejan Nickovic (2004): *Monitoring Temporal Properties of Continuous Signals*. In Yassine Lakhnech & Sergio Yovine, editors: *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 152–166, doi:10.1007/978-3-540-30206-3_12.

[17] Oded Maler & Dejan Ničković (2013): *Monitoring properties of analog and mixed-signal circuits*. *International Journal on Software Tools for Technology Transfer* 15(3), pp. 247–268, doi:10.1007/s10009-012-0247-9.

[18] Leonardo Mendonça de Moura & Nikolaj S. Bjørner (2008): *Z3: An Efficient SMT Solver*. In C. R. Ramakrishnan & Jakob Rehof, editors: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, *Lecture Notes in Computer Science* 4963, Springer, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[19] Hendrik Roehm, Jens Oehlerking, Thomas Heinz & Matthias Althoff (2016): *STL Model Checking of Continuous and Hybrid Systems*. In Cyrille Artho, Axel Legay & Doron Peled, editors: *Automated Technology for Verification and Analysis*, Springer International Publishing, Cham, pp. 412–427, doi:10.1007/978-3-319-46520-3_26.

[20] Mary Sheeran, Satnam Singh & Gunnar Stålmarck (2000): *Checking Safety Properties Using Induction and a SAT-Solver*. In Warren A. Hunt Jr. & Steven D. Johnson, editors: *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*, *Lecture Notes in Computer Science* 1954, Springer, pp. 108–125, doi:10.1007/3-540-40922-X_8.