An Application of Temporal Projection to Interleaving Concurrency

Ben Moszkowski[1] and Dimitar P. Guelev[2]

[1] School of Computing Science,
Newcastle University, Newcastle upon Tyne, UK
benmos63@gmail.com
[2] Department of Algebra and Logic,
Institute of Mathematics and Informatics, Sofia, Bulgaria
gelevdp@math.bas.bg

Abstract. We revisit the earliest temporal projection operator Π in discrete-time Propositional Interval Temporal Logic (PITL) and use it to formalise interleaving concurrency. The logical properties of Π as a normal modality and a way to eliminate it in both PITL and conventional point-based Linear-Time Temporal Logic (LTL), which can be viewed as a PITL subset, are examined. We also formalise concurrency without Π, and relate the two approaches. Furthermore, Π and another standard PITL projection operator are interdefinable and both suitable for reasoning about different time granularities. We mention other (mostly interval-based) temporal logics with similar forms of projection, as well as some related applications and international standards.

Keywords: interleaving concurrency · interval temporal logic · temporal projection · time granularities

1 Introduction

Temporal intervals, which are finite and infinite state sequences, offer a compellingly natural and flexible way to model computational processes involving hardware or software. *Interval Temporal Logic* (ITL) [35], [19], [36] is an established formalism for reasoning about such phenomena. In ITL, satisfaction of formulas is defined at intervals rather than time points which are used in other temporal logics. ITL operators for sequentially combining formulas $A; B$ ("*A chop B*") and A^* ("*A chop-star*") are related to the concatenation and Kleene star operators for regular expressions.

In the early 1980s, we proposed in [35], [19] a simple binary temporal operator Π for time granularities and projection to enhance ITL's usefulness for formalising digital circuits. Here we revisit Π's logical properties and use it to formalise interleaving concurrency. We also discuss a related operator for modelling time granularities and related work on temporal projection in general.

Structure of the paper: Section 2 overviews propositional ITL. Section 3 looks at the projection operator Π. Section 4 uses Π to formalise concurrent programs and also shows how to do this without Π. Section 5 discusses related work.

2 Propositional Interval Temporal Logic

For an in-depth presentation of PITL we refer the reader to [39]; see also [36], [31] and the ITL web pages [27]. The version of PITL used here has the syntax

$$A ::= \mathit{true} \mid p \mid \neg A \mid A \vee A \mid \bigcirc A \mid A \,\mathcal{U}\, A \mid A; A \mid A^* \ , \qquad (1)$$

where p denotes a propositional variable. Owing to our purposes here, the Until operator $\mathcal{U}$ is included. We define *false*, $\wedge$, $\supset$ and $\equiv$ as usual.

PITL models time using discrete (linear) state sequences. The *set of states* Σ is the powerset 2^V of the set V of propositional variables, so each state in Σ sets every propositional variable $p, q, \ldots$ to *true* or *false*. *Local* PITL is the (standard) version of PITL with such state-based variables (instead of interval-based ones). An *interval* $\sigma = \sigma^0\sigma^1 \ldots$ is any element of $\Sigma^+ \cup \Sigma^\omega$. If σ is finite, its *interval length* $|\sigma|$ is the number of σ's states minus 1, otherwise ω. Given $i \leq j \leq |\sigma|$, $j < \omega$, $\sigma^{i..j}$ denotes $\sigma^i \ldots \sigma^j$, and $\sigma^{i\uparrow}$ is the suffix subinterval $\sigma^i\sigma^{i+1} \ldots$ of σ. We write $\sigma \models A$ for *interval* σ *satisfies* A. Formula A is *valid* if all intervals satisfy A. The definition of $\sigma \models A$ by induction on the construction of A is as follows, where i, j, k, k_i and n are natural numbers:

$\sigma \models \mathit{true}$ for any σ $\qquad$ $\sigma \models p$ iff $p \in \sigma^0$ $\qquad$ $\sigma \models \neg A$ iff $\sigma \not\models A$

$\sigma \models A \vee B$ iff $\sigma \models A$ or $\sigma \models B$ $\qquad$ $\sigma \models \bigcirc A$ iff $|\sigma| \geq 1$ and $\sigma^{1\uparrow} \models A$

$\sigma \models A \,\mathcal{U}\, B$ iff, for some $k \leq |\sigma|, \sigma^{k\uparrow} \models B$ and for all $j < k, \sigma^{j\uparrow} \models A$

$\sigma \models A; B$ iff for some $k \leq |\sigma|$, $\sigma^{0..k} \models A$ and $\sigma^{k\uparrow} \models B$, or $|\sigma| = \omega$ and $\sigma \models A$

$\sigma \models A^*$ iff either (1) $|\sigma| = 0$,
or (2) there exists a finite sequence $k_0 = 0 < k_1 < \ldots < k_n \leq |\sigma|$
such that for all $i < n$, $\sigma^{k_i..k_{i+1}} \models A$, and $\sigma^{k_n\uparrow} \models A$,
or (3) $|\sigma| = \omega$ and there exists an infinite sequence
$k_0 = 0 < k_1 < \ldots$ such that $\sigma^{k_i..k_{i+1}} \models A$ for all $i < \omega$.

In the first case for chop, intervals $\sigma^{0..k}$ and $\sigma^{k\uparrow}$ have overlapping state σ^k. Cases (1)-(3) for chop-star concern zero, nonzero but finite, and infinite (*"chop-omega"*) iterations, respectively. Chop here is *weak*, like the weak version $\mathcal{W}$ of $\mathcal{U}$, for potentially nonterminating programs which ignore B. *Strong* chop, which forces the left subinterval to be finite, is derivable.

Consider a sample 5-state interval σ with the following alternating values for the variable p: p $\neg p$ p $\neg p$ p. Here are four formulas σ satisfies:

$$p \qquad (\bigcirc \neg \bigcirc \mathit{true}); \neg p \qquad p \wedge (\mathit{true}; \neg p) \qquad \big(p \wedge \bigcirc\bigcirc(p \wedge \neg \bigcirc \mathit{true})\big)^* \ .$$

For example, $(\bigcirc \neg \bigcirc \mathit{true}); \neg p$ is true since σ's prefix subinterval $\sigma^0\sigma^1$ satisfies $\bigcirc \neg \bigcirc \mathit{true}$ (which is true exactly on 2-state intervals) and the adjacent suffix subinterval $\sigma^1 \ldots \sigma^4$ satisfies $\neg p$ because $p \notin \sigma^1$. The formula $(p \wedge \bigcirc\bigcirc \neg \bigcirc \mathit{true})^*$ is true since σ's subintervals $\sigma^0\sigma^1\sigma^2$ and $\sigma^2\sigma^3\sigma^4$ both satisfy $p \wedge \bigcirc\bigcirc \neg \bigcirc \mathit{true}$, but σ does not satisfy formulas $\neg p$, $(\bigcirc \neg \bigcirc \mathit{true}); p$ and $\mathit{true}; (\neg p \wedge \neg(\mathit{true}; p))$.

Table 1. Some Useful Derived LTL Operators

$\circledcirc A \mathrel{\widehat{=}} \neg \bigcirc \neg A$	Weak Next	$more \mathrel{\widehat{=}} \bigcirc true$	≥ 2 states
$empty \mathrel{\widehat{=}} \neg more$	One state	$skip \mathrel{\widehat{=}} \bigcirc empty$	$= 2$ states
$\Diamond A \mathrel{\widehat{=}} true\, \mathcal{U}\, A$	Eventually	$\Box A \mathrel{\widehat{=}} \neg\Diamond\neg A$	Always
$inf \mathrel{\widehat{=}} \Box more$	Infinite interval	$finite \mathrel{\widehat{=}} \neg inf$	Finite interval
$fin\, A \mathrel{\widehat{=}} \Box(empty \supset A)$	Final state (weak)	$halt\, w \mathrel{\widehat{=}} \Box(w \equiv empty)$	Halt upon test

Let w, w_1 and w_2 denote *state formulas*, which have no temporal operators. Conventional LTL can be viewed as the subset of PITL with just the temporal operators $\bigcirc$ and $\mathcal{U}$. The infinite state sequences that are common with LTL are just infinite intervals. Table 1 shows useful derived LTL operators.

Here are some sample valid PITL formulas:

$$A \supset (A;true)\ \ skip^*\ \ inf \equiv true;false\ \ (w \wedge A);B \equiv w \wedge (A;B)\ \ A \equiv (empty;A)\ .$$

We note that PITL without chop-star has the same expressiveness as LTL. With chop-star, PITL has the same expressiveness as LTL with the addition of propositional quantification (explicitly defined later in Sect. 4). That is, having propositional quantification instead of chop-star gives the same *regular* expressiveness for finite intervals and *ω-regular expressive power* (i.e., MSO(ω,<)) for infinite intervals. The LTL operator $\mathcal{U}$ is also expressible using chop, $\bigcirc$ and quantification. More details about PITL's expressiveness are found in [35], [38], [39].

3 Temporal Projection

The binary temporal operator Π for *state projection* [35], [19] provides a way to examine dynamic behaviour at certain points in time and ignore all intermediate states. Given an interval σ and a state formula w, let $\sigma|_w$ denote the sequence of σ's states satisfying w. If σ is infinite, $\sigma|_w$ can be finite or infinite. The definition of Π, whose first argument is supposed to be a state formula, is

$$\sigma \models w\ \Pi\ A \quad \text{iff} \quad \sigma^i \models w, \text{ for some } i \leq |\sigma|, \text{ and } \sigma|_w \models A\ .$$

For example, $\sigma \models p\ \Pi\ \Box q$ if p is true at some state of σ, and q is true whenever p is, i.e., if $\sigma \models \Diamond p\ \wedge\ \Box(p \supset q)$. We can generalise Π to permit arbitrary formulas for selecting projected states by using $\sigma|_B = \langle \sigma^i : i \leq |\sigma|, \sigma^{i\uparrow} \models B\rangle$ to define $\sigma \models B\ \Pi\ A$. This does not alter Π's meaning when B is a state formula.

For a fixed w, $w\ \Pi\ A$ is a normal unary modality on A. Its accessibility relation $\sigma \mapsto \sigma|_w$ is deterministic. This entails the validity of the standard modal axioms **K** and $\mathbf{D_c}$, and the *necessitation rule* N [6, 23]. These are normally written in terms of the "universal" *dual* $\neg(w\ \Pi\ \neg A)$ of $w\ \Pi\ A$, denoted $w\ \Pi^{\mathrm{u}}\ A$:

$$(\mathbf{K})\ w\ \Pi^{\mathrm{u}}\ (A \supset B) \supset (w\ \Pi^{\mathrm{u}}\ A \supset w\ \Pi^{\mathrm{u}}\ B),\ (\mathbf{D_c})\ w\ \Pi\ A \supset w\ \Pi^{\mathrm{u}}\ A,\ (N)\ \frac{A}{w\ \Pi^{\mathrm{u}}\ A}.$$

$\mathbf{K}$, $\mathbf{D_c}$ and N are sufficient to infer equivalences such as:

$$\begin{array}{lcl} w \,\Pi\, (A \wedge B) & \equiv & w \,\Pi\, A \;\wedge\; w \,\Pi\, B \\ w \,\Pi^{\mathrm{u}}\, A \;\wedge\; w \,\Pi\, B & \supset & w \,\Pi\, (A \wedge B) \\ w \,\Pi^{\mathrm{u}}\, (A \;\supset\; B) \;\wedge\; w \,\Pi\, A & \supset & w \,\Pi\, B \;. \end{array}$$

The following valid formulas are specific to Π:

$$\Box(w_1 \equiv w_2) \quad \supset \quad (w_1 \,\Pi\, A) \equiv (w_2 \,\Pi\, A)$$

$$w_1 \,\Pi\, (w_2 \,\Pi\, A) \quad \equiv \quad (w_1 \wedge w_2) \,\Pi\, A \tag{2}$$

$$w \,\Pi^{\mathrm{u}}\, A \;\equiv\; \Box\neg w \;\vee\; w \,\Pi\, A, \qquad w \,\Pi\, A \;\equiv\; \Diamond w \;\wedge\; w \,\Pi^{\mathrm{u}}\, A \tag{3}$$

$$w_1 \,\Pi\, \Diamond w_2 \;\supset\; \Diamond w_2, \qquad \Box w_2 \;\supset\; w_1 \,\Pi^{\mathrm{u}}\, \Box w_2 \tag{4}$$

The equivalences (3) give a simpler way to define Π and Π^{u} in terms of each other because $\Diamond w$ is available to indicate whether the reference interval has a nonempty projection. The implications (4) facilitate importing and exporting properties into and from the scope of Π.

The valid equivalences below form a complete axiomatisation of Π relative to basic PITL and show that every PITL formula with Π has a Π-free equivalent.

$$\begin{array}{ll} w \,\Pi\, \mathit{true} \equiv \Diamond w & w \,\Pi\, (A \vee B) \equiv w \,\Pi\, A \;\vee\; w \,\Pi\, B \\ w \,\Pi\, p \;\; \equiv (\neg w)\,\mathcal{U}\,(w \wedge p) & w \,\Pi\, (A\,\mathcal{U}\,B) \equiv (w \,\Pi\, A)\,\mathcal{U}\,(w \,\Pi\, B) \\ w \,\Pi\, \neg A \;\equiv \Diamond w \;\wedge\; \neg(w \,\Pi\, A) & w \,\Pi\, \bigcirc A \;\; \equiv (\neg w)\,\mathcal{U}\,(w \wedge \bigcirc(w \,\Pi\, A)) \\ w \,\Pi\, (A;B) \;\equiv\; (w \,\Pi\, A); (w \;\wedge\; w \,\Pi\, B) & \\ w \,\Pi\, (A^*) \;\equiv\; \neg w \;\mathcal{U}\,\big(w \wedge ((w \,\Pi\, A) \wedge \mathit{fin}\, w)^*;\, \circledcirc\Box\neg w\big) & \end{array}$$

The equivalences about the LTL operators show that LTL formulas with Π have Π-free LTL equivalents too.

By (2), $A \equiv w \,\Pi\, B$ entails $w \,\Pi\, A \equiv w \,\Pi\, B$, so A has an equivalent of the form $w \,\Pi\, B$ iff $\models A \equiv w \,\Pi\, A$. This may be useful for *synthesising* a controller to be run in parallel with other code from a global requirement R. The synthesis is possible only if $\models R \equiv (w \,\Pi\, R)$, where w marks the controller's time slices. The latter reduces to a basic ITL validity after eliminating Π from $w \,\Pi\, R$.

We originally defined Π so that $\sigma \models w \,\Pi\, A$ *vacuously* holds when $\sigma|_w$ has no states [19], [35]. This holds for Π^{u} in this paper. Projection is *false* when no projection interval exists for the real-time projection operators from [15–17], and likewise for the projection operator from [36], [37] discussed in Sect. 4.2.

4 Formalisation of Imperative Concurrent Programs

We now look at a way to formalise in ITL imperative concurrent programs in which processes are interleaved. The availability of sequential composition operators such as chop has long made ITL well suited for expressing sequential and concurrent programs and executing them in ITL-based interpreters, as we previously investigated in [36]. Such an interpreter for an ITL programming language

subset called *Tempura* is available from [27]. ITL has also been productively used for *symbolic execution* for theorem proving [2, 3]. Some later research by others on expressing concurrent programs in variants of ITL is discussed in Sect. 5.

The approach described here is specifically meant to correspond to the popular notion of *state transition systems* (based on Keller's work [30] and extensively surveyed by Baier and Katoen [1]; see also [7], [31]), where at any time only one of the program's processes is allowed in *global* time to make a transition from the current state to its immediate successor state and possibly make assignments involving just these two adjacent states. This is a quite widely employed standard assumption for interleaving found in frameworks including Manna-Pnueli *Reactive Systems* [33] (see also [4], [31]), Lamport's TLA+ [32] (including the TLC model checker), Jones' *Rely-Guarantee Conditions* [28] (see also [43]), the SPIN model checker [22] and *Partial Order Reduction* [7], [1] used by some model checkers such as SPIN. Our intention is to develop a framework that *a priori* seeks to maximise the use of ITL together with the operator Π for the interleaving model. Projection constructs are not strictly required (since they can be eliminated, as discussed later in Sect. 4.1), but we consider them here because they bring succinctness and clarity.

Interleaved parallel composition We now define

$$A \,|||_p\, B \quad \widehat{=} \quad p \,\Pi\, A \;\wedge\; (\neg p) \,\Pi\, B$$

to express that two formulas A and B operate concurrently in an interleaved manner with a boolean variable p indicating which is active in any given state. We refer to this *three-operand* interleaving operator as $|||\_$. It is commutative and associative, subject to suitable manipulations of the middle operand:

$$\models \quad A \,|||_p\, B \quad \equiv \quad B \,|||_{\neg p}\, A \tag{5}$$

$$\models \quad (A \,|||_p\, B) \,|||_q\, C \quad \equiv \quad A \,|||_{p \wedge q}\, (B \,|||_q\, C) \tag{6}$$

$$\models \quad A \,|||_p\, (B \,|||_q\, C) \quad \equiv \quad (A \,|||_p\, B) \,|||_{p \vee q}\, C \tag{7}$$

Commutativity is easily proved, as is associativity with the next valid equivalence:

$$\models \quad p \,\Pi\, (A \,|||_q\, B) \quad \equiv \quad (p \wedge q) \,\Pi\, A \;\wedge\; (p \wedge \neg q) \,\Pi\, B \;\;.$$

Whenever irrelevant, $|||\_$'s middle operand can be quantified away:

$$A \,|||\, B \quad \widehat{=} \quad \exists p.\, (A \,|||_p\, B) \;\;.$$

Here, $\sigma \models \exists p.\, C$ holds iff $\sigma' \models C$ holds for some interval σ' identical to σ except possibly for p's behaviour. [3] The definition of $|||$ here corresponds to

[3] As noted in the introduction, such quantification does not increase PITL's expressiveness: quantified formulas have equivalent quantifier-free ones. Here is how to express $\mathcal{U}$: $\models A \,\mathcal{U}\, B \equiv \Diamond B \wedge \exists p.\, \big(p \wedge \Box(p \supset (B \vee (A \wedge \bigcirc p)))\big)$ (e.g., see [31, p. 84]), where the following straightforward valid equivalences are used: $\mathit{inf} \equiv (\mathit{true}; \mathit{false})$, $\mathit{finite} \equiv \neg \mathit{inf}$, $\Diamond C \equiv (\mathit{finite}; C)$ (with $\Box$ still being $\Diamond$'s dual: $\Box C \equiv \neg\Diamond\neg C$).

Baier and Katoen's notion in [1]. With the middle operand quantified away, $|||$ is commutative and associative in the usual way. Both Π and $|||\_$ are expressible using either $|||$ or $|||\_$, so these operators can be taken as a primitive instead of Π:

$$\begin{array}{lcl} \models & A \,|||_w\, B & \equiv \quad (\Box w \wedge A) \,|||\, (\Box \neg w \wedge B) \\ \models & w \,\Pi\, A & \equiv \quad A \,|||_w\, \mathit{true} \;\vee\; (\Box w \wedge A) \;. \end{array}$$

The equivalence for expressing $w \;\Pi\; A$ needs two cases because, unlike $w \;\Pi\; A$, the disjunct $A \,|||_w\, \mathit{true}$ ensures that sometimes w is *false*.

Multiple processes with process identifiers When dealing with multiple processes, it can be convenient to associate a numerical index with each one. An auxiliary variable *pid* can be readily used for this. For instance, for a formula $A \,|||_p\, B$ with two processes, we can take *pid* to range over $\{0, 1\}$ and construct it using the formula $\Box(\mathit{pid} = \mathit{if}\ p\ \mathit{then}\ 0\ \mathit{else}\ 1)$. For any expression e and formula A, define $e :: A$ to specify that e is the process id for A:

$$e :: A \quad \widehat{=} \quad \Box(\mathit{pid} = e) \;\wedge\; A \;.$$

The existence of a suitable *pid* then readily ensures the validity of $A \,|||\, B \;\equiv\; \exists \mathit{pid}.\, (0 :: A \;|||\; 1 :: B)$. The proof uses the validity of $A \,|||_{\mathit{pid}=0}\, B \;\equiv\; (0 :: A \;|||\; 1 :: B)$. The techniques easily generalise to any number of processes (e.g., $0 :: A_1 \;|||\; 1 :: A_2 \;|||\; 2 :: A_3$).

The rest of the imperative constructs When formalising programs and processes, the framework here takes the liberty of assuming that data variables' range over finite domains. Besides various constants such as the bit values 0 and 1, we also employ some finite sets and lists to deal with such program variables. For any given finite set of program variables, this can in principle be propositionally encoded.

Table 2 contains imperative programming constructs which can be viewed as derived operators in ITL. We let $\setminus$ denote *set difference*. Labels are optional, normally only added to each atomic assignment and noop, and do not affect program operation. When specified, *lab*'s value is the active process's current label. Labelling just the atomic statements suffices to *fully* determine *lab*'s value in all states but the final one, if the process terminates. Hence, each process ends with another labelled formula of the form $l_i \colon \mathit{empty}$.

As we already noted, interleaving-based transition systems only perform assignments involving two states adjacent in *global* time. However, a process within $|||\_$ in *projected* time might not see the *next global* state even if the *current projected* and *current global* states are identical. For example, suppose the *current global* interval is $s_1 s_2 s_3 s_4 \ldots$. Therefore, assignments from *current global* state s_1 should involve s_1 and the *next global* state s_2. If a process in $|||\_$ sees the *current projected* interval $s_1 s_4 \ldots$ without states s_2 and s_3, then any $:=$ within $|||\_$ that sees the current state s_1 cannot see *global* state s_2 and so cannot access s_2 with $\bigcirc$ to assign variables. Such an instance of $\bigcirc$ instead sees the *next projected* state s_4 (although an alternative approach *without projection* in Sect. 4.1

Table 2. Some imperative programming constructs expressed in ITL

$$
\begin{array}{rcl}
a := e & \widehat{=} & \mathit{skip} \ \wedge \ \mathit{nval}[.a] = e \\
 & & \wedge \ \forall v \in (\mathit{dom}(\mathit{nval}) \setminus \{.a\}).\, (\mathit{nval}[v] = v\hat{}) \\
a_1, \ldots, a_n := e_1, \ldots, e_n & \widehat{=} & \mathit{skip} \ \wedge \ \mathit{nval}[.a_1] = e_1 \wedge \cdots \wedge \mathit{nval}[.a_n] = e_n \\
 & & \wedge \ \forall v \in (\mathit{dom}(\mathit{nval}) \setminus \{.a_1, \ldots, .a_n\}).\, (\mathit{nval}[v] = v\hat{}) \\
\mathit{noop} & \widehat{=} & \mathit{skip} \ \wedge \ \forall v \in \mathit{dom}(\mathit{nval}).\, (\mathit{nval}[.v] = v\hat{}) \\
l_i \colon A & \widehat{=} & \mathit{lab} = l_i \ \wedge \ A \\
\mathit{empty} & & \text{(Already defined in Table 1)} \\
A; B & & \text{(Already defined as primitive ITL operator in Sect. 2)} \\
\mathit{if}\ w\ \mathit{then}\ A\ \mathit{else}\ B & \widehat{=} & (w \wedge A) \ \vee \ (\neg w \wedge B) \\
\mathit{while}\ w\ \mathit{do}\ A & \widehat{=} & (\neg w \wedge A)^*; (\mathit{empty} \wedge w) \\
\mathit{for\ some\ times\ do}\ A & \widehat{=} & A^* \\
A \sqcup B & \widehat{=} & A \vee B \qquad \text{(Nondeterministic choice)}
\end{array}
$$

can indeed see state s_2 by simply employing $\bigcirc$). *Exactly* the same issue applies to the remaining program variables which := needs to *frame* (i.e., leave unchanged) and likewise for *noop*.

The assignment construct := instead uses *state formulas* and a *state variable* *nval* which is a *record* (i.e., a finite list indexed by field names and like records in Lamport's TLA+ [32]). The purpose of *nval* is to store in the *current projected* state the values which are to be assigned to variables in the *next global* state (itself normally only accessible from *outside* of the scope of $|||\_$). In effect, *nval* helps *tunnel* from projected to global time. For each program variable a, *nval* has an element $\mathit{nval}[.a]$, where $.a$ is a *field name constant* (like a quoted atom in Lisp) serving as a subscript (TLA+ uses strings such as "a" to index records). The assignment $a := e$ *does not actually change* a or *frame* the remaining program variables (i.e., it does not explicitly keep them unchanged). Instead, in the *current projected* state (which is also the *current global* state), $a := e$ treats its first operand as a kind of *reference* (i.e., $.a$) and just sets $\mathit{nval}[.a]$ equal to e, and $\mathit{nval}[.b]$ equal to b's current value for every other (unaltered) program variable b. The desired setting of a's and b's values in the *next global* state (to equal the *current* values of $\mathit{nval}[.a]$ and $\mathit{nval}[.b]$, respectively) is handled separately outside of $|||\_$ in *global* time, as discussed later, where the operator $\bigcirc$ can indeed access the next global state.

The field name constant $.a$ can serve as a *reference* to the variable a itself because we let a be accessible via $.a$ using the *dereferencing* construct $.a\hat{}$ (e.g., the equality $.a\hat{} = a$ is valid). We can abbreviate $\mathit{nval}[.a]$ as $\mathit{nval}.a$ (as in TLA+, where $\mathit{nval}[\text{"}a\text{"}] = \mathit{nval}.a$). This shorthand is not applicable if the subscript is a variable whose value is a field name constant. For example, if b equals $.a$, then

Process Pr_0:	Process Pr_1:
$l_0 \colon x := 1;$	$l_2 \colon x := 1 - x;$
$l_1 \colon \mathit{empty}$	$l_3 \colon \mathit{empty}$

A. Let $dom(nval_{Pr}) = \{.x\}$. Initially $x = 0$.

Process Pr'_0:	Process Pr'_1:
$l'_0 \colon x := 1;$	*while* $y = 0$ *do*
$l'_1 \colon y := 1;$	$l'_3 \colon$ *noop*;
$l'_2 \colon \mathit{empty}$	$l'_4 \colon x := 1 - x;$
	$l'_5 \colon \mathit{empty}$

B. Let $dom(nval_{Pr'}) = \{.x, .y\}$. Initially both $x = 0$ and $y = 0$.

Fig. 1. Simple concurrent programs Pr and Pr'

$nval[b]$ equals both $nval[.a]$ and $nval.a$ (e.g., $\models\ b = .a\ \supset\ nval[b] = nval.a$) but not necessarily $nval.b$.

As in TLA+, we can regard the record $nval$ as a function from field name constants to values, and let $dom(nval)$ denote $nval$'s domain which is in fact the set of these field name constants. Hence, $dom(nval)$ can serve as a set of references to the program variables for use in the semantics of atomic statements (described shortly) when *framing* variables (e.g., for an assignment $a := e$, we need to explicitly formalise in the logic that all program variables referenced by $dom(nval)$ besides a remain unchanged.) For example, one concurrent program Pr' considered shortly has just two program variables x and y, so $dom(nval) = \{.x, .y\}$, where $.x$ and $.y$ are the field name constants associated with x and y, respectively. The set $dom(nval)$ especially helps to formalise framing for programs with several variables.

The framing construct *iframe* now defined, when used in *global* states, ensures that *intended* assignments of values recorded in $nval$ in each *projected* state actually *take effect* on the program variables themselves in the *next global state*:

$$\mathit{iframe} \quad \widehat{=} \quad \Box\big(\mathit{more}\ \supset\ \forall v \in dom(nval).\, (nval[v] = \bigcirc v^{\wedge})\big)\ .$$

For example, if $dom(nval) = \{.a\}$, then *iframe* is semantically equivalent to both of the formulas $\Box(\mathit{more} \supset nval[.a] = .a^{\wedge})$ and $\Box(\mathit{more} \supset nval.a = a)$.

Here are sample valid formulas involving *iframe* (assume $dom(nval) = \{.a\}$):

$$\models\quad \mathit{iframe}\ \wedge\ \Box(\mathit{more} \supset nval.a = a)\quad \supset\quad \Box(\mathit{more} \supset (\bigcirc a) = a) \qquad (8)$$

$$\models\quad \mathit{iframe}\ \wedge\ (\neg p)\ \Pi^{\mathrm{u}}\ \Box(\mathit{more} \supset nval.a = a)\ \supset\ p\ \Pi^{\mathrm{u}}\ \mathit{iframe}\ . \qquad (9)$$

According to (8), if *iframe* controls a and also $nval.a$ always equals a (except maybe at the end), then $\bigcirc a$ also always equals a (except maybe at the end), so, in other words, a is *stable*. Implication (9) describes that if *iframe* controls a and also in time projected by $\neg p$, $nval.a$ always equals a (except maybe at the end), then *iframe* as well controls a within time projected by p.

Figure 1 shows two simple concurrent programs Pr and Pr'. The next formula for Pr includes initialisation and framing (as noted in Fig. 1, $dom(nval_{Pr}) = \{.x\}$):

$$x = 0\ \wedge\ \mathit{iframe}\ \wedge\ Pr_0\ |||_r\ Pr_1\ .$$

Process $Peterson_i$, for $i \in \{0,1\}$
for some times do (
l_0: *noop*;
l_1: $flag_i := 1$;
l_2: $turn := 1$;
while($flag_{1-i} = 1 \wedge turn = 1 - i$) *do*
l_3: *noop*;
l_4: *noop*; /* Enter critical section */
l_5: *noop*; /* Critical section */
l_6: $flag_i := 0$; /* Leave critical section */
l_7: *noop*
);
l_8: *empty*

Let $dom(nval_{Peterson}) = \{.flag_0, .flag_1, .turn\}$.
Initially both $flag_0 = 0$ and $flag_1 = 0$, but *turn*'s initial value is unimportant.

Fig. 2. Version of Peterson's algorithm with processes $Peterson_0$ and $Peterson_1$

The middle operand r of $|||_-$ here need not be quantified away because we only use $|||_-$ on the left side of $\supset$. The first program can terminate with x equal to 0 or 1, but the second program ensures x ends equal to 0, as formalised below (as noted in Fig. 1, $dom(nval_{Pr'}) = \{.x, .y\}$):

$$\models \quad x = 0 \ \wedge \ y = 0 \ \wedge \ \mathit{iframe} \ \wedge \ Pr'_0 \ |||_r \ Pr'_1 \quad \supset \quad \mathit{fin}(x = 0 \wedge y = 1) \ .$$

The labels help link conditions on state to control points. Here is an example stating that x will equal 1 when process Pr'_1 is at label l'_4:

$$\models \quad x = 0 \ \wedge \ y = 0 \ \wedge \ \mathit{iframe} \ \wedge \ Pr'_0 \ |||_r \ Pr'_1 \quad \supset \quad \neg r \ \Pi \ (\mathit{lab} = l'_4 \supset x = 1) \ .$$

The next construct is a shorthand to test the current label in a process:

$$\begin{aligned} at_p \, l_i \quad &\widehat{=} \quad p \ \Pi \ (\mathit{lab} = l_i) \\ at_p \{l_{i_1}, \ldots, l_{i_k}\} \quad &\widehat{=} \quad p \ \Pi \ (\mathit{lab} \in \{l_{i_1}, \ldots, l_{i_k}\}) \ . \end{aligned}$$

The previously discussed translation of Π to LTL in Sect. 3 ensures that $at_p \, l_i$ can be expressed in LTL as $\neg p \ \mathcal{U} \ (p \ \wedge \ \mathit{lab} = l_i)$.

Figure 2 shows Peterson's mutual exclusion algorithm [42]. The two processes do not simultaneously access their critical sections (labels l_5 and l_6). Below are some valid properties, where we let *init* denote $flag_0 = 0 \wedge flag_1 = 0$ (also, as noted in Fig. 2, $dom(nval_{Peterson}) = \{.flag_0, .flag_1, .turn\}$):

$$\begin{array}{lll} \models & \mathit{init} \ \wedge \ \mathit{iframe} \ \wedge \ \mathit{Peterson}_0 \ |||_r \ \mathit{Peterson}_1 & (10) \\ & \quad \supset \quad \Box\neg(at_r\{l_5, l_6\} \ \wedge \ at_{\neg r}\{l_5, l_6\}) \\ \models & \mathit{init} \ \wedge \ \mathit{iframe} \ \wedge \ \mathit{Peterson}_0 \ |||_r \ \mathit{Peterson}_1 \\ & \quad \supset \quad \Box(at_r\, l_0 \ \supset \ \Diamond\, at_r\, l_5) \ \wedge \ \Box(at_{\neg r}\, l_0 \ \supset \ \Diamond\, at_{\neg r}\, l_5) \\ \models & \mathit{init} \ \wedge \ \mathit{iframe} \ \wedge \ (\mathit{inf} \wedge \mathit{Peterson}_0) \ |||_r \ (\mathit{inf} \wedge \mathit{Peterson}_1) \\ & \quad \supset \quad \Box\Diamond\, at_r\, l_0 \ \wedge \ \Box\Diamond\, at_r\, l_5 \ \wedge \ \Box\Diamond\, at_{\neg r}\, l_0 \ \wedge \ \Box\Diamond\, at_{\neg r}\, l_5 \\ \models & \mathit{Peterson}_i \quad \supset \quad \Box(\mathit{more} \ \supset \ \mathit{nval}.\mathit{flag}_{1-i} = \mathit{flag}_{1-i}) \\ \models & \mathit{Peterson}_i \quad \supset \quad \Box(\mathit{more} \ \supset \ \mathit{nval}.\mathit{turn} = \mathit{turn} \ \vee \ \mathit{nval}.\mathit{turn} = 1 - i) \ . \end{array}$$

Implication (10) concerns mutual exclusion. Surprisingly, variants with l_4 or $\{l_4, l_5\}$ instead of $\{l_5, l_6\}$ are *not* valid: Suppose $\mathit{Peterson}_0$'s process is active (i.e., r is true) with $\mathit{lab} = l_4$ (so $at_r\, l_4$ holds). If $\mathit{Peterson}_1$'s currently inactive process is beyond l_2 and l_3 but before l_4, it could *later on* have $\mathit{lab} = l_4$ when in its next active state entering its critical section. Then $at_{\neg r}\, l_4$ would be true now! Hence, our approach has an interesting idiom to formalise behaviour.

4.1 Formalising Interleaving without Projection

As already discussed above, modelling of interleaving with Π needs the variable *nval* and the *iframe* construct to ensure that each assignment to a program variable is suitably performed between two *globally* adjacent states. An alternative framework without Π now presented avoids the need for either *nval* or *iframe* and so is even closer to the interleaving semantics described by Baier and Katoen [1]. Instead of using a record *nval*, we simply let *pvars* denote the set of program variables' field name constants to play a role like that of $\mathit{dom}(\mathit{nval})$.

The only constructs in Table 2 which need to be changed are the assignment operator $:=$, *noop* and *empty*. Below is a definition of the alternative construct $:='$ for assigning to a single variable, where *pvars* is the set of program variable's field name constants:

$$\begin{array}{rcl} a :=' e & \widehat{=} & (\bigcirc a) = e \ \wedge \ \forall v \in \mathit{pvars} \setminus \{.a\}.\, ((\bigcirc v\hat{}) = v\hat{}) \\ & & \quad \wedge \ \mathit{active} \ \wedge \ \bigcirc\big(\mathit{finite} \wedge \Box(\mathit{more} \ \supset \ \neg\mathit{active})\big) \ . \end{array}$$

Here the operator $\bigcirc$ helps assign to a and frame other program variables over the first two (global) states. The variable *active* is initially *true*, but then *false* in the finite number of subsequent (intermediate) states, except in the last state, to indicate inactivity. Note that $:='$ does not determine *active*'s value in the last state since this is left for a follow-on atomic statement to do. Similar definitions for multiple assignments and the alternative construct noop' are omitted here. A variant of empty' ensures *active* holds in a process's final state:

$$\mathit{empty}' \quad \widehat{=} \quad \mathit{empty} \wedge \mathit{active} \ .$$

Any such process A has the valid implication $A \supset (\mathit{active} \wedge \mathit{fin}\ \mathit{active})$.

Here are variants of Π and $|||\_$ which seem suitable:

$$\begin{array}{rcl} w\ \pi\ A & \widehat{=} & \neg w\ \mathcal{U}\ \big((w \wedge A^{w}_{active}); (w \wedge \circledcirc\Box\neg w)\big) \\ A\ |||'_{w}\ B & \widehat{=} & (active \wedge w)\ \pi\ A\ \wedge\ (active \wedge \neg w)\ \pi\ B\ . \end{array}$$

The construct $w\ \pi\ A$ is similar to $w\ \Pi\ A$, but instead of projection, π uses the variable *active* to restrict A's active steps to when w holds. Properties of Π such as (2) can be adapted to π. The role of π in the definition of $|||'\_$ is similar to that of Π in the definition of $|||\_$. Variants of the properties of commutativity (5) and associativity (6)-(7) for $|||\_$ can also be shown for $|||'\_$. It is possible to formally relate programs with the projected and global constructs. Here is one possibility:

$$\models\quad iframe\ \wedge\ \Box active\quad \supset\quad Pgm_1\ \equiv\ Pgm_2\ ,$$

where Pgm_1 uses $:=$, $|||$ and so forth, which are replaced in Pgm_2 with primed versions such as $:='$ and $|||'$, and we let sets $dom(nval)$ and *pvars* be equal.

Incidentally, as a handy shorthand we can let $pvars\hat{}$ denote the record with indices in *pvars* such that for each $.a \in pvars$, the record element $pvars\hat{}[.a]$ equals a's value. For example, the equality $(\bigcirc pvars\hat{}) = pvars\hat{}$ keeps program variables' values unchanged in the next state. Also, $nval[dom(nval) \setminus \{.a\}]$ can denote the record equalling *nval* but without element $nval[.a]$.

4.2 Comparison of State Projection with Time-Step Projection

Somewhat after Π was introduced in [19], [35], another binary ITL operator was proposed in [36] (see also [37], [31]) for what can be referred to as *time-step projection*. It is alternatively written as *proj*, $\triangle$ or $\backslash\backslash$. Unlike for Π, temporal connectives almost always occur in both operands of *proj*. For finite σ,

$$\begin{array}{l} \sigma \models A\ proj\ B\ \text{ iff }\ \text{there exists } n \geq 0 \text{ and } i_0 = 0 < i_1 < \ldots < i_n = |\sigma| \text{ such that} \\ \qquad\qquad \sigma_{i_k} \ldots \sigma_{i_{k+1}} \models A,\ \text{for each } k < n,\ \text{and } \sigma_{i_0} \ldots \sigma_{i_n} \models B\ . \end{array}$$

Intuitively, A defines time steps and B is interpreted over the interval formed of the endpoints of a sequence of such steps that links the endpoints of the reference interval. The formula A^* is expressible as $A\ proj\ true$, so it expresses the mere possibility to represent the reference interval as a sequence of time steps specified by A. Note that an interval may admit more than one suitable partitioning.

The definition of *proj* generalises to infinite time by allowing an infinite number of adjacent finite subintervals. The validity of the implication

$$inf\quad \supset\quad A\ proj\ true\ \equiv\ (finite \wedge A)^*$$

shows how the operator *proj* can express *chop-omega*.

A primary application of *proj* is to define coarser time granularities, and it is included in the Tempura programming language for such purposes, whereas Π is best fit for interleaving concurrency. A variant of *proj* for projecting from real to discrete time has been studied in [20], [15].

Π and, consequently, parallel composition $|||\_$, can be expressed using *proj*:

$$\models \quad p \,\Pi\, A \quad \equiv \quad \neg p \;\mathcal{U}\; \big(p \wedge ((\bigcirc \mathit{halt}\, p)\ \mathit{proj}\ A);\, \circledcirc\Box\neg p\big) \ .$$

Conversely, *proj* can be defined using Π and propositional quantification, which, as noted in Sect. 2, does not add expressiveness to PITL:

$$\models \quad A\ \mathit{proj}\ B \quad \equiv \quad \exists p.\, (p \ \wedge\ (A \wedge \mathit{finite} \wedge \bigcirc \mathit{halt}\, p)^* \ \wedge\ p \,\Pi\, B) \ ,$$

where variable p does not occur in A or B.

5 Related Work

Projection in the Duration Calculus Dang [8] proposed for the Duration Calculus (DC) [45], [44], [41] a real-time version of the projection operator Π written / to reason about interleaving concurrency in hybrid systems. An operator for parallel composition involving global time is also defined by Dang. The definition does not use projection, although some connections to it are demonstrated. Guelev and Dang [17] further investigated this topic and other aspects of /. However, the approach does not define a simple nestable propositional three-operand concurrency operator such as $|||\_$ and $|||'\_$ (and their two-operand variants) or look at various associated valid properties presented here. A complete axiomatisation of DC with / is given in [18]. In [16], / is used to specify that pairs of corresponding flexible non-logical symbols from isomorphic predicate ITL vocabularies have the same meaning in projected subintervals. It is shown that this entails the existence of *interpolants* for implications between formulas written in the two vocabularies as in Craig's classical interpolation theorem.

Other kinds of temporal projection Several research groups have subsequently proposed and studied other forms of temporal projection [36], [37], [20] for use with ITL, DC and further variants such as Projection Temporal Logic [9–11] and RGITL [2, 3]. RGITL, which combines Jones' *Rely-Guarantee Conditions* [28] with ITL, also assumes interleaving and involves temporal projection and local time. It has concurrency operators which are akin to $|||$ but defined without using an explicit projection operator, and, as the authors acknowledge, are much more complicated to handle. RGITL has been used extensively to reasoning about interleaved concurrent programs in the KIV proof verifier. Maybe Π can help elucidate RGITL's operators.

Our new approach aims to avoid as much as possible the need to introduce new primitive temporal constructs (such as RGITL's addition of branching-time constructs) and assumptions about time. For example, reasoning in RGITL about an individual process involving both its own next step and the system's (environment's) next step uses for each program variable x two additional primed variants x' and x'' associated with these. Of course, our purist approach (both with and without a projection operator) will have some limitations (e.g., it might indeed be incompatible with RGITL's overall goals), but we would like to thoroughly research and assess the situation in future case studies and comparisons involving a range of concurrent applications.

Jones et al. observe in [29] that RGITL could perhaps be quite attractive ("seductive" in their words), although it might be *too expressive*, particularly for an unskilled person. On the other hand, recent experience by Newcombe et al. [40] at Amazon Web Services with successfully specifying and verifying subtle industrial-strength concurrent algorithms using Lamport's TLA+ [32] supports the view that logics which can equally express algorithms *and* their correctness properties are desirable, and can with care be made sufficiently accessible to significantly benefit nonspecialists. More evaluation and comparison will be needed to see whether powerful and general interval-based frameworks are overkill in relation to other approaches specifically developed for the required purposes.

Eisner et al. [13, 14] have developed LTL$^{@}$, which adds a *clock operator* to LTL to deal with time granularities in hardware systems. This is included in the international standards Property Specification Language (PSL, IEEE Standard 1850 [24]) [12] and SystemVerilog Assertions (SVA, in IEEE Standard 1800 [26]) [5]. The clock operator adds succinctness but not expressiveness and is its own dual. It requires modifying the semantics of formulas (e.g., "$\models$" includes both a state sequence and clock). The authors point out that the use of the term "projection" for the clock operator in LTL$^{@}$ and standards which adapt it is imprecise since states in between the projected ones are still accessible (unlike for Π). A similar construct called the *sampling operator* is found in *temporal 'e'* (part of IEEE Standard 1647 [25] and influenced by ITL [34], [21]).

Conclusions

We have explored new uses of the oldest known projection operator for ITL and also related it with other such constructs. In future work, we would like to apply this approach to larger concurrent applications. This research would include of an evaluation of the merits of the two approaches presented here for formalising concurrency in ITL with and without projection. Our plans also include exploring formal connections with RGITL and Projection Temporal Logic as well as clocked-based logics such as LTL$^{@}$ (all mentioned in Sect. 5).

Acknowledgments. This research was partially supported by Royal Society International Exchanges grant IE141148 and the EPSRC UNCOVER project (Ref.: EP/K001698/1). We thank Maciej Koutny and the anonymous reviewers for their comments and suggestions.

References

1. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press (2008)
2. Bäumler, S., Balser, M., Nafz, F., Reif, W., Schellhorn, G.: Interactive verification of concurrent systems using symbolic execution. AI Commun. 23(2–3), 285–307 (2010)
3. Bäumler, S., Schellhorn, G., Tofan, B., Reif, W.: Proving linearizability with temporal logic. Formal Aspects of Computing 23(1), 91–112 (2011)

4. Ben-Ari, M.: Principles of Concurrent and Distributed Programming. Addison-Wesley, second edn. (2006)
5. Cerny, E., Dudani, S., Havlicek, J., Korchemny, D.: SVA: The Power of Assertions in SystemVerilog. Springer, second edn. (2015)
6. Chellas, B.F.: Modal Logic: An Introduction. Cambridge University Press, Cambridge, England (1980)
7. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge, Massachusetts (1999)
8. Dang Van Hung: Projections: A technique for verifying real-time programs in DC. Tech. Rep. 178, UNU/IIST, Macau (1999), also in Proc. Conf. on Information Technology and Education, Ho Chi Minh City, Vietnam, January 2000
9. Duan, Z.: An Extended Interval Temporal Logic and a Framing Technique for Temporal Logic Programming. Ph.D. thesis, Dept. Comp. Sci., Newcastle University, UK (1996), `http://hdl.handle.net/10443/2075`, tech. rep. 556
10. Duan, Z., Koutny, M., Holt, C.: Projection in temporal logic programming. In: Pfenning, F. (ed.) LPAR '94. LNCS, vol. 822, pp. 333–344. Springer (1994)
11. Duan, Z., Yang, X., Koutny, M.: Framed temporal logic programming. Science of Computer Programming 70(1), 31–61 (2008)
12. Eisner, C., Fisman, D.: A Practical Introduction to PSL. Springer (2006)
13. Eisner, C., Fisman, D.: Temporal logic made practical. In: Clarke, E.M., Henzinger, T.A., Veith, H. (eds.) Handbook of Model Checking. Springer (Expected 2016), `http://www.cis.upenn.edu/~fisman/documents/EF_HBMC14.pdf`
14. Eisner, C., Fisman, D., Havlicek, J., McIsaac, A., Van Campenhout, D.: The definition of a temporal clock operator. In: Baeten, J.C., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 857–870. Springer (2003)
15. Guelev, D.P.: A complete proof system for first-order interval temporal logic with projection. J. Log. Comput. 14(2), 215–249 (2004)
16. Guelev, D.P.: Logical interpolation and projection onto state in the Duration Calculus. J. Applied Non-Classical Logics 14(1–2), 181–208 (2004)
17. Guelev, D.P., Dang Van Hung: Prefix and projection onto state in duration calculus. Electr. Notes Theor. Comput. Sci. 65(6), 101–119 (2002)
18. Guelev, D.P., Dang Van Hung: A relatively complete axiomatisation of projection onto state in the Duration Calculus. J. Applied Non-Classical Logics 14(1-2), 149–180 (2004)
19. Halpern, J., Manna, Z., Moszkowski, B.: A hardware semantics based on temporal intervals. In: Diaz, J. (ed.) ICALP 1983. LNCS, vol. 154, pp. 278–291. Springer (1983)
20. He, J.: A behavioral model for co-design. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM'99, Vol. II. LNCS, vol. 1709, pp. 1420–1438. Springer (1999)
21. Hollander, Y., Morley, M., Noy, A.: The *e* language: A fresh separation of concerns. In: Proc. TOOLS Europe 2001: 38th Int'l Conf. on Technology of Object-Oriented Languages and Systems, Components for Mobile Computing. pp. 41–50. IEEE Computer Society Press (2001)
22. Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional (2003)
23. Hughes, G.E., Cresswell, M.J.: A New Introduction to Modal Logic. Routledge, London (1996)
24. IEEE: Standard for Property Specification Language (PSL), Standard 1850-2010. ANSI/IEEE, New York (2010)
25. IEEE: Standard for the Functional Verification Language e, Standard 1647-2011. ANSI/IEEE, New York (2011)

26. IEEE: Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language, Standard 1800-2012. ANSI/IEEE, New York (2012)
27. ITL web pages. `http://www.antonio-cau.co.uk/ITL/`
28. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. 5(4), 596–619 (Oct 1983)
29. Jones, C.B., Hayes, I.J., Colvin, R.J.: Balancing expressiveness in formal approaches to concurrency. Formal Asp. Comput. 27(3), 475–497 (2015)
30. Keller, R.M.: Formal verification of parallel programs. Commun. ACM 19(7), 371–384 (1976)
31. Kröger, F., Merz, S.: Temporal Logic and State Systems. Texts in Theoretical Computer Science (An EATCS Series), Springer (2008)
32. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Professional (2002)
33. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specifications. Springer, New York (1992)
34. Morley, M.J.: Semantics of temporal $\boldsymbol{e}$. In: Melham, T.F., Moller, F.G. (eds.) *Banff'99* Higher Order Workshop: Formal Methods in Computation, Ullapool, Scotland, 9–11 Sept. 1999. pp. 138–142. Univ. Glasgow, Dept. Comp. Sci., tech. rep. (1999)
35. Moszkowski, B.: Reasoning about Digital Circuits. Ph.D. thesis, Department of Computer Science, Stanford University (Jun 1983), tech. rep. STAN–CS–83–970
36. Moszkowski, B.: Executing Temporal Logic Programs. Cambridge University Press, Cambridge, England (1986)
37. Moszkowski, B.: Compositional reasoning about projected and infinite time. In: Proc. 1st IEEE Int'l Conf. on Engineering of Complex Computer Systems (ICECCS'95). pp. 238–245. IEEE Computer Society Press (1995)
38. Moszkowski, B.: A hierarchical completeness proof for Propositional Interval Temporal Logic with finite time. J. Applied Non-Classical Logics 14(1–2), 55–104 (2004)
39. Moszkowski, B.: A complete axiom system for propositional Interval Temporal Logic with infinite time. Log. Meth. Comp. Sci. 8(3:10), 1–56 (2012)
40. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon Web Services uses formal methods. Commun. ACM 58(4), 66–73 (2015)
41. Olderog, E.R., Dierks, H.: Real-Time Systems: Formal Specification and Automatic Verification. Cambridge University Press, Cambridge, England (2008)
42. Peterson, G.L.: Myths about the mutual exclusion problem. Inf. Process. Lett. 12(3), 115–116 (1981)
43. de Roever, W.P., de Boer, F., Hanneman, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods. Cambridge University Press (2001)
44. Zhou Chaochen, Hansen, M.R.: Duration Calculus: A Formal Approach to Real-Time Systems. Springer (2004)
45. Zhou Chaochen, Hoare, C.A.R., Ravn, A.P.: A calculus of durations. Inf. Process. Lett. 40(5), 269–276 (1991)