This is an author-produced version of an article appearing in *Formal Aspects of Computing*.
Springer *Online First* publication date: 20 March 2017.
The final publication is available at Springer via
http://dx.doi.org/10.1007/s00165-017-0417-3

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
[2]Department of Algebra and Logic,
Institute of Mathematics and Informatics, Sofia, Bulgaria

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, as are stutter-invariant formulas. Striking similarities between the expressiveness of Π and the standard LTL operator $\mathcal{U}$ (*'until'*) are briefly illustrated. We also formalise concurrent imperative programming constructs with and without Π, and relate the two approaches. Peterson's mutual exclusion algorithm is used to illustrate reasoning with Π about a concrete programming example. Projection with fairness and non-fairness assumptions are both discussed. This all illustrates an approach to the analysis of such concurrent interleaving finite-state systems using temporal logic formulas with projection constructs to reason about correctness properties. Unlike conventional LTL formulas about concurrency which normally largely focus on global time, properties expressed in LTL combined with Π help to reveal and analyse important differing viewpoints involving global time and the local projected time seen by each individual process. Links between Π and another standard PITL projection operator, both suitable for reasoning about different time granularities, are demonstrated by showing the two operators to be interdefinable. We briefly look at other (mostly interval-based) temporal logics with similar forms of projection, as well as some related applications and industrial standards.

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

1. Introduction

Temporal intervals, which are finite and infinite state sequences, offer a compellingly natural and flexible way

Correspondence and offprint requests to: B. Moszkowski, School of Computing Science, Newcastle University, Newcastle upon Tyne, NE1 7RU, United Kingdom. e-mail: Ben.Moszkowski@ncl.ac.uk

to model computational processes involving hardware or software. *Interval Temporal Logic* (ITL) [Mos83, HMM83, Mos86] 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. The ITL operators *chop* (';') and *chop-star* ('*') 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 [Mos83, HMM83, MM84] 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 with and without an assumption of fairness. As later discussed, stutter invariance is an important consideration in model checking, so we examine classes of PITL formulas with and without Π which are stutter-invariant. Some sample concurrent programs, including Peterson's mutual exclusion algorithm [Pet81], are presented together with various correctness properties, including stutter-invariant ones. This all illustrates an approach to the analysis of such concurrent interleaving finite-state systems using temporal logic formulas with projection constructs to reason about correctness properties.

Unlike properties expressed in conventional point-based Linear-Time Temporal Logic (LTL) which primarily focus on *global* time, properties formulated in LTL combined with Π, denoted here as LTL+Π, can emphasise important differing viewpoints involving *global* time and the *local* projected time seen by each individual process. A projection-based framework for reasoning about the properties therefore readily permits techniques for formally relating formulas concerning the varied and interesting perspectives. Such techniques even include ways to export suitable useful formulas from the local time of a process into global time or vice versa. Moreover, formulas in LTL+Π can be easily reduced to equivalent ones in LTL, and the same applies for the reduction of formulas in PITL+Π to equivalents in PITL. Our presentation also briefly points out some intriguing similarities between the expressiveness of Π and the standard LTL operator $\mathcal{U}$ (*'until'*). In view of all of this, Π can be regarded as an important and convenient notational and mathematical gateway to largely unexplored intriguing and insightful vistas of reasoning potentially offering practical benefit, and not simply as an extra, mostly dispensable temporal construct.

Besides investigating the theory and application of projected time using the projection operator Π, we also discuss Π's interdefinability with a related ITL operator for modelling time granularities, and then look at other research on temporal projection in general. This paper is an extended version of work published in [MG15]. The expanded presentation here incorporates several improvements as well as more discussions, definitions, examples and proofs concerning Π, an operator derived from Π for interleaving concurrency, and some imperative programming constructs for use with and without Π. It also includes material on stutter invariance, non-fairness as well as support for channels formalised using a variant of concurrency permitting synchronized shared steps.

Structure of the paper: Section 2 overviews propositional ITL. Section 3 looks at the projection operator Π, presents various properties of it and shows why its addition to LTL and PITL does not alter their expressiveness. This section also addresses some issues involving stutter-invariant formulas. Section 4 uses Π to formalise concurrent programs with an assumption of fairness, and illustrates this with Peterson's algorithm. Section 5 considers some alternative approaches to reason about interleaving concurrency with PITL, such as without Π, with an assumption of non-fairness and with channels. Section 6 discusses related work. The appendix contains proofs about the correctness of the version of Peterson's algorithm presented here.

2. Propositional Interval Temporal Logic

For an in-depth presentation of PITL we refer the reader to [Mos12]; see also [Mos86, KM08] and the ITL web pages [ITL]. The version of PITL used here has the syntax

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

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).

Table 1. Some Useful Derived LTL Operators

$\textcircled{w}A \mathrel{\widehat{=}} \neg \bigcirc \neg A$	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

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:

$\sigma \models true$ 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 $\mathcal{U}$ in LTL, 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:

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

For example, $(\bigcirc\neg\bigcirc true); \neg p$ is true since σ's prefix subinterval $\sigma^0\sigma^1$ satisfies $\bigcirc\neg\bigcirc true$ (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 $(p \wedge \bigcirc\bigcirc\neg\bigcirc true)^*$ is true since σ's subintervals $\sigma^0\sigma^1\sigma^2$ and $\sigma^2\sigma^3\sigma^4$ both satisfy $p \wedge \bigcirc\bigcirc\neg\bigcirc true$, but σ does not satisfy formulas $\neg p$, $(\bigcirc\neg\bigcirc true); p$ and $true; (\neg p \wedge \neg(true; p))$.

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. Here we regard LTL as a sublogic of PITL and routinely use LTL and PITL with *both* finite and infinite intervals. Table 1 shows useful derived LTL operators. Unlike in conventional LTL which exclusively employs infinite intervals (so the formula *inf* is valid), most of the derived operators presented in Table 1 can detect whether or not an interval is finite. For example, *more* is true for any finite interval having two or more states and likewise true for all infinite intervals. Furthermore, *inf* and *finite* are not valid formulas. See [LPZ85, Eme90] for more on the expressiveness of LTL and QLTL with finite and infinite time. Here are derived unary PITL constructs *chop-plus* and *chop-omega* denoting when the operand is iterated *at least* once or using chop-omega, respectively:

$$A^+ \quad \widehat{=} \quad A; A^* \tag{2}$$

$$A^\omega \quad \widehat{=} \quad inf \ \wedge\ (finite \wedge A)^* \ . \tag{3}$$

Below are some sample valid PITL formulas:

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

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 [Mos83, Mos04, Mos12].

3. Temporal projection

The binary temporal operator Π for *state projection* [Mos83, HMM83, MM84] 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, $\sigma \models p \;\Pi\; \Box q$ holds 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 $\sigma|_B = \langle \sigma^i \colon i \leq |\sigma|,\ \sigma^{i\uparrow} \models B \rangle$ to define $\sigma \models B \;\Pi\; A$. This does not alter Π's meaning when B is a state formula. Section 5.2 employs this kind of projection.

The operator Π is said to be *existential* or *strong* because it requires its left operand to be true somewhere, so that there is at least one projected state. The *dual* $\neg(w \;\Pi\; \neg A)$ of $w \;\Pi\; A$ is analogously *universal*, denoted $w \;\Pi^{\mathrm{u}}\; A$, and can also be referred to as being *weak*. The natural distinction between the two operators is reflected in the fact that the existential temporal formula $\Diamond w$ and the dual universal temporal formula $\Box w$ can be expressed as $w \;\Pi\; \mathit{true}$ and $\neg w \;\Pi^{\mathrm{u}}\; \mathit{false}$, respectively. Of course, $\Box$ can alternatively be derived directly from $\Diamond$ (i.e., using the valid equivalence $\Box w \equiv \neg\Diamond\neg w$) in the exactly the same manner as is done in LTL (e.g., recall the earlier definition of $\Box$ in Table 1). In fact, these observations are not limited to state formulas (but perhaps easiest to understand for them) and even hold for any temporal formula A:

$$\models \Diamond A \;\equiv\; A \;\Pi\; \mathit{true} \qquad \models \Box A \;\equiv\; \neg A \;\Pi^{\mathrm{u}}\; \mathit{false} \qquad \models \Box A \;\equiv\; \neg\Diamond\neg A \;. \tag{4}$$

We let the notations LTL+Π and PITL+Π respectively denote LTL and PITL together with the operator Π having arbitrary formulas permitted even on the left-hand side (i.e., not just state formulas).

Section 4 later on shows how Π can be used to derive propositional operators for *interleaved parallel composition*. We now briefly preview this to help motivate the benefits of Π for concurrent reasoning. The *three-operand* interleaving construct $A \;|||_p\; B$ derived in Sect. 4 using Π expresses 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. It is proved there to be commutative and associative, subject to suitable manipulations of the middle operand. A closely related binary version $A \;||| B$ with the middle operand existentially hidden is also discussed. A way to express in PITL individual concurrent sequential processes to serve as such interleaving constructs' concrete operands is presented in Sect. 4.3, thereby enabling interleaving programs to be constructed and formally analysed in the notation. However, before the interleaving construct $A \;|||_p\; B$ is formally introduced, various properties of the underlying primitive projection operator Π are first discussed below to give a better idea of Π's nature and potential.

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 $\mathbf{K}$ and $\mathbf{D_c}$, and the *necessitation rule* N [HC96,Che80]. These are normally written in terms of the 'universal' weak projection operator Π^{u} defined above:

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

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

$$w \;\Pi\; (A \wedge B) \quad\equiv\quad w \;\Pi\; A \;\wedge\; w \;\Pi\; B \tag{5}$$

$$w \;\Pi^{\mathrm{u}}\; A \;\wedge\; w \;\Pi\; B \quad\supset\quad w \;\Pi\; (A \wedge B) \tag{6}$$

$$w \;\Pi^{\mathrm{u}}\; (A \;\supset\; B) \;\wedge\; w \;\Pi\; A \quad\supset\quad w \;\Pi\; B \;. \tag{7}$$

The following valid formulas are specific to Π:

$$\Box(w_1 \equiv w_2) \quad \supset \quad (w_1 \,\Pi\, A) \equiv (w_2 \,\Pi\, A) \tag{8}$$

$$w_1 \,\Pi\, (w_2 \,\Pi\, A) \quad \equiv \quad (w_1 \wedge w_2) \,\Pi\, A \tag{9}$$

$$w \,\Pi^{\mathrm{u}}\, A \;\equiv\; \Box\neg w \;\vee\; w \,\Pi\, A \qquad w \,\Pi\, A \;\equiv\; \Diamond w \;\wedge\; w \,\Pi^{\mathrm{u}}\, A \tag{10}$$

$$w_1 \,\Pi\, \Diamond w_2 \;\supset\; \Diamond w_2 \qquad \Box w_2 \;\supset\; w_1 \;\Pi^{\mathrm{u}}\, \Box w_2 \tag{11}$$

$$w \,\Pi\, A \;\equiv\; (\neg w)\; \mathcal{U}\; (w \,\Pi\, A) \tag{12}$$

$$\Box w \quad \supset \quad A \;\equiv\; (w \,\Pi\, A) \tag{13}$$

$$A \quad \equiv \quad \mathit{true} \,\Pi\, A \;. \tag{14}$$

The two equivalences in (10) 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 in (11) facilitate importing and exporting properties into and from the scope of Π. Equivalence (12) shows that Π in a sense has an implicit until-operator. Implication (13) reflects the fact that in an interval where the state formula w is true in all states, the projected interval obtained using w is identical to the original one. Hence, any formula is satisfied by the original interval iff this formula is satisfied by the projected interval. Equivalence (14) likewise shows that no formula can distinguish between the original interval and the one projected using *true* because the two intervals are identical. This equivalence's validity can be readily obtained from the validity of the LTL formula $\Box \mathit{true}$, together with the previous valid implication (13) with w taken to be *true*, and modus ponens.

The valid equivalences (15)–(17) below yield a complete axiomatisation of LTL+Π to basic LTL. On the other hand, the last two equivalences shown for the PITL operators chop and chop-star are not sound if the left operand of Π is an arbitrary formula, so we instead restrict this operand here to being a state formula w. The equivalences then indeed provide a way to reduce a formula in this subset of PITL+Π to an equivalent one in PITL.

$$A \,\Pi\, \mathit{true} \;\equiv\; \Diamond A \qquad\qquad A \,\Pi\, (B \vee C) \equiv\; A \,\Pi\, B \;\vee\; A \,\Pi\, C \tag{15}$$

$$A \,\Pi\, p \;\equiv\; (\neg A)\; \mathcal{U}\; (A \wedge p) \qquad\qquad A \,\Pi\, (B\, \mathcal{U}\, C) \equiv\; (A \,\Pi\, B)\; \mathcal{U}\; (A \,\Pi\, C) \tag{16}$$

$$A \,\Pi\, \neg B \;\equiv\; \Diamond A \;\wedge\; \neg(A \,\Pi\, B) \qquad\qquad A \,\Pi\, \bigcirc B \;\equiv\; (\neg A)\; \mathcal{U}\; \big(A \wedge \bigcirc(A \,\Pi\, B)\big) \tag{17}$$

$$w \,\Pi\, (A;B) \equiv\; (w \,\Pi\, A); (w \;\wedge\; w \,\Pi\, B) \qquad\qquad w \,\Pi\, (A^*) \;\equiv\; \neg w\; \mathcal{U}\; \big(w \wedge ((w \,\Pi\, A) \wedge \mathit{fin}\, w)^*;\, ⓦ\Box\neg w\big) \tag{18}$$

Here is a proof for the equivalence in (16) concerning $A \,\Pi\, (B\, \mathcal{U}\, C)$:

Proof. ($\supset$): Let $\tau = \sigma|_A$. Then $\sigma \models A \,\Pi\, (B\, \mathcal{U}\, C)$ means that:

$$\tau \neq \langle\rangle \qquad \tau^{k\uparrow} \models C \qquad \tau^{0\uparrow} \models B,\; \ldots,\; \tau^{k-1\uparrow} \models B, \quad \text{for some } k < |\tau|.$$

Now for some $i_0, i_1, \ldots$, we have $\tau = \sigma^{i_0}\sigma^{i_1}\ldots$. If $k > 0$ and $j \leq i_{k-1}$, then $\sigma^{j\uparrow}|_A = \sigma^{i_n\uparrow}|_A = \tau^{n\uparrow}$, where i_n is the nearest member of $\{i_0, \ldots, i_{k-1}\}$ on the right of j. Hence, $\sigma^{j\uparrow} \models A \,\Pi\, B$ for all $j \leq i_{k-1}$. Similarly, $\sigma^{i_{k-1}+1\uparrow}|_A = \tau^{k\uparrow}$, whence $\sigma^{i_{k-1}+1\uparrow} \models A \,\Pi\, C$. This entails $\sigma \models (A \,\Pi\, B)\; \mathcal{U}\; (A \,\Pi\, C)$, except for $k = 0$, in which case $\sigma|_A = \tau^{k\uparrow}$ and the satisfaction of $\mathcal{U}$ reduces to $\sigma \models A \,\Pi\, C$.

($\subset$) Suppose $\sigma \models (A \,\Pi\, B)\; \mathcal{U}\; (A \,\Pi\, C)$. Let $l \leq |\sigma|$ be such that $\sigma^{l\uparrow} \models A \,\Pi\, C$ and $\sigma^{m\uparrow} \models A \,\Pi\, B$ for all $m < l$. Then $\sigma^{l\uparrow}|_A \models C$ and $\sigma^{m\uparrow}|_A \models B$ for all $m < l$. Now $\sigma|_A \models B\, \mathcal{U}\, C$ follows because all the suffixes τ of $\sigma|_A$ which have $\sigma^{l\uparrow}|_A$ as their proper suffix have the form $\sigma^{m\uparrow}|_A$, $m < l$. □

Remark 3.1. Many until-formulas commonly occurring in applications have a state formula as the left operand, but the associated equivalence in (16) does not preserve this. For example, the rather simple formula $p \,\Pi\, (q\, \mathcal{U}\, r)$ gets ultimately transformed into the following LTL formula:

$$\big((\neg p)\; \mathcal{U}\; (p \wedge q)\big)\; \mathcal{U}\; \big((\neg p)\; \mathcal{U}\; (p \wedge r)\big) \;. \tag{19}$$

The valid equivalence below can sometimes be used instead of the one in (16) to more succinctly eliminate Π without introducing an until-formula having a non-state formula as its left operand:

$$w_1 \,\Pi\, (w_2\, \mathcal{U}\, A) \quad \equiv \quad (w_1 \supset w_2)\; \mathcal{U}\; (w_1 \,\Pi\, A) \;.$$

Hence, $p \,\Pi\, (q\, \mathcal{U}\, r)$ is equivalent to next LTL formula which likewise only has state formulas on the left of

until-formulas and is moreover shorter than (19):

$$(p \supset q)\ \mathcal{U}\ \big((\neg p)\ \mathcal{U}\ (p \wedge r)\big)\ .$$

This in turn is equivalent to the next LTL formula, which is certainly much simpler than (19):

$$(p \supset q)\ \mathcal{U}\ (p \wedge r)\ . \tag{20}$$

Demonstrating the equivalence of until-formulas (19) and (20) is not hard:

- Proof for (19) $\supset$ (20): Here is a chain of valid implications:

$$\begin{aligned}(p \supset q)\ \mathcal{U}\ \big((\neg p)\ \mathcal{U}\ (p \wedge r)\big) \quad &\supset \quad (p \supset q)\ \mathcal{U}\ \big((\neg p \vee q)\ \mathcal{U}\ (p \wedge r)\big)\\ &\supset \quad (p \supset q)\ \mathcal{U}\ \big((p \supset q)\ \mathcal{U}\ (p \wedge r)\big) \quad \supset \quad (p \supset q)\ \mathcal{U}\ (p \wedge r)\ .\end{aligned}$$

- Proof for (20) $\supset$ (19): For any formulas A and B, we have the valid implication $A \supset (B\,\mathcal{U}\,A)$, from which follows the validity of the next equivalence:

$$(p \supset q)\ \mathcal{U}\ (p \wedge r) \quad \supset \quad (p \supset q)\ \mathcal{U}\ \big((\neg p)\ \mathcal{U}\ (p \wedge r)\big)\ .$$

(End of Remark 3.1)

Below are two interesting alternatives to the valid equivalence in (18) for eliminating chop-star, and also related valid equivalences which could be optionally used to eliminate the derived PITL constructs *chop-plus* (where A^+ denotes $A; A^*$ as defined in (2)) and *chop-omega* (defined in (3)):

$$\begin{aligned}&w\ \Pi\ (A^*) \equiv \big((w\ \Pi\ (A \vee \mathit{empty})) \wedge \mathit{fin}\, w\big)^+;\ ⓦ\Box\neg w && w\ \Pi\ (A^*) \equiv (w\ \Pi\ \mathit{empty})\\ &&& \qquad \vee\ \big((w\ \Pi\ A) \wedge \mathit{fin}\, w\big)^+;\ ⓦ\Box\neg w\\ &w\ \Pi\ (A^+) \equiv \big((w\ \Pi\ A) \wedge \mathit{fin}\, w\big)^+;\ ⓦ\Box\neg w && w\ \Pi\ (A^+) \equiv \big((w\ \Pi\ A) \wedge \mathit{fin}\, w\big)^*; (w\ \Pi\ A)\\ &w\ \Pi\ (A^\omega) \equiv \big((w\ \Pi\ A) \wedge \mathit{fin}\, w\big)^\omega\ .\end{aligned}$$

All of these equivalences omit the until-formula found in (18) and have chop-plus or chop-omega (which can be viewed as a kind of chop-plus) on the right-hand side. This is so even for the first two equivalences for chop-star to ensure that w is true at least once.

Later in Sect. 5.2 we discuss how simple variants of the two equivalences in (18) are in fact sound for a useful class of Π-formulas which concern non-fairness and where the left operand is not a state formula.

By (9), $A \equiv w\ \Pi\ B$ 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 $w\ \Pi\ R$.

The equivalences in (15)–(17) for eliminating Π yield the next theorem about LTL+Π:

Theorem 3.2 (Expressiveness of LTL+Π). The logic LTL+Π is no more expressive than LTL.

As noted earlier, the equivalences in (18) for the PITL constructs chop and chop-star do not generalise to allowing the left-hand operand of Π to be an arbitrary formula. However, we can still show that PITL+Π is no more expressive than PITL in a less constructive way:

Theorem 3.3 (Expressiveness of PITL+Π). The logic PITL+Π is no more expressive than PITL.

Proof. Suppose A and B are PITL formulas. Then the Π-formula $A\ \Pi\ B$ is equivalent to the following formula in *Quantified PITL* (QPITL):

$$\exists r.\ \big(\Box(r \equiv A)\ \wedge\ r\ \Pi\ B\big)\ ,$$

where the propositional variable r does not occur in either A or B. Here, for any propositional variable p, formula C and interval σ, the quantified formula $\exists p.\, C$ has the following semantics:

$$\sigma \models \exists p.\, C \quad \text{iff} \quad \sigma' \models C, \text{ for some interval } \sigma' \text{ identical to } \sigma \text{ except possibly for } p\text{'s behaviour.} \tag{21}$$

We then use the equivalences in (15)–(18) to obtain a PITL formula C equivalent to $r\ \Pi\ B$. Hence, $A\ \Pi\ B$ is equivalent to the next QPITL formula:

$$\exists r.\ \big(\Box(r \equiv A)\ \wedge\ C\big)\ .$$

This technique can then be inductively used to transform any formula in PITL+Π with arbitrarily nested instances of Π into an equivalent QPITL formula. Now QPITL has exactly the same expressiveness for both finite- and infinite state sequences as *Quantified LTL* (QLTL) and PITL. Here is a brief summary of how to establish this:

- It is not hard to express in PITL both regular and omega-regular expressions, so PITL is at least as expressive as QLTL (see [LPZ85, Eme90] for more on the expressiveness of LTL and QLTL with finite and infinite time).
- Furthermore, any formula in PITL and even QPITL can be readily re-expressed as a semantically equivalent formula in QLTL by using existentially quantified variables to encode chop and chop-star[1].

Hence, PITL+Π, which is reducible to PITL, itself has exactly the same expressiveness as PITL, QPITL and QLTL[2]. □

We originally defined Π so that $\sigma \models w\ \Pi\ A$ *vacuously* holds when $\sigma|_w$ has no states [HMM83, Mos83, MM84]. This holds for Π^{u} in our presentation here. Projection is false when no projection interval exists for the projection operator from [Mos86, Mos95] discussed in Sect. 5.4. However, this is not the case for the real-time projection operators from [GD02, Gue04b, Gue04a].

3.1. Expressing some until-formulas using projection

We showed earlier that LTL+Π is no more expressive than LTL. Perhaps surprisingly, Π possesses some aspects of the LTL until-operator $\mathcal{U}$ so can at least to a limited degree be regarded as an alternative to it. For example, as already pointed out, both can be used to derive the temporal operators $\Diamond$ and $\Box$. Furthermore, the simple until-formula $p\,\mathcal{U}\,q$ is readily expressible using Π as demonstrated by the following valid equivalence:

$$p\,\mathcal{U}\,q \quad \equiv \quad (\neg p \vee q)\,\Pi\,q\ .$$

The projection formula here selects states satisfying either $\neg p$ or q and then ensures that the first such state satisfies q. This valid equivalence generalises to permit some arbitrary formula A in place of p:

$$A\,\mathcal{U}\,q \quad \equiv \quad (\neg A \vee q)\,\Pi\,q\ .$$

The nested until-formula $p\,\mathcal{U}\,(q\,\mathcal{U}\,r)$ can also be expressed using Π:

$$p\,\mathcal{U}\,(q\,\mathcal{U}\,r) \quad \equiv \quad \big(\neg p \vee (q\,\mathcal{U}\,r)\big)\,\Pi\,(q\,\mathcal{U}\,r)\ .$$

Further right-nesting of $\mathcal{U}$ in this manner works similarly. At least some until-formulas with negated until-formulas in the right-hand side have equivalent formulas expressed using Π. For instance, below is a valid equivalence involving the $\mathcal{U}$-formula $p\,\mathcal{U}\,\Box q$ containing the subformula $\Box q$ (itself equivalent to $\neg(\mathit{true}\,\mathcal{U}\,\neg q)$). This subformula is re-expressible without $\mathcal{U}$ by instead just using $\Diamond$ and $\Box$, both of which were previously shown in (4) to be derivable from Π:

$$p\,\mathcal{U}\,\Box q \quad \equiv \quad \Diamond\Box q \wedge \Box(\neg p \supset \Box q)\ .$$

Below is a valid equivalence for the three-variable example $p\ \mathcal{U}\ \neg(q\ \mathcal{U}\ r)$ containing a negated $\mathcal{U}$-formula and re-expressible with Π using substitution instances of the previous two equivalences for $p\,\mathcal{U}\,(q\,\mathcal{U}\,r)$ and $p\,\mathcal{U}\,\Box q$:

$$p\,\mathcal{U}\,\neg(q\,\mathcal{U}\,r) \quad \equiv \quad p\,\mathcal{U}\,\Box\neg r \ \vee\ p\,\mathcal{U}\,\big(\neg r\ \mathcal{U}\ (\neg q \wedge \neg r)\big)\ .$$

---

[1] The techniques involved for both directions were developed with J. Halpern and originally described in [Mos83] (and reproduced in [Mos04]) for PITL and QPITL with just finite time and without chop-star, but are easily extended to handle both infinite time and chop-star as well. It follows that the relationship between PITL and QPITL is quite different from that for the following pairs of logics not sharing expressiveness: (a) first-order and second-order logic and (b) LTL and QLTL.

[2] Here is how to use the quantifier $\exists$ 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 [KM08, 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: $\Box C \equiv \neg\Diamond\neg C$).

The justification for this employs the next valid LTL equivalence concerning $\neg(q \mathcal{U} r)$ together with the distributivity of $\mathcal{U}$ over a logical-or in the right-hand operand of $\mathcal{U}$:

$$\neg(q \mathcal{U} r) \quad \equiv \quad \Box\neg r \ \vee \ \neg r \ \mathcal{U} \ (\neg q \wedge \neg r) \ .$$

We are inclined at present to *rather tentatively* conjecture that the two logics LTL and LTL with Π instead of $\mathcal{U}$ have the same expressiveness (see [LPZ85, Eme90, DG08] for various alternative characterisations of the class of formal languages expressible using LTL formulas), but more study is needed. In connection with this, observe that the operator $\mathcal{U}$ permits substitution of arbitrary formulas into the scope of *both* of a valid formula's operands to obtain a valid instance of the overall formula, but Π restricts *right-hand* substitutions. For example, here is a valid formula involving $\mathcal{U}$, together with a valid substitution instance:

$$\Box\neg q \quad \supset \quad \neg(p \mathcal{U} q)$$
$$\Box\neg\Diamond r_2 \quad \supset \quad \neg((\Box r_1) \mathcal{U} \Diamond r_2) \ .$$

In contrast, Π only permits substitution of arbitrary formulas into its *left-hand* operand because the right-hand operand is evaluated in a projected subinterval, thus complicating substitution except for *state* formulas, which do not present any problem whatsoever. Here is a pair of formulas to illustrate the problem with substitution into a Π-formula's right-hand side:

$$\Box\neg q \quad \supset \quad \neg(p \ \Pi \ q)$$
$$\Box\neg \mathit{empty} \quad \supset \quad \neg(p \ \Pi \ \mathit{empty}) \ .$$

The first implication is valid because if the propositional variable q is always false, then it cannot be true at the start of some projected subinterval. However, the second implication, which is a substitution instance of the first one, is not valid because any infinite interval with p true exactly once satisfies the antecedent but falsifies the consequent. Interestingly, the valid equivalences previously presented in (4) for using Π to express the unary temporal operators $\Diamond$ and $\Box$ (e.g., $\models \Diamond A \ \equiv \ A \ \Pi \ \mathit{true}$ and $\models \Box A \ \equiv \ \neg A \ \Pi^{\mathrm{u}} \ \mathit{false}$) are not at all affected by this limitation since the operand A here only occurs in Π's *left-hand* side.

For PITL formulas, substitutions into the left-hand scope of the operator chop and into the scope of the operator chop-star's sole operand raise similar issues, yet this limitation has not precluded obtaining a complete axiom system even permitting infinite time [Mos12]. Furthermore, chop can be used instead of Π to derive a restricted but quite useful version of the operator $\mathcal{U}$ (see [Mos12, Mos13, Mos14]) which permits right-hand substitutions. For various applications of $\mathcal{U}$, the left-hand operands anyway tend to be fairly simple (i.e., they only concern the current state and perhaps the next one as well) and avoid any problems with chop's restrictions on left-hand substitution. In view of all this, reasoning in PITL+Π with $\mathcal{U}$ derived using chop could sometimes be rather unaffected by Π's restrictions on right-hand substitution, but more investigation is needed.

It is also worth noting here a curious interaction exhibited by the following equivalence concerning both $\mathcal{U}$ and Π:

$$(p \mathcal{U} q) \ \Pi \ (p \mathcal{U} q) \quad \equiv \quad \Diamond(p \mathcal{U} q) \ . \tag{22}$$

This works because if an interval satisfies the formula $p \ \mathcal{U} \ q$, then all suffixes of the interval up to and including the first state satisfying q also satisfy the formula $p \mathcal{U} q$. Therefore, whenever an interval satisfying $p \mathcal{U} q$ has its first state projected by the Π-formula here, the starting states of all of these suffixes will likewise be projected, so included as well in the local projected interval visible the projection operator's right-hand operand. For comparison, here is a related valid LTL equivalence illustrating the same kind of phenomenon:

$$(p \mathcal{U} q) \mathcal{U} q \quad \equiv \quad p \mathcal{U} q \ .$$

Note that the projection-based equivalence (22) differs from this one by possibly first skipping the some states of the global interval before finding a suffix subinterval satisfying the formula $p \mathcal{U} q$. The next variant of (22) avoids the need for the $\Diamond$ operator in (22):

$$\big(\neg p \vee (p \mathcal{U} q)\big) \ \Pi \ (p \mathcal{U} q) \quad \equiv \quad p \mathcal{U} q \ .$$

3.2. Stutter Invariance

Any LTL formula not containing instances of the operator $\bigcirc$ (i.e., in the sublogic denoted here as LTL$-\bigcirc$) is *stutter-invariant* [Lam83, Lam02], that is, for any state sequence (interval) σ, the formula is satisfied by σ iff it is satisfied by any *stutter-equivalent* variant of σ. (Some use the term *stutter-insensitive* instead of stutter-invariant.) To help with understanding the theory of Π and also with using Π in model checking, we now discuss how to extend some standard results about stutter-invariant LTL formulas to include both finite and infinite time as well as Π and chop. Some relevant and natural connections with subsets of PITL+Π having the same expressiveness as LTL are discussed as well. The material presented here helps us later establish that some formulas encountered for describing useful temporal properties are indeed stutter-invariant. Other work on the treatment of stutter invariance for frameworks which can express regular and omega-regular languages is described in Sect. 6.3.

It seems reasonable to include here some further justification for our inclusion of material on stutter invariance: Model checking often involves stutter-invariant safety properties combined with partial order reductions (e.g., in SPIN [Hol03]) or other such techniques (e.g., in the model checking tool TLC for TLA$^+$ [Lam02]). Our presentation does not discuss practical model checking. Nevertheless, stutter invariance has great relevance to projection and its practical application because, unlike with standard approaches, certain natural safety properties use the operator $\bigcirc$ and therefore are not necessarily stutter-invariant. Consequently, they would in all likelihood be much less attractive for model checking. However, some of these can in fact be shown with little difficulty to actually be stutter-invariant. Furthermore, we have found that looking for stutter invariance greatly helps assess the temporal safety properties. If the projection operator Π were to force us to have even simple safety properties be without stutter invariance, the results would have much less practical relevance, particularly for software engineers. Fortunately, projection seems compatible with stutter-invariant properties rather than clashing with them.

Here are some formal definitions of stutter equivalence and stutter invariance adapted from Peled and Wilke [PW97] for the standard kind of infinite state sequences used with LTL:

Definition 3.4 (Stutter equivalence for infinite state sequences). Two infinite state sequences σ and τ are said to be stutter-equivalent if there are two infinite sequences $0 = i_0 < i_1 < i_2 < \ldots$ and $0 = j_0 < j_1 < j_2 < \ldots$ such that for every $k \geq 0$, the states $\sigma^{i_k}, \sigma^{i_k+1}, \ldots, \sigma^{i_{k+1}-1}$ and $\tau^{j_k}, \tau^{j_k+1}, \ldots, \tau^{j_{k+1}-1}$ are all identical.

Definition 3.5 (Stutter invariance for sets of infinite state sequences). A set $S \subseteq \Sigma^\omega$ of infinite state sequences is stutter-invariant if whenever $\sigma, \tau \in \Sigma^\omega$ are stutter-equivalent, then $\sigma \in S$ iff $\tau \in S$.

It follows from this definition that a stutter-invariant set of state sequences is a union of stutter equivalence classes.

The conventional notion of stutter-invariant temporal formulas for infinite time naturally follows from stutter-invariant sets:

Definition 3.6 (Stutter-invariant formulas for infinite time). A formula said to be stutter-invariant if the set of infinite state sequences satisfying it is stutter-invariant.

Stutter-invariant formulas are important for model checkers such as SPIN [Hol03] which benefit from *Partial Order Reductions* independently developed by Godefroid and Wolper [GW91a, GW91b, GW93, God96], Peled [Pel93, Pel96] and Valmari [Val91, Val92] (see also the textbooks by Clarke et al. [CGP99] and Baier and Katoen [BK08]).

It does not seem hard to extend the definitions of stutter equivalence and stutter invariance to include finite state sequences. For our purposes, we prefer to consider finite and infinite state sequences separately, *so a finite one is never stutter-equivalent to an infinite one.* We therefore adapt Definitions 3.4 and 3.6 to also deal with finite state sequences:

Definition 3.7 (Stutter equivalence extended to finite state sequences). Two finite state sequences σ and τ are said to be stutter-equivalent if for some $n \geq 1$, there are two n-element finite sequences $0 = i_0 < i_1 < i_2 < \ldots < i_{n-1}$ and $0 = j_0 < j_1 < j_2 < \ldots < j_{n-1}$ such that both of the following hold:

- For every $k : 0 \leq k < n-1$, the states $\sigma^{i_k}, \sigma^{i_k+1}, \ldots, \sigma^{i_{k+l}-1}$ and $\tau^{j_k}, \tau^{j_{k+1}}, \ldots, \tau^{j_{k+1}-1}$ are all identical.
- The states $\sigma^{i_{n-1}}, \sigma^{i_{n-1}+1}, \ldots, \sigma^{|\sigma|-1}$ and $\tau^{j_{n-1}}, \tau^{j_{n-1}+1}, \ldots, \tau^{|\tau|-1}$ are all identical.

Definition 3.8 (Stutter-invariant formulas for both finite and infinite time). A formula said to be stutter-invariant if the set of *all* state sequences (both finite and infinite) satisfying it is stutter-invariant.

Proposition 3.9 (Formulas in LTL$-\bigcirc$ are stutter-invariant). Every formula A in LTL$-\bigcirc$ is stutter-invariant for both finite and infinite time.

Proof. We simply generalise the observation noted for just infinite time by, for example, Clarke et al. [CGP99] that this can be checked by a simple induction on the size of formulas in LTL$-\bigcirc$. □

Proposition 3.10. If A and B are formulas in LTL$-\bigcirc$, then the disjunction below is stutter-invariant:

$$(\mathit{finite} \wedge A) \vee (\mathit{inf} \wedge B) \ . \tag{23}$$

Proof. Since both A and B are in LTL$-\bigcirc$, the previous Proposition 3.9 ensures that they are stutter-invariant. A simple check then ensures that the disjunction (23) is indeed stutter-invariant as formalised in Definition 3.8. □

Theorem 3.11 (Peled and Wilke [PW97]). Every stutter-invariant formula in LTL with just infinite time has an equivalent formula in LTL$-\bigcirc$ with just infinite time.

The proof of Peled and Wilke's Theorem 3.11 does not really depend on whether time is finite or infinite, so a generalisation of the theorem extends to deal with finite time as well:

Theorem 3.12 (Variant of Peled and Wilke's Theorem 3.11 for both finite and infinite time). Every stutter-invariant formula in LTL has an equivalent formula of the following form:

$$(\mathit{finite} \wedge A) \vee (\mathit{inf} \wedge B) \ , \tag{24}$$

where subformulas A and B are in LTL$-\bigcirc$.

Observe that the simple derived LTL constructs *finite* and *inf* mentioned in Proposition 3.10 and Theorem 3.12 are themselves defined in Table 1 using the operator $\bigcirc$. The need for $\bigcirc$ here seems to be unavoidable, but we do not have a proof. As a consequence of the subformulas *finite* and *inf* being in disjunction (24), the disjunction is not in LTL$-\bigcirc$. This is not a problem for our purposes. Of course, if we restrict time to being just finite or just infinite, such an issue does not arise.

Note that *finite*, *inf*, and any formula *fin* w are stutter-invariant, as the following valid equivalences show:

$$\begin{array}{lcl} \mathit{finite} & \equiv & (\mathit{finite} \wedge \mathit{true}) \vee (\mathit{inf} \wedge \mathit{false}) \\ \mathit{inf} & \equiv & (\mathit{finite} \wedge \mathit{false}) \vee (\mathit{inf} \wedge \mathit{true}) \\ \mathit{fin}\, w & \equiv & (\mathit{finite} \wedge \Diamond\Box w) \vee (\mathit{inf} \wedge \mathit{true}) \ . \end{array}$$

If we were to sometimes let a finite state sequence be stutter-equivalent to an infinite one, then none of these three formulas would be regarded as stutter-invariant. First of all, *finite* and *inf* would not be because any finite state sequence could be stretched to match up with some infinite one. Furthermore, instances of the construct *fin* w could fail to be stutter-invariant since *inf* can be expressed as *fin false*. This seems to us too restrictive because some contexts in fact only involve finite state sequences or naturally differentiate between the two kinds.

Any formula A in LTL$-\bigcirc$ is stutter-invariant for both finite and infinite intervals, so can be expressed as the disjunction $(\mathit{finite} \wedge A) \vee (\mathit{inf} \wedge A)$. Therefore, the use of such disjunctions to distinguish behaviour in finite and infinite time does not actually restrict working with formulas already in LTL$-\bigcirc$. Rather, these disjunctions help to broaden the range of behaviour considered stutter-invariant for our purposes.

Proposition 3.13. Every stutter-invariant formula in LTL+Π has an equivalent LTL formula of the form $(\mathit{finite} \wedge B) \vee (\mathit{inf} \wedge C)$ for some formulas B and C both in LTL$-\bigcirc$.

Proof. The proposition follows by immediate application of Theorems 3.2 and 3.12. □

Furthermore, we have the following proposition:

Proposition 3.14. If formulas A and B in PITL+Π are stutter-invariant, then so is the Π-formula $A \,\Pi\, B$.

Proof. We first consider the case when the left operand of $A \,\Pi\, B$ is a state formula w. Now w is in LTL$-\bigcirc$ so stutter-invariant. Suppose B is likewise stutter-invariant, but $w \,\Pi\, B$ is not. Then there are two stutter-equivalent state sequences σ and σ', both finite or both infinite, with $\sigma \models w \,\Pi\, B$ and $\sigma' \not\models w \,\Pi\, B$. It follows

that w is true in some state of σ and hence also in σ' because it is stutter-equivalent to σ. Therefore the projected state sequences $\sigma|_w$ and $\sigma'|_w$ each have at least one state. Furthermore, they are stutter-equivalent because only the state formula w, which is stutter-invariant, is used to project out their states from the stutter-equivalent state sequences σ and σ'. However, we have from $\sigma \models w \;\Pi\; B$ and $\sigma' \not\models w \;\Pi\; B$ that $\sigma|_w \models B$ and $\sigma'|_w \not\models B$ both hold. This contradicts the assumption that the formula B is stutter-invariant.

We now generalise the proof to handle an arbitrary formula A: Suppose A and B are stutter-invariant, but $A \;\Pi\; B$ is not. Hence, there are stutter-equivalent state sequences σ and σ', both finite or both infinite, with $\sigma \models A \;\Pi\; B$, but $\sigma' \not\models A \;\Pi\; B$. Let p be some propositional variable not occurring in either A or B. The earlier proof of stutter invariance for any formula $w \;\Pi\; B$ ensures that $p \;\Pi\; B$ has this property. Construct the state sequence τ from σ to set p's value in each state to that of A in the associated suffix state sequence starting from there. Let the state sequence τ' likewise be obtained from σ'. We then have the following hold for the formula $\Box(p \equiv A) \wedge (p \;\Pi\; B)$:

$$\tau \models \Box(p \equiv A) \;\wedge\; (p \;\Pi\; B) \qquad\qquad \tau' \not\models \Box(p \equiv A) \;\wedge\; (p \;\Pi\; B) \;\; .$$

If A is stutter-invariant, it readily follows that so is $\Box(p \equiv A)$ by reasoning similar to that for proving Proposition 3.9 but with the formula A treated like a propositional variable. In addition, we already have from the proof's first case that $p \;\Pi\; B$ is stutter-invariant. These together contradict the formula $\Box(p \equiv A) \;\wedge\; (p \;\Pi\; B)$ being stutter-invariant. □

Here is a quite different proof of Proposition 3.14 for when A and B are in LTL+Π. While not essential, it is presented here to nicely illustrate some interesting reasoning using equivalences in (15)–(17) for eliminating Π from a formula:

Proof. As already established by Theorem 3.2, the operator Π does not add expressiveness to LTL. Let us therefore assume without loss of generality that A and B are themselves stutter-invariant LTL formulas not containing any instances of Π. By Theorem 3.12, we have the following two valid semantic equivalences:

$$\begin{aligned} A \quad &\equiv \quad (\mathit{finite} \;\wedge\; A'_1) \;\vee\; (\mathit{inf} \;\wedge\; A'_2) \\ B \quad &\equiv \quad (\mathit{finite} \;\wedge\; B'_1) \;\vee\; (\mathit{inf} \;\wedge\; B'_2) \;\; , \end{aligned}$$

where subformulas A'_1, A'_2, B'_1 and B'_2 are all in LTL−○. Consequently, the Π-formula $A \;\Pi\; B$ can be expressed as follows:

$$\big((\mathit{finite} \;\wedge\; A'_1) \;\vee\; (\mathit{inf} \;\wedge\; A'_2)\big) \;\Pi\; \big((\mathit{finite} \;\wedge\; B'_1) \;\vee\; (\mathit{inf} \;\wedge\; B'_2)\big) \;\; . \tag{25}$$

The semantics of Π then lets us transform this into a disjunction of three cases distinguishing between when the global and projected intervals have finite or infinite time:

$$\big(\mathit{finite} \;\wedge\; (A'_1 \;\Pi\; B'_1)\big) \;\vee\; \big(\mathit{inf} \;\wedge\; (A'_2 \;\Pi\; (\mathit{finite} \;\wedge\; B'_1))\big) \;\vee\; \big(\mathit{inf} \;\wedge\; (A'_2 \;\Pi\; (\mathit{inf} \;\wedge\; B'_2))\big) \;\; . \tag{26}$$

There are only three such cases because if the global interval is finite, then the projected one cannot be infinite. Let us now consider each case in turn:

1. $\mathit{finite} \;\wedge\; (A'_1 \;\Pi\; B'_1)$: Both A'_1 and B'_1 are in LTL−○, so the equivalences in (15)–(17) for eliminating Π yield from the Π-formula $A'_1 \;\Pi\; B'_1$ some equivalent formula C_1 in LTL−○.
2. $\mathit{inf} \;\wedge\; (A'_2 \;\Pi\; (\mathit{finite} \;\wedge\; B'_1))$: A simple generalisation of valid equivalence (5) for when the left side of Π is an arbitrary formula rather then just a state formula helps us to express $A'_2 \;\Pi\; (\mathit{finite} \;\wedge\; B'_1)$ as a conjunction:

   $$\models \quad A'_2 \;\Pi\; (\mathit{finite} \;\wedge\; B'_1) \quad \equiv \quad (A'_2 \;\Pi\; \mathit{finite}) \;\wedge\; (A'_2 \;\Pi\; B'_1) \;\; .$$

   In infinite state sequences the left conjunct $A'_2 \;\Pi\; \mathit{finite}$ is equivalent to a conjunction in LTL−○:

   $$\models \quad \mathit{inf} \quad \supset \quad A'_2 \;\Pi\; \mathit{finite} \;\equiv\; (\Diamond A'_2 \;\wedge\; \Diamond\Box\neg A'_2) \;\; .$$

   We then use equivalences in (15)–(17) for eliminating Π from the Π-formula $A'_2 \;\Pi\; B'_1$ to obtain a formula C_2 in LTL−○. Therefore we have the following valid equivalence for the second case involving $\mathit{inf} \;\wedge\; (A'_2 \;\Pi\; (\mathit{finite} \;\wedge\; B'_1))$:

   $$\big(\mathit{inf} \;\wedge\; (A'_2 \;\Pi\; (\mathit{finite} \;\wedge\; B'_1))\big) \quad \equiv \quad \big(\mathit{inf} \;\wedge\; \Diamond A'_2 \;\wedge\; \Diamond\Box\neg A'_2 \;\wedge\; C_2\big) \;\; .$$

3. $\mathit{inf} \wedge (A'_2 \mathbin{\Pi} (\mathit{inf} \wedge B'_2))$: This is similar to the second case. We obtain the next equivalent formula:

$$\big(\mathit{inf} \wedge (A'_2 \mathbin{\Pi} (\mathit{inf} \wedge B'_2))\big) \quad \equiv \quad \big(\mathit{inf} \wedge \Box\Diamond A'_2 \wedge C_3\big) \ ,$$

where C_3 is a Π-free formula equivalent to $A'_2 \mathbin{\Pi} B'_2$ and in LTL$-\bigcirc$.

The results of the three cases are now combined to conclude the next equivalence's validity:

$$A \mathbin{\Pi} B \quad \equiv \quad (\mathit{finite} \wedge C_1) \vee \big(\mathit{inf} \wedge ((\Diamond A'_2 \wedge \Diamond\Box\neg A'_2 \wedge C_2) \vee (\Box\Diamond A'_2 \wedge C_3))\big) \ .$$

The right-hand operands in both disjuncts are in LTL$-\bigcirc$, so the overall disjunction is stutter-invariant by Proposition 3.10. Hence, the original formula $A \mathbin{\Pi} B$ is as well. □

If two formulas A and B are stutter-invariant, it readily follows that so is their sequential composition $A; B$. Hence, formulas built from LTL$-\bigcirc$ with chop (i.e., LTL$-\bigcirc$+chop) are stutter-invariant. The next proposition concerns stutter invariance, LTL+chop and finite time (and is extendable to infinite time).

Proposition 3.15. In finite time, any stutter-invariant formula in LTL+chop is equivalent to one in LTL$-\bigcirc$.

Proof. It is known that LTL with finite time expresses exactly *star-free languages* [LPZ85, Eme90, DG08], which are a proper subset of the regular languages with the operations of concatenation, union and complementation but not Kleene star. Star-free languages, also known as *first-order definable languages*, readily correspond to the sublogic of PITL with just the temporal operators $\bigcirc$ and chop, so this sublogic has the same expressiveness as LTL. The addition of the operator $\mathcal{U}$ to obtain LTL+chop does not increase expressiveness because for any formula in LTL+chop, all subformulas with chop can be inductively replaced by equivalent LTL formulas. Therefore, any formula A in LTL+chop has an equivalent LTL formula A'. If A is stutter-invariant, then so is A'. Hence, our adaptation for finite time of Peled and Wilke's theorem in [PW97] about LTL ensures the existence of a formula A'' in LTL$-\bigcirc$ equivalent to A' and hence also to A. □

On the other hand, a PITL formula without $\bigcirc$ but with chop-star might not be stutter-invariant. For example, the formula false^*, which is equivalent to *empty*, is true exactly on one-state intervals, which can therefore not be 'stretched' even though the subformula *false* is clearly stutter-invariant. However, it is straightforward to show for any stutter-invariant formula A that the derived unary PITL operator A^+ ('A *chop-plus*') previously defined in (2) is likewise stutter-invariant.

4. Formalisation of imperative concurrent programs

We now look at a way to formalise in ITL imperative concurrent programs in which processes are interleaved using the operator Π as the basis. 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 [Mos86]. Such an interpreter for an ITL programming language subset called *Tempura* is available from [ITL]. ITL has also been productively used for *symbolic execution* for theorem proving [BBN+10, BSTR11].

Our presentation describes a self-contained decidable logical framework for expressing and reasoning about both interleaved finite-state programs, abstracted skeletal versions of their control structure, and associated correctness properties. This enables a demonstration of the framework by means of properties later considered which concern basic safety and liveness issues. Others relate global and local time in order, for example, to suitably import a state formula concerning a program variable's initialisation in the initial *global* state into the possibly much later *local* initial state of an individual process, or to export some properties about the *local* intervals of one or more interleaved processes into the common *global* interval.

The approach described here is specifically meant to correspond to the popular notion of *state transition systems* (based on Keller's work [Kel76] and extensively surveyed by Baier and Katoen [BK08]; see also [CGP99, KM08]), where at any instant 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* [MP92] (see also [BA06, KM08]), Jones' *Rely-Guarantee Conditions* [Jon83] (see also [dRdBH+01]), the SPIN model checker [Hol03] and *Partial Order Reduction* (see Sect. 3.2 above) used by some model checkers such as SPIN. Furthermore, Lamport's TLA+ [Lam02] (including the TLC model checker) is often used with an assumption of interleaving (but

supports non-interleaving as well). 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. This is because interleaving programming constructs can be defined *without* them, as discussed later in Sect. 5.1. Nevertheless, we consider them here because they bring succinctness and clarity and also have interesting mathematical properties. The projection operator Π also helps to formally bridge and compare the different possible notational and semantics perspectives. Some later research by others on expressing concurrent programs in variants of ITL is discussed in Sect. 6.

4.1. Interleaved parallel composition

We now define an operator 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:

$$A \,|||_p\, B \quad \widehat{=} \quad p \,\Pi\, A \;\wedge\; (\neg p) \,\Pi\, B \;. \tag{27}$$

We refer to this *three-operand* interleaving operator as $|||\_$. It is commutative and associative, subject to suitable manipulations of the middle operand:

$$\models \quad A \,|||_p\, B \quad \equiv \quad B \,|||_{\neg p}\, A \tag{28}$$

$$\models \quad (A \,|||_p\, B) \,|||_q\, C \quad \equiv \quad A \,|||_{p \wedge q}\, (B \,|||_q\, C) \tag{29}$$

$$\models \quad A \,|||_p\, (B \,|||_q\, C) \quad \equiv \quad (A \,|||_p\, B) \,|||_{p \vee q}\, C \tag{30}$$

Commutativity is easily proved, and so is associativity using the validity of

$$p \,\Pi\, (A \,|||_q\, B) \quad \equiv \quad (p \wedge q) \,\Pi\, A \;\wedge\; (p \wedge \neg q) \,\Pi\, B \;. \tag{31}$$

Proof of (31). Here is a chain of equivalences:

$$\begin{array}{lll} p \,\Pi\, \big(q \,\Pi\, A \;\wedge\; (\neg q) \,\Pi\, B\big) & & \text{Def. of } p \,\Pi\, (A \,|||_q\, B) \\ \quad \equiv \quad p \,\Pi\, \big(q \,\Pi\, A\big) \;\wedge\; p \,\Pi\big((\neg q) \,\Pi\, B\big) & & (5) \\ \quad \equiv \quad (p \wedge q) \,\Pi\, A \;\wedge\; (p \wedge \neg q) \,\Pi\, B & & (9) \end{array}$$

□

Proof of commutativity (28). Here is a chain of equivalences:

$$\begin{array}{lll} p \,\Pi\, A \;\wedge\; (\neg p) \,\Pi\, B & & \text{Def. of } A \,|||_p\, B \\ \quad \equiv \quad (\neg p) \,\Pi\, B \;\wedge\; p \,\Pi\, A & & \text{Prop.} \\ \quad \equiv \quad (\neg p) \,\Pi\, B \;\wedge\; (\neg\neg p) \,\Pi\, A & & \text{Prop.} \\ \quad \equiv \quad B \,|||_{\neg p}\, A & & \text{Def. of } |||\_ \end{array}$$

□

Proofs of associativity for (29) *and* (30). We first use a chain of equivalences to demonstrate the validity of (29):

$$\begin{array}{lll} q \,\Pi\, (p \,\Pi\, A \wedge \neg p \,\Pi\, B) \;\wedge\; \neg q \,\Pi\, C & & \text{Def. of } (A \,|||_p\, B) \,|||_q\, C \\ \quad \equiv \quad (q \wedge p) \,\Pi\, A \;\wedge\; (q \wedge \neg p) \,\Pi\, B \;\wedge\; \neg q \,\Pi\, C & & (31) \\ \quad \equiv \quad (p \wedge q) \,\Pi\, A \;\wedge\; (\neg(p \wedge q) \wedge q) \,\Pi\, B & & \text{Prop.} \\ \qquad \wedge \;\; (\neg(p \wedge q) \wedge \neg q) \,\Pi\, C & & \\ \quad \equiv \quad (p \wedge q) \,\Pi\, A \;\wedge\; \neg(p \wedge q) \,\Pi\, (B \,|||_q\, C) & & (31) \\ \equiv \quad A \,|||_{p \wedge q}\, (B \,|||_q\, C) & & \text{Def. of } |||\_ \end{array}$$

The validity of (30) can then be established from that of (29):

$$\begin{array}{lll} A \,|||_p\, (B \,|||_q\, C) & & \\ \quad \equiv \quad (C \,|||_{\neg q}\, B) \,|||_{\neg p}\, A & & (28) \\ \quad \equiv \quad C \,|||_{\neg q \wedge \neg p}\, (B \,|||_{\neg p}\, A) & & (29) \\ \quad \equiv \quad C \,|||_{\neg(p \vee q)}\, (B \,|||_{\neg p}\, A) & & \text{Prop.} \\ \quad \equiv \quad (A \,|||_p\, B) \,|||_{p \vee q}\, C & & (28) \end{array}$$

□

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

$$A \,|||\, B \quad \widehat{=} \quad \exists p.\, (A \,|||_p\, B) \ . \tag{32}$$

Recall the semantics of existential quantification given in (21) and that it does not add expressiveness to PITL. The notation $|||$ here for interleaving concurrency follows Baier and Katoen's in [BK08]. With the middle operand quantified away, $|||$ is commutative and associative in the usual sense. Both Π and $|||\_$ are expressible using either $|||$ or $|||\_$:

$$\models \quad A \,|||_w\, B \quad \equiv \quad (\Box w \wedge A) \,|||\, (\Box\neg w \wedge B) \qquad\qquad \models \quad w \,\Pi\, A \quad \equiv \quad (A \,|||_w\, \mathit{true}) \ \vee \ (\Box w \wedge A) \ .$$

Hence, at least theoretically, either operator $|||$ or $|||\_$ can be taken as primitive instead of Π for applications where Π's left operand is restricted to being a state formula. The equivalence for expressing $w \,\Pi\, A$ needs two cases because, unlike $w \,\Pi\, A$, the disjunct $A \,|||_w\, \mathit{true}$ entails that w is sometimes false.

It might at first seem that when one operand is *false*, the operator $|||\_$ can be readily eliminated. However, closer examination reveals that this is not so. The equivalence below is therefore not valid:

$$A \,|||_p\, \mathit{false} \quad \equiv \quad \Box p \ \wedge \ A \ .$$

A key reason for this is that $|||\_$ always requires each operand to be satisfied:

$$\models \quad \neg(A \,|||_p\, \mathit{false}) \qquad \models \quad \neg(\mathit{false} \,|||_p\, B) \ .$$

Therefore, an interval in which an instance of $|||\_$ is true contains at least *two* states — one for each operand. This is expressed by the next two valid implications:

$$\models \quad A \,|||_p\, B \quad \supset \quad \mathit{more} \qquad\qquad \models \quad A \,|||_p\, B \quad \supset \quad \Diamond p \ \wedge \ \Diamond\neg p \ . \tag{33}$$

Consequently, when an interval σ satisfies the formula $A \,|||_p\, B$, the semantic evaluation of operands A and B is performed in projected subintervals which omit some of σ's states. Even an apparently simple formula such as $\mathit{empty} \,|||_p\, \mathit{empty}$ requires two states. This behaviour can be confusing because of the formula's similarity in syntax to the chop-formula $\mathit{empty}; \mathit{empty}$, which in contrast only needs one state. Furthermore, unlike the chop-formulas $A; \mathit{empty}$ and $\mathit{empty}; A$ which are both equivalent to A, the formula $A \,|||\, \mathit{empty}$ requires an *extra* state to satisfy the subformula *empty*! Here are several valid equivalences illustrating differences between the constructs chop, $|||\_$ and $|||$:

$$\begin{array}{lll}
\models \quad \mathit{empty}; \mathit{empty} \quad \equiv \quad \mathit{empty} & \models \quad \mathit{true}; \mathit{true} \quad \equiv \quad \mathit{true} \\
\models \quad \mathit{empty} \,|||_p\, \mathit{empty} \quad \equiv \quad \mathit{skip} \ \wedge \ (p \not\equiv \bigcirc p) & \models \quad \mathit{true} \,|||_p\, \mathit{true} \quad \equiv \quad \mathit{more} \ \wedge \ \Diamond p \ \wedge \ \Diamond\neg p \\
\models \quad \mathit{empty} \,|||\, \mathit{empty} \quad \equiv \quad \mathit{skip} & \models \quad \mathit{true} \,|||\, \mathit{true} \quad \equiv \quad \mathit{more} \ .
\end{array}$$

The three-operand interleaving operator $|||\_$ is derived from Π without using $\bigcirc$, so all formulas in LTL$-\bigcirc$+chop+Π+$|||\_$ are stutter-invariant. However, formulas containing the derived binary operator $|||$ are not necessarily stutter-invariant. For example, as noted shortly before, $\mathit{true} \,|||\, \mathit{true}$ is equivalent to *more* and $\neg\mathit{empty}$, which are not stutter-invariant.

4.2. 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:

$$e :: A \quad \widehat{=} \quad \Box(\mathit{pid} = e) \ \wedge \ A \ . \tag{34}$$

The existence of a suitable *pid* then readily ensures the validity of the next formula:

$$A \,|||\, B \quad \equiv \quad \exists \mathit{pid}.\, (0 :: A \ |||\ 1 :: B) \ .$$

The proof uses the validity of the formula below:

$$A \,|||_{\mathit{pid}=0}\, B \quad \equiv \quad (0 :: A \ |||\ 1 :: B) \ .$$

Table 2. Some imperative programming constructs expressed in ITL

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

Note: Only atomic statements :=, *noop* and *enoop* are labelled (but without multiple labels). Also, *enoop* is normally only used at the end of a process.

The techniques easily generalise to any number of processes (e.g., $0 :: A_1 \ ||| \ 1 :: A_2 \ ||| \ 2 :: A_3$). Note that the projection formula $(pid = i) \ \Pi \ (i :: A)$ is equivalent to the simpler one $(pid = i) \ \Pi \ A$.

4.3. The rest of the imperative constructs

So far, the presentation has been largely propositional in nature. 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 program variables. For any given finite set of program variables, this can in principle still be propositionally encoded. Indeed, we adapted a similar approach in earlier work such as [Mos00] which represents finite domains in PITL to obtain axiomatic completeness for quantified ITL with finite time and such domains. Therefore, the programs presented here and the associate formal reasoning concern finite-state systems. The overall framework can be regarded as being built on *propositional logical foundations* which can avoid a truly first-order temporal logic, thereby staying suitable for decision procedures, model checking and complete axiomatisations[3]. Indeed, our aim to support the model checking of concurrent algorithms' correctness properties, with both the algorithms and the properties expressed within our decidable framework, is the primary motivation and justification for emphasising formulas with stutter invariance previously in Sect. 3.2. Of course, the framework here could also be employed in a genuinely first-order way which, when possible, takes maximum advantage of natural holistic connections to the much simpler propositional variant, as is routinely done in the theory and application of conventional propositional and first-order logics.

For any expression e, let the temporal construct $\bigcirc e$ be a term denoting e's value in the next state if there is one (e.g., as in the formula $e \neq \bigcirc e$). Table 2 contains imperative programming constructs which can be viewed as derived operators in ITL. We let $\setminus$ denote *set difference*.

Labels are special constants. For any $i \neq j$, the label constants l_i and l_2 are distinct (i.e., $l_i \neq l_j$). They are optional, normally only added to each atomic statement such as 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, we adapt the convention that each process normally ends with another labelled formula of the form $l_i \colon enoop$, where '*enoop*' stands for *'empty noop'*. This construct also helps with framing at the end of a process and is further explained later in Sect. 4.5. Even if the labels are omitted, each process should contain *enoop* at its end for framing. Only one label constant should be used per atomic statement because multiple distinct label constants as in the statement $l_0 \colon l_1 \colon noop$ clash since the control variable *lab* cannot simultaneously equal two different values (e.g., $\models \neg(lab = l_0 \wedge lab = l_1)$).

It is important to note that the label constants employed here are just intended for reasoning about densely labelled, relatively abstract concurrent programs. Other researchers such as Manna and Pnueli [MP92] (see also Ben-Ari [BA06]), Kröger and Merz [KM08] and Taubenfeld [Tau06] likewise extensively label such

[3] In contrast, unlike standard first-order logic, which is undecidable but at least has a complete axiomatisation, first-order LTL temporal logic lacks *both* of these useful properties (see [KM08] for more details and references).

concurrent programs when analysing them. On the other hand, our framework is not suited for formalising a semantics of general-purpose programming languages permitting optional labelling as well as gotos.

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 though 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 instance of $:=$ within $|||\_$ that sees the current state s_1 cannot see *global* state s_2 and so cannot access s_2 with $\bigcirc$ to assign program variables. Such an instance of $\bigcirc$ instead sees the *next projected* state s_4 (although an alternative approach *without projection* in Sect. 5.1 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$^+$ [Lam02]). The purpose of *nval* is to store in the current *projected* state the values which are to be assigned to program 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 $nval[.a]$, 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 $nval[.a]$ equal to e, and $nval[.b]$ 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 $nval[.a]$ and $nval[.b]$, respectively) is handled separately outside of $|||\_$ in *global* time, as discussed later, where the operator $\bigcirc$ can indeed access the next global state.

We employ the notation a' here as an abbreviation for the record element $nval[.a]$. A program variable's value in next global state is not readily accessible in projected time, but this element always is. In contrast to here, TLA$^+$ uses the primed variable a' to refer the value of a variable a in the next *global* state (local states are not used). The field-name constant $.a$ also serves 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).

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, themselves possibly being aggregates such as vectors.

It is important to note that the second version of $:=$ in Table 2 has the form $a_1, \ldots, a_n := e_1, \ldots, e_n$ for *simultaneously* multiple assignments to several variables $a_1, \ldots, a_n$ and requires precisely 2 states (just as a simple assignment $a := e$ does). This is because the semantics of $a_1, \ldots, a_n := e_1, \ldots, e_n$ is defined to operated on all of the variables $a_1, \ldots, a_n$ *at once* by setting $nval[.a_i] = e_i$ in the current state for each $i : 1 \leq i \leq n$. Hence, if $n \neq 1$, then the multiple assignment is *not* semantically equivalent to n successive assignments $a_1 := e_1; \ldots; a_n := e_n$ sequentially done one after the other and needing altogether exactly $n+1$ states.

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*:

$$iframe \quad \widehat{=} \quad \Box\big(more \supset \forall v \in dom(nval).\, (nval[v] = \bigcirc v\hat{})\big) \ . \tag{35}$$

For example, if $dom(nval) = \{.a\}$, then *iframe* is equivalent the following two formulas:

$$\Box(more \supset nval[.a] = \bigcirc .a\hat{}) \qquad \Box(more \supset a' = \bigcirc a) \ .$$

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

$$\models \quad iframe \ \wedge \ \Box(more \ \supset \ a' = a) \quad \supset \quad \Box(more \ \supset \ (\bigcirc a) = a) \tag{36}$$

$$\models \quad iframe \ \wedge \ (\neg p) \ \Pi^{\mathrm{u}} \ \Box(more \ \supset \ a' = a) \quad \supset \quad p \ \Pi^{\mathrm{u}} \ iframe \ . \tag{37}$$

According to (36), if *iframe* controls a, and if also a' (i.e., $nval[.a]$) 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 (37) describes that if *iframe* controls a and also in time projected by $\neg p$ that a' always equals a (except maybe at the end), then *iframe* as well controls a within time projected by p.

The definition of *iframe* can be extended to permit optional parameters limiting its effect to specific references when other ones are irrelevant:

$$iframe(.a_1, \ldots, .a_n) \quad \widehat{=} \quad \Box\big(more \ \supset \ \forall v \in \{.a_1, \ldots, .a_n\}. \, (nval[v] = \bigcirc v\hat{}\,)\big)$$

$$iframe(u) \quad \widehat{=} \quad \Box\big(more \ \supset \ \forall v \in u). \, (nval[v] = \bigcirc v\hat{}\,)\big) \ ,$$

for any references $.a_1, \ldots, .a_n \in dom(nval)$ and set $u \subseteq dom(nval)$. For example, both $iframe(.a, .b)$ and $iframe(\{.a, .b\})$ only concern the programs variables a and b. Here is a variant of the valid implication (37), but now only involving the variable a:

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

4.4. Fair scheduling of atomic statements

The semantics of the programming constructs so far presented assumes that individual assignment and *noop* constructs always take a finite amount of time. Scheduling of processes is *fair* in the sense that a process is never denied a chance to execute its next atomic statement. The interleaving formula $A \ |||_p \ B$ is fair by definition because for any state sequence σ satisfying it, the projected state sequences $\sigma|_p$ and $\sigma|_{\neg p}$ each must accommodate a complete run of $|||_p$'s respective operand.

4.5. Justification for the empty noop construct '*enoop*'

As noted above, the construct *enoop* ensures that *nval* is defined in the final state of a process if the process terminates. This helps to frame program variables. Another purpose of *enoop* is to serve as a placeholder for a label in the process's last state.

Let us now consider how *enoop* helps with framing involving the interleaving operator $|||\_$. As we pointed out earlier, an instance of $|||\_$ requires at least two states (e.g., see the valid implication (33) above). Consequently, even the simplest two-state interleaved program has to deal with framing variables from its own *first* state to its *second* one, even if no label constants are used. For this reason, such a two-state interleaved program with label constants should have the form shown below for some i, j with $i \neq j$:

$$l_i\colon enoop \ |||_p \ l_j\colon enoop \ .$$

The necessary framing information provided by *enoop* between the two projected subintervals' states is straightforward and indeed defined exactly in the same way as the framing done with *noop* (as can be seen from the respective definitions in Table 2). If we incorrectly use *empty* instead *enoop*, as in the following program, then the essential framing information for all program variables from the first state to the second one is missing:

$$l_i\colon empty \ |||_p \ l_j\colon empty \ .$$

It also appears to us that the presence of *enoop* at the end a process has a further significant benefit of helping to make some temporal properties stutter-invariant. This is because without *enoop*, the behaviour of *nval* in the last state would be unspecified, and so an instance of the operator $\bigcirc$ (in the form of *more*) would be required to ignore that state. Thus, the presence of *enoop* sometimes helps avoid the need to distinguish in temporal properties between the last state and earlier ones.

Figure 1 shows two simple concurrent programs Pr and Pr'. The next formula for Pr includes initialisation

Process Pr_0:	Process Pr_1:	Process Pr'_0:	Process Pr'_1:
l_0: $x := 1;$	l_2: $x := 1 - x;$	l'_0: $x := 1;$	*while* $y = 0$ *do*
l_1: *enoop*	l_3: *enoop*	l'_1: $y := 1;$	l'_3: *noop*;
		l'_2: *enoop*	l'_4: $x := 1 - x;$
			l'_5: *enoop*
A. Let $dom(nval_{Pr}) = \{.x\}$. Initially $x = 0$.		B. Let $dom(nval_{Pr'}) = \{.x, .y\}$. Initially both $x = 0$ and $y = 0$.	

Fig. 1. Simple concurrent programs Pr and Pr'

and framing (as noted in Fig. 1, $dom(nval_{Pr}) = \{.x\}$):

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

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 label constants 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:

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

4.6. Sequentially composing two interleaving formulas

The construct *enoop* is useful at the end of processes for handling label constants and framing. However, if we want to employ the operator chop to sequentially combine together one existing process terminating with *enoop* and a second process, then either *skip* (not mentioned amongst the programming constructs in Table 2) or *noop* can be placed in between to create a small one-state gap which avoids a clash with the label constants and framing information:

$$A; \mathit{skip}; B \qquad A; \mathit{noop}; B \ .$$

This involves the equivalence of the next three formulas:

$$\mathit{noop} \qquad \mathit{enoop}; \mathit{skip} \qquad \mathit{enoop}; \mathit{noop} \ .$$

Therefore, putting a *skip* after *enoop* combines to do exactly the same as *noop*. A simple alternative is to remove the instance of *enoop* in A, but this might not always be practical.

If it is essential to sequentially join program A unchanged to program B without the extra step entailed by an intervening *skip* or *noop*, then the formula $(\mathit{uncap}\, A);B$ can be used, where the construct $\mathit{uncap}\, A$ is defined as follows using auxiliary variables $\mathit{lab}1$ and $\mathit{nval}1$:

$$\mathit{uncap}\, A \quad \widehat{=} \quad \exists \mathit{lab}1, \mathit{nval}1. \left(A^{\mathit{lab}1,\mathit{nval}1}_{\mathit{lab},\mathit{nval}} \wedge \Box(\mathit{more} \ \supset \ (\mathit{lab} = \mathit{lab}1 \ \wedge \ \mathit{nval} = \mathit{nval}1))\right) \ . \tag{38}$$

Sequential composition with chop of two interleaving formulas both using *enoop* for all four processes could likewise be separated by an instance of *skip*. As already explained above, this helps ensure that label constants and framing information do not clash between the last state of the first interleaving formula and the first state of the second one. Here is an sample illustrative program requiring exactly 4 states:

$$(l_0 \colon \mathit{enoop} \ |||_p \ l_1 \colon \mathit{enoop}); \ \mathit{skip}; \ (l_2 \colon \mathit{enoop} \ |||_p \ l_3 \colon \mathit{enoop}) \ .$$

If the formula *skip* were not included, then the program's second state would have to satisfy the contradictory conjunction $\mathit{lab} \in \{l_0, l_1\} \wedge \mathit{lab} \in \{l_2, l_3\}$. It is possible to use *noop* instead of *skip* because even though *noop* provides redundant framing information here, it does not cause a logical clash. Another solution not requiring an added *skip* or *noop* involves omitting any instances of *enoop* in the left-hand interleaving formula, as already discussed ealier. However, this might sometimes not be feasible if the two interleaving formulas must be used unchanged, but the solution mentioned above involving the operator *uncap* defined in (38) is then still an option.

Process $Peterson_i$, for $i \in \{0,1\}$
for some times do (
l_0: *noop*;
l_1: $\mathit{flag}_i := 1$;
l_2: $\mathit{turn} := 1 - i$;
while $(\mathit{flag}_{1-i} = 1 \wedge \mathit{turn} = 1 - i)$ *do*
l_3: *noop*;
l_4: *noop*; /* Critical section */
l_5: $\mathit{flag}_i := 0$; /* Leave critical section */
l_6: *noop*
);
l_7: *enoop*

Let $dom(nval_{Peterson}) = \{.\mathit{flag}_0, .\mathit{flag}_1, .\mathit{turn}\}$.
Initially both $\mathit{flag}_0 = 0$ and $\mathit{flag}_1 = 0$, but *turn* can start as either 0 or 1.

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

4.7. Peterson's mutual exclusion algorithm

Figure 2 shows Peterson's mutual exclusion algorithm [Pet81]. The two processes $Peterson_0$ and $Peterson_1$ do not simultaneously access their critical sections (label l_4). We assume that the control variable *pid*, which indicates the currently active process, ranges over $\{0,1\}$. The interleaving compositional of the two processes is expressed as $(0 :: Peterson_0) \,|||\, (1 :: Peterson_1)$, which is equivalent here to $Peterson_0 \,|||_{pid=0}\, Peterson_1$.

Below are some valid properties, where we let *init* denote $\mathit{flag}_0 = 0 \wedge \mathit{flag}_1 = 0$ (also, as noted in Fig. 2, $dom(nval_{Peterson}) = \{.\mathit{flag}_0, .\mathit{flag}_1, .\mathit{turn}\}$). Implications (39)–(41) all concern mutual exclusion:

$$\models \quad \mathit{init} \wedge \mathit{iframe} \wedge (0 :: Peterson_0) \,|||\, (1 :: Peterson_1) \quad \supset \quad \Box\neg\big(\mathit{lab} = l_4 \wedge \bigcirc(\mathit{lab} = l_4)\big) \tag{39}$$

$$\models \quad \mathit{init} \wedge \mathit{iframe} \wedge (0 :: Peterson_0) \,|||\, (1 :: Peterson_1) \tag{40}$$
$$\supset \quad \Box\neg\big(\mathit{pid} \neq \bigcirc \mathit{pid} \wedge \mathit{lab} = l_4 \wedge \bigcirc(\mathit{lab} = l_4)\big)$$

$$\models \quad \mathit{init} \wedge \mathit{iframe} \wedge (0 :: Peterson_0) \,|||\, (1 :: Peterson_1) \tag{41}$$
$$\supset \quad \Box\neg\big((\mathit{pid} = 0 \wedge \mathit{lab} = l_4 \wedge (\mathit{pid} = 0)\,\mathcal{U}\,(\mathit{pid} = 1 \wedge \mathit{lab} = l_5)\big)$$
$$\wedge \ \Box\neg\big((\mathit{pid} = 1 \wedge \mathit{lab} = l_4 \wedge (\mathit{pid} = 1)\,\mathcal{U}\,(\mathit{pid} = 0 \wedge \mathit{lab} = l_5)\big)$$

Only the second two implications (40) and (41) have consequents which are stutter-invariant, as is explained shortly. Interestingly, in implication (39) we do not need to mention the value of *pid* in the consequent because each process is in its critical section for only one state at a time.

The consequent of implication (39) is not stutter-invariant. Observe that it is a substitution instance of the formula $\Box\neg(p \wedge \bigcirc p)$. A one-state interval with a state s having just p true satisfies this formula, and the two-state interval with both states being s is stutter-equivalent to this one-state interval but falsifies the formula. On the other hand, implication (40), which is a modified version of implication (39) and indeed a consequent of it, has a right-hand side $\Box\neg\big(\mathit{pid} \neq \bigcirc \mathit{pid} \wedge \mathit{lab} = l_4 \wedge \bigcirc(\mathit{lab} = l_4)\big)$ that is stutter-invariant even though it contains the operator $\bigcirc$. We now establish that this right-hand side is stutter-invariant:

Proof. Using p and q for $\mathit{pid} = 0$ and $\mathit{lab} = l_4$ for brevity, we can write (40) as

$$\Box\neg\big((p \not\equiv \bigcirc p) \wedge q \wedge \bigcirc q\big) \ . \tag{42}$$

Next we show that (42) has an equivalent in LTL$-\bigcirc$. First we re-express it as

$$\Box\neg\big(p \wedge q \wedge \bigcirc(\neg p \wedge q)\big) \wedge \Box\neg\big((\neg p \wedge q \wedge \bigcirc(p \wedge q)\big) \ ,$$

which in turn is equivalent to

$$\neg\Diamond\big(p \wedge q \wedge \bigcirc(\neg p \wedge q)\big) \wedge \neg\Diamond\big((\neg p \wedge q \wedge \bigcirc(p \wedge q)\big) \ .$$

The operator $\bigcirc$ in the scope of $\Diamond$ can now be supplanted by a use of $\mathcal{U}$:

$$\models \quad \Diamond\big(p \wedge q \wedge \bigcirc(\neg p \wedge q)\big) \quad \equiv \quad \Diamond\big(p \wedge q \wedge (p \wedge q)\,\mathcal{U}\,(\neg p \wedge q)\big)$$
$$\models \quad \Diamond\big((\neg p \wedge q \wedge \bigcirc(p \wedge q)\big) \quad \equiv \quad \Diamond\big(\neg p \wedge q \wedge (\neg p \wedge q)\,\mathcal{U}\,(p \wedge q)\big)\ .$$

Combining these and the previous equivalences gives an equivalent to (42) in LTL$-\bigcirc$. The consequent of implication (41) is therefore equivalent to a substitution instance of this $\Box$-formula (42), where p is replaced by $pid = 0$ and q by $lab = l_4$, so is stutter-invariant as well. □

The two implications (43) and (44) below deal with liveness:

$$\begin{aligned} \models \quad & \mathit{init} \wedge \mathit{iframe} \wedge (0 :: \mathit{Peterson}_0) \,|||\, (1 :: \mathit{Peterson}_1) \\ & \supset \quad \Box\big((\mathit{pid} = 0 \wedge \mathit{lab} = l_0) \supset \Diamond(\mathit{pid} = 0 \wedge \mathit{lab} = l_4)\big) \\ & \qquad \wedge \ \Box\big((\mathit{pid} = 1 \wedge \mathit{lab} = l_0) \supset \Diamond(\mathit{pid} = 1 \wedge \mathit{lab} = l_4)\big) \end{aligned} \tag{43}$$

$$\begin{aligned} \models \quad & \mathit{init} \wedge \mathit{iframe} \wedge (\mathit{inf} \wedge 0 :: \mathit{Peterson}_0) \,|||\, (\mathit{inf} \wedge 1 :: \mathit{Peterson}_1) \\ & \supset \quad \Box\Diamond(\mathit{pid} = 0 \wedge \mathit{lab} = l_0) \wedge \Box\Diamond(\mathit{pid} = 0 \wedge \mathit{lab} = l_4) \\ & \qquad \wedge \ \Box\Diamond(\mathit{pid} = 1 \wedge \mathit{lab} = l_0) \wedge \Box\Diamond(\mathit{pid} = 1 \wedge \mathit{lab} = l_4)\ . \end{aligned} \tag{44}$$

The consequents of implications (43) and (44) are clearly stutter-invariant because they are in LTL$-\bigcirc$. Both implications (43) and (44) can alternatively be expressed somewhat more concisely using the projection operator in the consequents:

$$\begin{aligned} \models \quad & \mathit{init} \wedge \mathit{iframe} \wedge (0 :: \mathit{Peterson}_0) \,|||\, (1 :: \mathit{Peterson}_1) \\ & \supset \quad (\mathit{pid} = 0)\ \Pi\ \Box(\mathit{lab} = l_0 \supset \Diamond \mathit{lab} = l_4) \ \wedge \ (\mathit{pid} = 1)\ \Pi\ \Box(\mathit{lab} = l_0 \supset \Diamond \mathit{lab} = l_4) \end{aligned} \tag{45}$$

$$\begin{aligned} \models \quad & \mathit{init} \wedge \mathit{iframe} \wedge (\mathit{inf} \wedge 0 :: \mathit{Peterson}_0) \,|||\, (\mathit{inf} \wedge 1 :: \mathit{Peterson}_1) \\ & \supset \quad (\mathit{pid} = 0)\ \Pi\ (\Box\Diamond \mathit{lab} = l_0 \wedge \Box\Diamond \mathit{lab} = l_4) \ \wedge \ (\mathit{pid} = 1)\ \Pi\ (\Box\Diamond \mathit{lab} = l_0 \wedge \Box\Diamond \mathit{lab} = l_4)\ . \end{aligned} \tag{46}$$

Our earlier presentation in [MG15] used a different approach with a derived operator used to observe *lab* in the course of process execution. However, after some further investigation, we now prefer the one employed here since it seems more straightforward (e.g., our approach in [MG15] required adding extra instances of *noop*). Furthermore, it is also better suited for reasoning about some proofs of mutual exclusion discussed here.

Now that Peterson's algorithm has been presented here, let us consider three possible approaches to verifying the correctness of it and other such finite-state programs:

- *Decision procedures and model checkers*: Perhaps the most immediately appealing method is to simply use an implemented decision procedure or model-checking tool to mechanically check that the algorithm's behaviour complies with the desired correctness properties. No detailed formal proof is required about the intricacies of how the algorithm's individual processes interact.
- *Reduction using LTL and global time*: It is not hard to first transform the individual processes in our version of Peterson's algorithm to LTL formulas concerning how each process operates between each adjacent pair of its *local* states but as seen from the standpoint of the common *global* interval. This is routinely done when representing such algorithms in LTL as low-level *transition systems* (see for example the detailed presentation of Peterson's algorithm by Kröger and Merz [KM08]). Such an approach has the advantage of employing more established notation and techniques without any explicit projection being done. However, the resulting LTL formulas retain little of the sequential structure of the original processes. Consequently, exclusively transition-based reasoning might sacrifice some benefits potentially obtained by doing more reasoning about the structure of each process in its original sequential form and in its own *local* time, rather than leaving this to be largely carried out in *global* time.
- *Reasoning about each processing using LTL and local time*: One can seek to increase the amount of reasoning in *local* time about a process. As with the previous approach, the relevant behaviour of each process in its *local* time is reduced to LTL formulas, but these can be obtained more directly by focusing mainly on the structure of the individual process. Although such formulas primarily pertain to *local* time, when the interaction between the processes ultimately needs to be considered, some global reasoning can be employed. This style of reasoning therefore employs a much less familiar idiom than the previous one solely involving global reasoning, so certainly requires greater investment. However, this could result in a powerful reusable infrastructure which in the long run might be more appropriate for some purposes.

The actual proofs of the correctness of Peterson's algorithm are presented in the appendix and are based on the third approach, thereby increasing the amount of reasoning carried out in the local time of each individual process using the original structure of the process. The appendix then concludes by briefly examining another variant of Peterson's algorithm without labels.

5. Some alternative techniques and notations

We now consider some other ways to reason about interleaving concurrency with PITL, such as without Π.

5.1. Formalising interleaving without projection

As discussed above, our modelling of interleaving employs Π together with the record variable *nval* and the *iframe* construct that ensure each assignment to a program variable is suitably performed between two *globally* adjacent states. Two alternative frameworks which can be viewed as *mimicking* projection without using Π are now discussed:

- The first simply dispenses with Π but still uses *nval* and *iframe*.
- The second also avoids needing *nval* and *iframe*, instead just using a variable *pvars* which denotes the set of program variables' field-name constants and plays a role like that of $dom(nval)$.

The second approach is the closest to the interleaving semantics described by Baier and Katoen [BK08].

In what follows, the two projection-less approaches are presented and shown under some general assumptions to express the same kind of behaviour as the earlier framework using projection. It seems natural to initially just prove a notion of equivalence between the previously described framework with projection and the first projection-less approach, and only then similarly relate the two projection-less approaches. As a result, all three are shown to indeed capture the same behaviour.

5.1.1. First projection-less approach using nval and iframe

We now present the first globally oriented projection-less approach which retains both *nval* and *iframe*. The only constructs in Table 2 which need to be changed are the assignment operator :=, *noop* and *enoop*. The alternatives $:='$, $noop'$ and $enoop'$ defined shortly can be viewed as *mimicking* projection with no need for the operator Π itself. The previously derived construct *iframe* and the record-variable *nval* are still used with $:='$, $noop'$ and $enoop'$ *exactly* as they were used above with :=, *noop* and *enoop*, so do not require variants to be introduced here. Below is a definition of the alternative construct $:='$ for assigning to a single variable, where the locally scoped boolean variable *active* indicates when the process is active:

$$a :=' e \quad \widehat{=} \quad nval[.a] = e \ \wedge \ \forall v \in (dom(nval) \setminus \{.a\}).\, (nval[v] = v^{\wedge}) \\ \wedge \ active \ \wedge \ finite \ \wedge \ \bigcirc \Box (more \supset \neg active) \ .$$

Each process in effect has its own private instance of *active*. In the first state of a finite interval satisfying the assignment formula, the variable *active* is *true*, but then *false* in the finite number of any subsequent intermediate states (which exclude the interval's last state), to indicate temporary inactivity of the process. It is important to note that $:='$ *does not* determine *active*'s value in the interval's last state since this is left for a follow-on atomic statement to do.

The following are definitions for multiple assignments and the alternative construct $noop'$:

$$a_1, \ldots, a_n :=' e_1, \ldots, e_n \quad \widehat{=} \quad nval[.a_1] = e_1 \ \wedge \ \cdots \ \wedge \ nval[.a_n] = e_n \\ \wedge \ \forall v \in (dom(nval) \setminus \{a_1, \ldots, a_n\}).\, ((nval[v]) = v^{\wedge}) \\ \wedge \ active \ \wedge \ finite \ \wedge \ \bigcirc \Box (more \supset \neg active)$$

$$noop' \quad \widehat{=} \quad \forall v \in dom(nval).\, (nval[v] = v^{\wedge}) \\ \wedge \ active \ \wedge \ finite \ \wedge \ \bigcirc \Box (more \supset \neg active) \ .$$

The variant $enoop'$ below of *enoop* includes an addition conjunct ensuring *active* holds in a process's final state:

$$enoop' \quad \widehat{=} \quad \forall v \in dom(nval).\, (nval[v] = v^{\wedge}) \ \wedge \ active \ \wedge \ empty \ .$$

Consider the following simple adaptation of process Pr'_1, which itself was already presented in Fig. 1 in Sect. 4.5:

$$\begin{array}{ll} & \mathit{while}\, y = 0\, \mathit{do} \\ l'_3: & \quad \mathit{noop}'; \\ l'_4: & x :=' 1 - x; \\ l'_5: & \mathit{enoop}' \end{array}$$

An interval satisfies this formula iff the interval has the next few properties:

- The interval's first state satisfies *active* and so does the last state if the interval terminates. Furthermore, the sequence of states with *active* true in them starts with a series of zero or more states (possibly an infinite number) all with $y = 0$ and $lab = l'_3$. If the interval is finite, then there are two further states with *active* true: The first satisfies $y \neq 0$ and $lab = l'_4$, and the final one satisfies $lab = l'_5$. All of this is expressed by the formula below which satisfies the interval and reflects the behaviour of the control variables *active* and *lab*:

$$\begin{array}{l} \mathit{active} \ \wedge \ \mathit{fin}\, \mathit{active} \\ \wedge \ \mathit{active}\ \Pi\ \big(\mathit{while}\, y \neq 0\, \mathit{do}\, (\mathit{skip} \wedge \mathit{lab} = l'_3); (\mathit{skip} \wedge \mathit{lab} = l'_4); (\mathit{empty} \wedge \mathit{lab} = l'_5)\big)\ . \end{array} \tag{47}$$

- If a state in the interval has *active* true and $lab \in \{l'_3, l'_5\}$, then the state satisfies the conjunction $nval[.x] = x \wedge nval[.y] = y$.
- If a state in the interval has *active* true and $lab = l'_4$, then the state satisfies the conjunction $nval[.x] = x - 1 \wedge nval[.y] = y$.

Below is a formula which is semantically equivalent to (47) and likewise characterises the behaviour of the label variable *lab* but uses the operators $\bigcirc$ and *halt* instead of the projection operator Π:

$$\big(\mathit{while}\, y \neq 0\, \mathit{do}\, (\mathit{skip}_{\mathit{active}} \wedge \mathit{lab} = l'_3)\big);\ \ (\mathit{skip}_{\mathit{active}} \wedge \mathit{lab} = l'_4);\ \ (\mathit{empty} \wedge \mathit{active} \wedge \mathit{lab} = l'_5)\ ,$$

where for any state formula w the derived construct skip_w denotes the conjunction $\mathit{finite} \wedge \bigcirc \mathit{halt}\, w$. Therefore, the formula $\mathit{skip}_{\mathit{active}}$ satisfies an interval iff the interval is finite with at least two states, and has *active* false in all states starting from the *second* one except the final one, where *active* must be true. The value of *active* in the first state is ignored by $\mathit{skip}_{\mathit{active}}$.

Any process A constructed by combining the atomic statements $:='$ and noop' together with conditional statements, chop, chop-star, while-loops, etc. and ending with enoop' has the next valid implication:

$$A \ \supset\ (\mathit{active} \wedge \mathit{fin}\, \mathit{active})\ .$$

This is because the atomic statements $:='$ and noop' require *active* to be true in the first state of their interval. Furthermore, the construct enoop', when placed at the end of the definition of a process, makes *active* equal *true* in a process's final state if the process terminates. When used with a label constant, enoop' also determines *lab*'s value in that state (just as a labelled *enoop* does.)

Here is a valid implication relating $:=$ and $:='$ and concerning how $:='$ can be regarded as *mimicking* projection:

$$(\mathit{active} \ \wedge \ \mathit{fin}\, \mathit{active}) \quad \supset \quad a :=' e \ \equiv \ (\mathit{active}\ \Pi\ a := e)\ . \tag{48}$$

Proof of validity of implication (48). Let σ be an interval satisfying the antecedent of (48). Hence, *active* is true in the first state of σ. If σ has finite length, then *active* is also true in the last state of σ. Now if σ satisfies $a :=' e$, then the following hold by the definition of $:='$:

1. σ has finite length and at least two states.
2. The variable *active* is true in the first state of σ and false in any intermediate states of σ.
3. The first state of σ satisfies the state formula below:

$$nval[.a] = e \ \wedge \ \forall v \in (dom(nval) \setminus \{.a\}).\, (nval[v] = v\hat{}\,)\ . \tag{49}$$

Let σ' denote here the projected interval obtained from σ by selecting just the states in which *active* is true. It readily follows that the three properties just enumerated hold for σ iff the following three hold for σ':

1. σ' has finite length.
2. σ' has exactly two states and so satisfies *skip*.

3. The first state of σ satisfies the state formula (49).

Hence, σ satisfies $a :=' e$ iff σ' satisfies $a := e$. It then follows from the assumption of σ satisfying the conjunction $active \wedge fin\ active$ that σ satisfies $a :=' e$ iff σ satisfies $active\ \Pi\ a := e$, so implication (48) is indeed valid. □

The following special case of valid implication (48) shows that for any interval where *active* is *always* true (not just at the beginning and end), the operators := and :=′ can be viewed as equivalent:

$$\Box active \quad \supset \quad a :=' e \ \equiv \ a := e \ . \tag{50}$$

Proof of validity of implication (50). Let σ be an interval satisfying the formula $\Box active$. Hence, *active* is true in every state of σ. The next instance of the simple valid implication $\Box w \supset (w \wedge fin\ w)$ for any state formula w is likewise valid:

$$\Box active \quad \supset \quad active \ \wedge \ fin\ active \ .$$

The chaining of this implication with the earlier valid one (48) yields the next valid implication:

$$\Box active \quad \supset \quad a :=' e \ \equiv \ (active\ \Pi\ a := e) \ . \tag{51}$$

Now the following is an instance of valid implication (13) concerning when the global interval and a projected one are identical:

$$\Box active \quad \supset \quad a := e \ \equiv \ (active\ \Pi\ a := e) \ . \tag{52}$$

The combination of the two valid implications (51) and (52) together with simple proposition reasoning yields that implication (50) is indeed valid. □

Here is a variant of the valid implication (48) which further illustrates how to relate :=′ and := and also how :=′ can be seen as mimicking projection:

$$\neg active\ \mathcal{U}\ \big(a :=' e; (active \wedge \circledcirc\Box\neg active)\big) \quad \equiv \quad (active\ \Pi\ a := e) \ . \tag{53}$$

Proof of validity of implication (53). This follows because the until-formula's left operand $\neg active$ causes the $\mathcal{U}$-operator to skip past any starting portion of the global interval prior to the first state satisfying *active*. The chop-formula's right-hand operand $active \wedge \circledcirc\Box\neg active$ similarly ignores any tailing portion of the interval beyond the furthest state satisfying *active* if *active* is true only in a finite number of the interval's states. If the interval has an infinite number of states satisfying *active*, then both sides of equivalence (53) are readily shown to be falsified by the interval. □

Valid implications quite similar to (53) can be readily obtained for relating *noop* and *noop*′ and the two versions of multiple assignments.

We can use induction on syntax to relate an individual process built with the various constructs presented earlier in Table 2 with a projection-less variant:

Proposition 5.1. Suppose that A is some process built using the constructs in Table 2. Let A' be obtained from A by replacing all instances of :=, *noop* and *enoop* with :=′, *noop*′ and *enoop*′, respectively. Then the following implications are valid:

$$(active \ \wedge \ fin\ active) \quad \supset \quad A' \ \equiv \ (active\ \Pi\ A) \tag{54}$$

$$\Box active \quad \supset \quad A' \ \equiv \ A \ . \tag{55}$$

Proof. The proof of the validity of implication (54) is by straightforward induction on the structure of A's syntax using the previous valid implication (48), an analogous one for relating *noop* and *noop*′ and then extending this to handle the various control structures and *enoop*/*enoop*′.

The second implication (55) is a special instance of (54) and generalises the earlier valid implication (50). The proof of the validity of implication (55) likewise generalises the proof of implication (50)'s validity to handle other atomic statements and also the various constructs such as chop and while-loops for combining statements together. □

Below are variants of Π and $|||\_$ which seem suitable for use for expressing interleaving concurrency

without the projection operator Π:

$$w\ \pi\ A \quad \widehat{=} \quad \neg w\ \mathcal{U}\ \big((w \wedge A^{w}_{active}); (w \wedge \circledcirc\Box\neg w)\big) \tag{56}$$

$$A\ |||'_{w}\ B \quad \widehat{=} \quad (active \wedge w)\ \pi\ A\ \ \wedge\ \ (active \wedge \neg w)\ \pi\ B\ . \tag{57}$$

Here the formula A^{w}_{active} denotes an instance of A with all free occurrences of *active* replaced by the state formula w. This substitution is done so that w selects which states in an interval are in effect regarded as projected when A's truth value is evaluated (thus achieving a variant of the technique used above in valid equivalence (53)). The derived operator $w\ \pi\ A$ is similar to $w\ \Pi\ A$, but instead of examining a projected subinterval of the original interval, π relies on substituting w into all free occurrences of variable *active* to just examine states in the interval where w holds. Therefore, unlike with Π, the use of π relies on atomic statements such as $:='$ in A, which when *active* is replaced in them by w, help to mimic projection by ignoring any states falsifying w. Properties of Π such as (9) can be adapted to π. The role of π in the definition of $|||'_{\_}$ is similar to that of Π in the definition of $|||\_$. Here is a way to formally relate π with Π:

Proposition 5.2. Let A and A' be formulas such that the variable *active* does not occur freely in A and the next implication is valid:

$$(active\ \wedge\ fin\ active) \quad \supset \quad A' \ \equiv\ (active\ \Pi\ A)\ . \tag{58}$$

Then the following equivalence is valid as well:

$$w\ \pi\ A' \quad \equiv \quad w\ \Pi\ A\ . \tag{59}$$

Proof of Proposition 5.2. Let σ be some interval. If w is false in all states of σ, then σ does not satisfy either $w\ \pi\ A'$ or $w\ \Pi\ A$, so trivially satisfies equivalence (59). Otherwise, w is true somewhere in σ. Let σ' be the unique subinterval of σ with one of the following two properties, depending on whether w occurs finitely or infinitely often in σ:

- If w occurs *finitely* often in σ, then σ' is the finite *infix* subinterval of σ starting the first state of σ satisfying w and terminating with the last such state.
- If w occurs *infinitely* often in σ, then σ' is the infinite *suffix* subinterval of σ starting with the first state of σ satisfying w.

Let the interval σ'' be obtained from σ' by in each state setting the variable *active* to the formula w's value in that state. The construction of σ'' ensures that it satisfies valid implication (58)'s antecedent, so it also satisfies the implication's consequent $A' \equiv (active\ \Pi\ A)$. Hence, σ'' also satisfies the equivalence $(A')^{w}_{active} \equiv (w\ \Pi\ A)$ because *active* and w are state formulas and equivalent in all the states in σ''. The equivalence $(A')^{w}_{active} \equiv (w\ \Pi\ A)$ has no free instances of *active*, so σ' satisfies it as well. Furthermore, the construction of σ' then ensures that for any formula B, σ' satisfies B iff the original interval σ satisfies $\neg w\ \mathcal{U}\ \big((w \wedge B); (w \wedge \circledcirc\Box\neg w)\big)$. In addition, as another result of the formula A having no occurrences of the variable *active*, the interval σ satisfies $w\ \Pi\ A$ iff the interval σ'' satisfies $w\ \Pi\ A$ because the projection done here ignores any starting and ending portions of σ before and after σ' and σ''. Such portions can only contain states falsifying w. Hence, the original interval σ satisfies the next equivalence:

$$\neg w\ \mathcal{U}\ \big((w \wedge (A')^{w}_{active}); (w \wedge \circledcirc\Box\neg w)\big) \quad \equiv \quad w\ \Pi\ A\ .$$

The equivalence's left-hand operand is the definition of $w\ \pi\ A'$, so the equivalence is semantically identical to equivalence (59), and σ therefore likewise satisfies this. □

Proposition 5.2 helps to formally relate $|||\_$ and $|||'_{\_}$ in a similar manner, but we omit the details. Variants of the properties of commutativity (28) and associativity (29)-(30) for $|||\_$ can also be shown for $|||'_{\_}$.

5.1.2. Second projection-less approach without nval and iframe

Let us now consider how to dispense with the use of the record-variable *nval* and *iframe*. Instead of *nval*, a set variable *pvars* is employed to denote the set of program variables' field-name constants and play a role like that of $dom(nval)$. There is no need for *nval* and primed variables because the assignment statement used here, which is denoted as $a :='' e$, assigns e to the value of the variable a in the second state of the interval using the *temporal* equality $(\bigcirc a) = e$ rather than the state formula $nval[.a] = e$ or its equivalent

$a' = e$. This dispenses with any need for *iframe* to do the actual assignment. However, the treatment of suitable versions of *enoop* and π, respectively denoted here as $enoop''$ and π'', is more complicated than before.

Below is a definition of the alternative construct $:=''$ which uses the next-operator to assign to a single variable:

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

Here the operator $\bigcirc$ helps assign to a and frame other program variables over the first two (global) states. As before in the previous Sect. 5.1.1, we use the boolean variable *active* to indicate when the process is active. Each process has its own private instance of *active*. Here are analogous definitions for multiple assignments and the alternative construct $noop''$:

$$a_1, \ldots, a_n :='' e_1, \ldots, e_n \quad \widehat{=} \quad (\bigcirc a_1) = e_1 \ \wedge \ \cdots \ \wedge \ (\bigcirc a_n) = e_n \\ \wedge \ \forall v \in pvars \setminus \{a_1, \ldots, a_n\}.\, ((\bigcirc v\hat{}) = v\hat{}) \\ \wedge \ active \ \wedge \ finite \ \wedge \ \bigcirc\Box(more \supset \neg active)$$

$$noop'' \quad \widehat{=} \quad \forall v \in pvars.\, ((\bigcirc v\hat{}) = v\hat{}) \\ \wedge \ active \ \wedge \ finite \ \wedge \ \bigcirc\Box(more \supset \neg active) \ .$$

Below is the definition of $enoop''$:

$$enoop'' \quad \widehat{=} \quad \big(more \supset \forall v \in pvars.\, ((\bigcirc v\hat{}) = v\hat{})\big) \\ \wedge \ active \ \wedge \ \circledcirc\Box\neg active \ .$$

As can be seen from this definition, $enoop''$ does not restrict its interval to having just one state. This is because $enoop''$, unlike *enoop* and $enoop'$, cannot employ *nval* to instruct *iframe* to set the value of each program variable v in the second state, if there is one, to equal v's value in the first state. Instead, $enoop''$ *itself* must set the variables' values in the second state if the interval has more than one.

Here is a valid implication concerning $enoop''$ and *active*:

$$enoop'' \quad \supset \quad active \ \wedge \ \circledcirc\Box\neg active \ .$$

This implication shows that $enoop''$ can be true in an interval with multiple states, so is quite different from the corresponding valid implications below for *enoop* and $enoop'$ which show that *enoop* and $enoop'$ each require an interval having exactly one state:

$$enoop \quad \supset \quad empty$$

$$enoop' \quad \supset \quad empty \wedge active \ .$$

If $pvars = dom(nval)$, then the following valid formulas relate $enoop''$ to *enoop* and $enoop'$, respectively:

$$enoop'' \ \wedge \ \forall v \in dom(nval).\, (nval[v] = v\hat{}) \quad \equiv \\ active \ \wedge \ \big(more \supset \forall v \in dom(nval).\, (nval[v] = \bigcirc v\hat{})\big) \ \wedge \ (enoop;\ \circledcirc\Box\neg active) \tag{60}$$

$$iframe \quad \supset \quad enoop'' \equiv (enoop';\ \circledcirc\Box\neg active) \ .$$

Proof. We only consider here the proof for the first formula (60) since the second one's proof is similar. Let σ be an interval satisfying the first implication's antecedent $iframe \wedge active$. There are two cases, depending on whether σ has just one state or not:

- *Only one state*: In a one-state interval, $enoop''$ just sets *active* to true. On the other hand, *enoop* does not control the boolean variable *active* but does set *nval*'s elements equal to the values of the associated program variables. Therefore, equivalence (60) compensates by including the formula $\forall v \in dom(nval).\, (nval[v] = v\hat{})$ in the equivalence's left-hand conjunction containing $enoop''$ and likewise having *active* as a conjunct in the equivalence's right-hand conjunction.
- *More than one state*: In an interval with multiple states, $enoop''$ sets *active* to true in the first state and to false in the remaining ones. Furthermore, for each program variable mentioned in *pvar*, $enoop''$ itself frames the variable between interval σ's first and second states by setting explicitly the variable's value in the second state to remain equal to the variable's value in the first one. On the other hand, *enoop* does not control *active*, so *active* must be explicitly set to true in equivalence (60)'s right-hand conjunction.

Moreover, *enoop* sets *nval*'s elements equal to the values of the associated program variables. Therefore, equivalence (60) includes this explicitly in the left-hand conjunction containing $enoop''$. Furthermore, *enoop* is only true in one-state intervals and so cannot itself preserve the values of program variables from the first state of σ to the second one. Therefore, an explicit conjunct needs to be included here in equivalence (60)'s right-hand conjunction and is based on *iframe*'s definition in (35) but unlike *iframe* only needs to operate between interval σ's first and second states.

□

Any such process A ending with $enoop''$ has the valid implication below which does not require *active* to be true in the last state of a finite interval satisfying A:

$$A \quad \supset \quad active \ \wedge \ (inf \vee \Diamond \circledcirc \Box \neg active) \ . \tag{61}$$

Proof. The program for a process A has the form $B; l\colon enoop''$, for some subprogram B and label l. Here is a valid implication with $B; l\colon enoop''$ serving as the antecedent:

$$B; l\colon enoop'' \quad \supset \quad (inf \vee \Diamond enoop'') \ .$$

The validity of this follows from it being an instance of the next valid implication concerning chop, where C and C' are arbitrary PITL formulas:

$$C; C'' \quad \supset \quad (inf \vee \Diamond C'') \ .$$

The chop operator is *weak* (as already observed in Sect. 2), so it is does not require the formula C to terminate. Hence, C'' might never take place in an infinite interval. Now the definition of $enoop''$ contains $\circledcirc \Box \neg active$ as a conjunct. Hence, $\Diamond enoop''$ implies $\Diamond \circledcirc \Box \neg active$, so implication (61) is indeed valid. □

Implication (61) is quite different from the corresponding implication below for programs constructed using the first approach without Π already presented in Sect. 5.1.1:

$$A \quad \supset \quad active \wedge fin \ active \ .$$

Here is a valid implication relating $:='$ and $:=''$, provided that we take sets $dom(nval)$ and *pvars* to be equal:

$$iframe \quad \supset \quad a :=' e \ \equiv \ a :='' e \ . \tag{62}$$

Note that the definition of $:='$ uses *nval*, whereas the definition of $:='$ instead uses the operator $\bigcirc$. Therefore, we need to include *iframe* in the implication to relate the two approaches.

If A_1 is a program constructed using the first approach without projection, and A_2 an analogous program constructed using the second approach without projection (e.g., each $:='$ in A_1 is replaced by $:=''$ in A_2), then the following generalisation of valid implication (62) can proved valid by structural induction on the syntax of A_1:

$$iframe \quad \supset \quad A_1 \ \equiv \ A_2 \ . \tag{63}$$

Proof. The proof makes use of valid implication (62), and other similar valid implications for other atomic statements (e.g., to likewise relate $noop'$ and $noop''$) together with induction on the syntax of program A_1. □

Valid implication (62) can be combined with the earlier one (54) to relate programs built using $:=$, *noop* and *enoop* (and so requiring *nval* and *iframe*) to those constructs using the second approach without projection (which completely dispenses with *nval* and *iframe*):

$$(active \ \wedge \ fin \ active \ \wedge \ iframe) \quad \supset \quad A_2 \ \equiv \ (active \ \Pi \ A_0) \ ,$$

where A_0 is a program built using constructs such as $:=$ and *noop*, and A_2 is an corresponding program instead built using $:=''$ and $noop''$.

Proof. Let A_1 be the same program as A_0 but with constructs such as $:='$ and $noop'$. Validity can be readily shown by combining an instance of valid implication (54) for programs A_0 and A_1 and valid implication (63) for programs A_1 and A_2. □

Here are suitable variants of Π and $|||\_$ for expressing interleaving concurrency by means of the second approach without the projection operator Π, so without the need for either *nval* or *iframe*:

$$w\ \pi''\ A \quad \widehat{=} \quad \neg w\ \mathcal{U}\ (w \wedge A^w_{active}) \tag{64}$$

$$A\ |||''_w\ B \quad \widehat{=} \quad (active \wedge w)\ \pi''\ A\ \ \wedge\ \ (active \wedge \neg w)\ \pi''\ B\ . \tag{65}$$

Definition (64) of π'' omits the subformula $w \wedge \circledcirc\Box\neg w$ found in Definition (56) of π because this aspect of the behaviour of π in the first projection-less approach is instead the responsibility of *enoop*$''$ to enforce in the second projection-less approach as explained above regarding the definition of *enoop*$''$.

Remark 5.3. The variable *pvars* used here can also be employed instead of *nval* to obtain a variant of our projection-based framework. Each program variable a has an associated primed variable a' as well as a name constant $.a$. The set *pvars* contains all the program variables' name constants. The value of the variable a can alternatively be accessed via the name constant $.a$ using the dereferencing operation $.a\hat{}$. The operation $prime(.a)$ takes the name constant $.a$ for a and yields a name constant $.a'$ for the corresponding primed variable a'. Hence, $prime(.a)\hat{}$ can be used to access the value of the primed variable a'. Here is a semantics for $:=$ using this convention:

$$a := e \quad \widehat{=} \quad skip\ \wedge\ (.a)\hat{} = e\ \wedge\ \forall v \in pvars \setminus \{.a\}.\ \big(prime(v)\hat{} = v\hat{}\big)$$

5.2. Projection and interleaving with non-fairness

In this section we describe two ways to represent non-fair interleaving of processes. The first one is a modification of the approach in Sect. 5.1 on formalising interleaving without projection. The second involves Π and a variant of $|||\_$.

Recall from Sect. 4.4 that the programming semantics used there provides a form of *fairness* at the statement level since a process is never denied the chance to execute an atomic statement such as $:=$ or *noop*. Sometimes it is useful to dispense with this and instead allow for the possibility of *non-fair* scheduling in which a process can get completely ignored after a certain time point.

We now discuss one way to do this in which assignment statements and noops are the possible starting points of starvation. First of all, the following non-fair variants of the *projection-less* constructs $:='$, *noop*$'$ and *enoop*$'$ are introduced:

$$a :='_{nf}\ e \quad \widehat{=} \quad (inf \wedge \Box\neg active)\ \vee\ a :='\ e$$

$$noop'_{nf} \quad \widehat{=} \quad (inf \wedge \Box\neg active)\ \vee\ noop'$$

$$enoop'_{nf} \quad \widehat{=} \quad (inf \wedge \Box\neg active)\ \vee\ enoop'\ .$$

As these definitions show, each of the non-fair atomic statement constructs has two modes of operation in order for the construct to appropriately determine whether it is scheduled to run or has been timed out from now on. The behaviour of the control variable *active* determines which mode is selected. Either *active* can be false infinitely long ($inf \wedge \Box\neg active$) or each corresponding *fair* atomic statement's definition requires *active* to true in the statement's initial state. Unlike the earlier constructs such as $:=$ and $:='$, the non-fair constructs here can therefore cause the process to cease functioning normally. It is no longer active, and any further statements get completely ignored because they can occur only after the preceding weak chop operand which does not terminate.

The definition of the non-fair construct for multiple assignments follows naturally:

$$a_1, \ldots, a_n :='_{nf}\ e_1, \ldots, e_n \quad \widehat{=} \quad (inf \wedge \Box\neg active)\ \vee\ a_1, \ldots, a_n :='\ e_1, \ldots, e_n\ .$$

The remaining sequential constructs in Table 2 can be used unchanged.

Here is a valid implication relating $a :='\ e$ with $a :='_{nf}\ e$ in a finite interval:

$$finite \quad \supset \quad a :='_{nf}\ e\ \equiv\ a :='\ e\ .$$

Earlier in (56) and (57), we introduced variants π and $|||'_\_$ of Π and $|||\_$, respectively, for expressing

concurrency without the projection operator Π. The versions below seem suitable here:

$$w\ \pi_{nf}\ A \quad \widehat{=} \quad (\mathit{halt}\ w);\big((w \wedge A^{w}_{\mathit{active}});(w \wedge \circledcirc\Box\neg w)\big)$$
$$A\ |||'_{w}\ B \quad \widehat{=} \quad (\mathit{active} \wedge w)\ \pi_{nf}\ A\ \ \wedge\ \ (\mathit{active} \wedge \neg w)\ \pi_{nf}\ B\ .$$

Unlike its fair counterpart $w\ \pi\ A$ in (56), the formula $w\ \pi_{nf}\ A$ permits w to remain forever false. This happens when the associated process never gets scheduled to run, and the formula w in $\mathit{halt}\ w$ is always false. In contrast, the (strong) until-formula in (56) excludes such behaviour there.

Let us now consider versions of the programming constructs in Table 2 for use with non-fairness and the projection operator Π. Here are non-fair versions of the original *projection-based* constructs :=, *noop* and *enoop*:

$$\begin{array}{lcl} a :=_{nf} e & \widehat{=} & (\mathit{inf}\ \wedge\ \Box\neg\mathit{active})\ \vee\ (\mathit{active}\ \wedge\ a := e) \\ \mathit{noop}_{nf} & \widehat{=} & (\mathit{inf}\ \wedge\ \Box\neg\mathit{active})\ \vee\ (\mathit{active}\ \wedge\ \mathit{noop}) \\ \mathit{enoop}_{nf} & \widehat{=} & (\mathit{inf}\ \wedge\ \Box\neg\mathit{active})\ \vee\ (\mathit{active}\ \wedge\ \mathit{enoop}) \end{array}$$

The constructs :=, *noop* and *enoop* (unlike their counterparts :=′, *noop*′ and *enoop*′ defined earlier) do not set the control variable *active* to true, so the definitions of the non-fair versions $:=_{nf}$, noop_{nf} and enoop_{nf} need to explicitly do this.

Before presenting a suitable non-fair variant of the interleaving operator $|||\_$ for use with such constructs, we need to look at how to use Π to express non-fair projected behaviour. Consider the following instance of projection with Π:

$$(p \vee (\mathit{inf} \wedge \Box\neg p))\ \Pi\ A\ . \tag{66}$$

The left-hand LTL subformula of this Π construct is equivalent to the LTL formulas below:

$$\neg p\ \supset\ (\mathit{inf} \wedge \Box\neg p) \qquad p\ \vee\ (\neg p \wedge \Box(\neg p\ \supset\ \bigcirc\neg p)\big) \qquad p \vee \Box(\neg p\ \supset\ \bigcirc\neg p)\ .$$

The key difference between the Π-formula (66) and $p\ \Pi\ A$ is as follows: If p eventually becomes continuously false for an *infinite* number of states in the global interval, then the projected interval seen in the right-hand operand of (66) includes all of these global states. This projected interval never includes any other global states in which p is false. Therefore, the following Π-formula is satisfiable:

$$(p \vee (\mathit{inf} \wedge \Box\neg p))\ \Pi\ (\mathit{inf} \wedge \Diamond\Box\neg p)\ .$$

In contrast, the projected interval for $p\ \Pi\ A$ never sees *any* global states in which p is false. Furthermore, the following Π-formulas are *not* satisfiable:

$$p\ \Pi\ \Diamond\neg p \qquad (p \vee (\mathit{inf} \wedge \Box\neg p))\ \Pi\ (\mathit{finite} \wedge \Diamond\neg p)$$
$$(p \vee (\mathit{inf} \wedge \Box\neg p))\ \Pi\ (\mathit{inf} \wedge \Box\Diamond p \wedge \Diamond\neg p)\ .$$

The first two show the contrast between projection using just p and using $p \vee (\mathit{inf} \wedge \Box\neg p)$. In the first case, the projected interval never has any states satisfying $\neg p$, but in the second one, such states can indeed occur but only if the projected interval is itself *infinite*. The third sample formula is unsatisfiable because when the subformula $p \vee (\mathit{inf} \wedge \Box\neg p)$ is used for projecting, some projected state can falsify p iff there are an infinite number of such states which form a *suffix* of the projected interval (and the global interval as well). Therefore, if p is infinitely often true in the projected interval, no such infinite suffix of states exists with p false in all of them. Hence, p cannot be false *anywhere* in the projected interval if p is true infinitely often in it.

Here is an instance of (66) to help further illustrate its behaviour:

$$(p \vee (\mathit{inf} \wedge \Box\neg p))\ \Pi\ \Box q\ . \tag{67}$$

An interval σ satisfying this has all of the following hold:

- If σ is finite, then p is true in σ's last state.
- Either p is somewhere true in σ or some suffix subinterval of σ (possibly σ itself) has infinite length with p being false in all states of the suffix.
- Every state of σ that has p true also has q true.
- If σ has infinite length, then for every suffix in which p is always false, q is true in all of the suffix's states.

As indicated above, there are alternative formulas which can be used on the left of Π instead of $p \vee (\mathit{inf} \wedge \Box\neg p)$ to achieve exactly the same non-fair behaviour, while simultaneously giving a somewhat different notational perspective. For example, the subformula $\mathit{inf} \wedge \Box\neg p$ can be expressed as shown below:

$$\neg p \ \wedge \ \Box\big(\neg p \ \supset \ \bigcirc\neg p\big) \ .$$

Therefore, the Π-formula (66) is semantically equivalent to the following one which replaces Π's left operand $p \vee (\mathit{inf} \wedge \Box\neg p)$ with the semantically equivalent formula $\neg p \supset (\mathit{inf} \wedge \Box\neg p)$:

$$(\neg p \ \supset \ (\mathit{inf} \wedge \Box\neg p)) \ \Pi \ A \ .$$

The non-fair Π-formula (66) has the following simple property:

$$\models \quad (p \vee (\mathit{inf} \wedge \Box\neg p)) \ \Pi \ \Diamond\neg p \quad \supset \quad (p \vee (\mathit{inf} \wedge \Box\neg p)) \ \Pi \ \Diamond(\mathit{inf} \wedge \Box\neg p) \ . \tag{68}$$

Implication (68) states that if a projected interval obtained obtained using $(p \vee (\mathit{inf} \wedge \Box\neg p)) \ \Pi \ \Diamond\neg p$ satisfies $\neg p$ in some state, then this projected interval must be infinite and eventually have p false in all states of some (likewise infinite) suffix subinterval.

Proof of validity of implication (68). Let σ be some interval, and let σ' the interval resulting from projecting from σ using the formula $p \vee (\mathit{inf} \wedge \Box\neg p)$. Now if σ satisfies implication (68)'s antecedent, then the projected interval has some state s satisfying $\neg p$. The left-hand operator of Π in implication (68)'s antecedent ensures that interval σ's suffix starting with this state s satisfies the projection formula $p \vee (\mathit{inf} \wedge \Box\neg p)$. Therefore, s's global suffix in σ is infinite and *all* the global states in this suffix falsify p. Hence, the interval σ's suffix starting at each of *these* successor states likewise satisfies the conjunction $\mathit{inf} \wedge \Box\neg p$, so also satisfies the left-hand operator $p \vee (\mathit{inf} \wedge \Box\neg p)$ of Π in both of implication (68)'s Π-subformulas. It follows from this that all of these states are included in the projected interval σ', so it is infinite and has as an infinite suffix satisfying the conjunction $\mathit{inf} \wedge \Box\neg p$. Therefore, σ' itself satisfies the LTL formula $\Diamond(\mathit{inf} \wedge \Box\neg p)$, and the global interval σ similarly satisfies implication (68)'s consequent. Hence, implication (68) is valid. □

The following valid formulas containing the universal projection operator show how the test $p \vee (\mathit{inf} \wedge \Box\neg p)$ affects the projected interval:

$$(p \vee (\mathit{inf} \wedge \Box\neg p)) \ \Pi^{\mathrm{u}} \ \Box(p \vee (\mathit{inf} \wedge \Box\neg p)) \qquad (p \vee (\mathit{inf} \wedge \Box\neg p)) \ \Pi^{\mathrm{u}} \ \Box(\neg p \ \supset \ \bigcirc\neg p) \ .$$

A process projected in this way which is non-fairly ignored eventually sees an infinite sequence of states each with $\neg p$. Before then, p is true in every projected state. Therefore, within the projected time it is possible that p is *always true* or eventually *continuously false infinitely long*, but, as already noted, there are never *finite* projected sequences of states in which p is somewhere false.

As the next proposition indicates, non-fair projection is compatible with stutter invariance:

Proposition 5.4 (Non-fair projection and stutter invariance). If A is a stutter-invariant formula, then so is the Π-formula $(p \vee (\mathit{inf} \wedge \Box\neg p)) \ \Pi \ A$.

Proof. Observe that the left-hand operand of the Π-formula can be expressed as an equivalent disjunction separating between behaviour in finite time and infinite time:

$$(\mathit{finite} \ \wedge \ p) \ \vee \ \big(\mathit{inf} \ \wedge \ (p \ \vee \ \Box\neg p)\big) \ . \tag{69}$$

The two subformulas p and $p \vee \Box\neg p$ are both in LTL$-\bigcirc$, so disjunction (69) is stutter-invariant by Proposition 3.10. We already assume that the formula A is stutter-invariant, so by Proposition 3.14 the original Π-formula is as well. □

We already mentioned in Sect. 3 that the valid equivalences in (15)–(17) for eliminating an LTL construct permit the left-hand operand of Π to be an arbitrary formula, but the last two valid equivalences in (18) for chop and chop-star are not sound if an arbitrary formula is used on the left-hand side of Π instead of the state formula w. The following proposition is a pleasant surprise showing valid elimination equivalences for chop and chop-star in non-fair Π-formulas:

Proposition 5.5. The equivalences below, which are variants of the valid ones in (18) for eliminating Π but instead with the disjunction $p \vee (\mathit{inf} \wedge \Box\neg p)$ on the left side of Π-formulas, are themselves valid:

$$(p \vee (\mathit{inf} \wedge \Box\neg p)) \ \Pi \ (A;B) \quad \equiv \quad \big((p \vee (\mathit{inf} \wedge \Box\neg p)) \ \Pi \ A\big); \big(p \ \wedge \ (p \vee (\mathit{inf} \wedge \Box\neg p)) \ \Pi \ B\big) \tag{70}$$

$$(p \vee (\mathit{inf} \wedge \Box\neg p)) \ \Pi \ (A^*) \quad \equiv \quad (\mathit{halt}\, p); \big(p \ \wedge \ ((p \vee (\mathit{inf} \wedge \Box\neg p)) \ \Pi \ A \ \wedge \ \mathit{fin}\, p)^*; \circledcirc\Box\neg p\big) \ . \tag{71}$$

Hence, these equivalences can be used with the other such equivalences (15)–(17) to eliminate any non-fair Π-formulas from a PITL+Π formula.

Proof. Observe that the following implication about finite time is valid for any formula C:

$$\mathit{finite} \quad \supset \quad (p \vee (\mathit{inf} \wedge \Box\neg p))\ \Pi\ C \quad \equiv \quad p \ \wedge\ (p\ \Pi\ C) \ \wedge\ \mathit{fin}\, p \ . \tag{72}$$

The formula $(p \vee (\mathit{inf} \wedge \Box\neg p))\ \Pi\ (A;B)$ is expressible as a disjunction separating between the two cases when the subformula A does terminates and when it does not (i.e., $(\mathit{finite} \wedge A);B$ and $\mathit{inf} \wedge A$):

$$\big(\mathit{finite} \wedge p \wedge (p\ \Pi\ A) \wedge \mathit{fin}\, p\big); \big((p \vee (\mathit{inf} \wedge \Box\neg p))\ \Pi\ B\big) \quad \vee \quad \big(\mathit{inf} \ \wedge\ (p \vee (\mathit{inf} \wedge \Box\neg p))\ \Pi\ A\big) \ . \tag{73}$$

The first operand of the chop formula in (73) is then equivalent (using (72)) to $\mathit{finite} \wedge (p \vee (\mathit{inf} \wedge \Box\neg p))\ \Pi\ A$. The two disjuncts are then combined using properties of chop for finite and infinite time to obtain the next formula:

$$\big((p \vee (\mathit{inf} \wedge \Box\neg p))\ \Pi\ A \ \wedge\ \mathit{fin}\, p\big); \big((p \vee (\mathit{inf} \wedge \Box\neg p))\ \Pi\ B\big) \ .$$

This is equivalent the one below which has the effect of $\mathit{fin}\, p$ in the chop's left operand moved over to the right operand:

$$\big((p \vee (\mathit{inf} \wedge \Box\neg p))\ \Pi\ A\big); \big(p \ \wedge\ (p \vee (\mathit{inf} \wedge \Box\neg p))\ \Pi\ B\big) \ .$$

Hence, equivalence (70) is indeed valid. The proof for (71) involving chop-star is similar because all but the last of any iterations are in finite time which permits using valid implication (72). □

We now define a version of $|||\_$ for non-fairness using the projection operator Π:

$$A\ ||||_p\ B \quad \widehat{=} \quad (p \vee (\mathit{inf} \wedge \Box\neg p))\ \Pi\ A^{p}_{\mathit{active}} \ \wedge\ (\neg p \vee (\mathit{inf} \wedge \Box p))\ \Pi\ B^{\neg p}_{\mathit{active}} \ .$$

As can be seen here, the interleaving of two processes is quite similar that involving the operator $|||\_$ but needs to employ the variable *active* to deal with non-fairness.

Here is a valid implication showing the equivalence of fair and non-fair projection in both finite intervals and in suitable infinite intervals where the fair projection operator's left operand p is always eventually true in the global interval:

$$(\mathit{inf} \supset \Box\Diamond p) \quad \supset \quad p\ \Pi\ A \ \equiv \ (p \vee (\mathit{inf} \wedge \Box\neg p))\ \Pi\ A \ . \tag{74}$$

The implication's antecedent $\mathit{inf} \supset \Box\Diamond p$ ensures that the process represented by subformula A is always eventually scheduled even if its local projected interval is infinite. This is only needed for infinite time because in finite-time intervals (where the semantics of Π guarantee fair scheduling here) the disjunct $\mathit{inf} \wedge \Box\neg p$ is false and hence ignored, so the left-hand operands of the two Π-subformulas in (74) are semantically equivalent.

5.3. Synchronous transitions and communication in terms of Π

Binary $|||$, as defined in (32), only captures interleaved parallel composition of processes which share no steps and therefore synchronize exclusively through shared variables. A variant $|||_H$ of $|||$ with a set of actions H which are shared between the operands as an additional parameter on labelled transitions systems is considered in [BK08, p. 48]. In this section we first suitably extend the PITL definition of $|||$ to achieve the expressive power of $|||_H$.[4] Then we illustrate the use of $|||_H$ by extending our set of PITL+Π-defined imperative constructs to include CSP-style communication, or, equivalently, communication over channels of capacity 0 as in Promela, the input language of the SPIN model checker [Hol03]. The implementation follows the readiness model for communication of Olderog and Hoare [OH83, OH86], with synchronous transitions taking place as soon as both sides are ready to communicate. Interestingly, synchronous transitions are employed in the semantics of the **await** construct of RGITL too, which is otherwise based on interleaving. *Blocked* processes perform (idle) transitions in parallel with non-blocked ones until eventually they become

[4] Note that the middle operand H of $A\ |||_H\ B$ is the set of synchronous transitions, and should not be confused with the p of $|||_p$ as defined in (27).

enabled again [STE+14]. Finite H are sufficient to model CSP-style communication in case the data transmitted is from a finite domain, or in the degenerate case of communication for the sake of synchronization only, with no data transmitted.

In [BK08], $|||_H$ operates on labelled transition systems and H is the set of synchronized transitions. In our setting, the same meaning for $|||_H$ can be achieved by attaching the action names a of labelled transitions $s \xrightarrow{a} s'$ to the respective destination states s'. This means that a $s \xrightarrow{a} s'$ is represented as the set of unlabelled transitions $\langle s, b\rangle \to \langle s', a\rangle$, where b can be any action. Furthermore, action names are added to the propositional vocabulary and $\langle s, a\rangle \models b$ is defined to be true iff $b = a$. Consequently, $\sigma \models \bigcirc a$ means that the transition from σ^0 to σ^1 is labelled by a, and H can be represented by the state formula $\bigwedge_{a\in H} a$, which, by abuse of notation, we call H too. In this setting we can put

$$A_1 \,|||_H\, A_2 \quad \widehat{=} \quad \exists p.\ ((H \vee p)\ \Pi\ A_1 \ \wedge\ (H \vee \neg p)\ \Pi\ A_2)\ .$$

In words, states which are reached by shared actions appear in the projected intervals for both A_1 and A_2, which means that both A_1 and A_2 can impose conditions on such states.

Next we show how $|||_H$ can be used to define parallel composition for processes with communication over channels. Given channel c, $c?x$ and $c!e$ are implemented in terms of two atomic propositions r_c and w_c, which indicate readiness to read from and write to c, respectively, and a variable v_c for the datum to be transmitted. Given a process A, we write $I(A)$ and $O(A)$ for the sets of A's input and output channels, respectively. Channels used for internal communication between A's parallel components are excluded from $I(A)$ and $O(A)$, that is,

$$I(A_1 \,|||\, A_2) = (I(A_1) \setminus O(A_2)) \cup (I(A_2) \setminus O(A_1)),\ \ O(A_1 \,|||\, A_2) = (O(A_1) \setminus I(A_2)) \cup (O(A_2) \setminus I(A_1)),$$

and $I(A) \cap O(A) = \emptyset$ is required for A to be well-formed. Given some sets of channels I and O, we write

$$\mathit{notReady}(I, O) \quad \widehat{=} \quad \bigwedge_{c\in I} \Big(\neg r'_c \ \wedge\ (w'_c \equiv w_c)\Big) \ \wedge\ \bigwedge_{c\in O} \Big(\neg w'_c \ \wedge\ (r'_c \equiv r_c) \ \wedge\ v'_c = v_c\Big)\ .$$

to express that none of the channels from I and O is ready for input or, respectively, output. Furthermore, we write

$$\mathit{const}(X, C) \quad \widehat{=} \quad \bigwedge_{x\in X} x' = x \ \wedge\ \bigwedge_{c\in C} \Big((w'_c \equiv w_c) \ \wedge\ (r'_c \equiv r_c) \ \wedge\ v'_c = v_c\Big)\ .$$

to express that the variables from X remain unchanged, and so do the atomic propositions and the variables that implement the channels from C.

Let V and C stand for the set of all process variables and channels in use, respectively. Let $I_i = I(A_i)$ and $O_i = O(A_i)$, $i = 1, 2$, for the sake of brevity. We define occurrences of $c?x$ and $c!e$ in A_i as follows:

$$\begin{aligned}
c?x \ \stackrel{\text{def}}{=}\ & (\mathit{const}(V, C \setminus (I_i \cup O_i)) \ \wedge\ \mathit{notReady}(I_i \setminus \{c\}, O_i) \ \wedge\ \neg w_c \ \wedge\ \neg w'_c \ \wedge\ \neg r_c \ \wedge\ r'_c \ \wedge\ \mathit{skip});\\
& (\mathit{const}(V, C \setminus (I_i \cup O_i)) \ \wedge\ \mathit{notReady}(I_i \setminus \{c\}, O_i) \ \wedge\ r_c \ \wedge\ r'_c \ \wedge\ \neg w_c \ \wedge\ \neg w'_c \ \wedge\ \mathit{skip})^*;\\
& (\mathit{const}(V \setminus \{x\}, C \setminus (I_i \cup O_i)) \ \wedge\ \mathit{notReady}(I_i \setminus \{c\}, O_i)\\
& \qquad \wedge\ r_c \ \wedge\ w_c \ \wedge\ \neg r'_c \ \wedge\ \neg w'_c \ \wedge\ x' = v_c \ \wedge\ \mathit{skip})\\
& \vee\\
& (\mathit{const}(V \setminus \{x\}, C \setminus (I_i \cup O_i)) \ \wedge\ \mathit{notReady}(I_i \setminus \{c\}, O_i)\\
& \qquad \wedge\ w_c \ \wedge\ w'_c \ \wedge\ \neg r_c \ \wedge\ r'_c \ \wedge\ x' = v_c \ \wedge\ \mathit{skip});\\
& (\mathit{const}(V, C \setminus (I_i \cup O_i)) \ \wedge\ \mathit{notReady}(I_i \setminus \{c\}, O_i) \ \wedge\ r_c \ \wedge\ w_c \ \wedge\ \neg r'_c \ \wedge\ \neg w'_c \ \wedge\ \mathit{skip})
\end{aligned}$$

$$\begin{aligned}
c!e \ \stackrel{\text{def}}{=}\ & (\mathit{const}(V, C \setminus (I_i \cup O_i)) \ \wedge\ \mathit{notReady}(I_i, O_i \setminus \{c\})\\
& \qquad \wedge\ \neg r_c \ \wedge\ \neg r'_c \ \wedge\ \neg w_c \ \wedge\ w'_c \ \wedge\ v'_c = e \ \wedge\ \mathit{skip});\\
& (\mathit{const}(V, C \setminus (I_i \cup O_i)) \ \wedge\ \mathit{notReady}(I_i, O_i \setminus \{c\}) \ \wedge\ w_c \ \wedge\ w'_c \ \wedge\ \neg r_c \ \wedge\ \neg r'_c \ \wedge\ v'_c = e \ \wedge\ \mathit{skip})^*;\\
& (\mathit{const}(V, C \setminus (I_i \cup O_i)) \ \wedge\ \mathit{notReady}(I_i, O_i \setminus \{c\}) \ \wedge\ w_c \ \wedge\ r_c \ \wedge\ \neg w'_c \ \wedge\ \neg r'_c \ \wedge\ \mathit{skip})\\
& \vee\\
& (\mathit{const}(V, C \setminus (I_i \cup O_i)) \ \wedge\ \mathit{notReady}(I_i, O_i \setminus \{c\}) \ \wedge\ r_c \ \wedge\ r'_c \ \wedge\ \neg w_c \ \wedge\ w'_c \ \wedge\ v'_c = e \ \wedge\ \mathit{skip});\\
& (\mathit{const}(V, C \setminus (I_i \cup O_i)) \ \wedge\ \mathit{notReady}(I_i, O_i \setminus \{c\}) \ \wedge\ w_c \ \wedge\ r_c \ \wedge\ \neg w'_c \ \wedge\ \neg r'_c \ \wedge\ \mathit{skip})\ .
\end{aligned}$$

Furthermore, to take account of the use of channels, the definitions of assignment and *iframe* change as

follows:

$$x_1,\ldots,x_n := e_1,\ldots,e_n \quad \widehat{=} \quad \mathit{notReady}(I_i \cup O_i) \ \wedge \ \mathit{const}(V \setminus \{x_1,\ldots,x_n\}, C \setminus (I_i \cup O_i)) \ \wedge \ \bigwedge_{i=1}^{n} x_i' = e_i$$

$$\mathit{iframe} \quad \widehat{=} \quad \Box\Big(\bigwedge_{x\in V} x' = \bigcirc x \ \wedge \bigwedge_{c\in I_1\cup I_2\cup O_1\cup O_2} w_c' \equiv \bigcirc w_c \ \wedge \ r_c' \equiv \bigcirc r_c \ \wedge \ v_c' = \bigcirc v_c\Big) \ .$$

Informally $c?x$ first indicates readiness for input by setting r_c. Then it spins on w_c until w_c, which indicates readiness of the partner process for output, is set. As soon as this happens, the datum to be transmitted is copied form v_c to x, and the readiness variables are cleared. The second disjunct is about the possibility that w_c is already set when $c?x$ starts execution. Then communication proceeds in the same manner without delay. Of course, if the partner process never becomes ready, spinning on w_c continues indefinitely. Complementarily, $c!e$ first copies the value of e to v_c and sets w_c. Then it spins on r_c until that becomes true indicating that the datum has been received. Finally readiness variables are cleared. The second disjunct is about communication without delay, in case r_c is set from the beginning or the execution of $c!e$.

Now parallel composition of processes which communicate along 0-capacity channels can be defined by putting

$$A_1 \mid\mid\mid A_2 \quad \widehat{=} \quad A_1 \mid\mid\mid_H A_2, \text{ where } H \quad \widehat{=} \quad \bigvee_{c\in I_1\cap O_2\cup I_2\cap O_1} r_c \ \wedge \ w_c \ .$$

In words, steps of A_1 and A_2 in which both processes are ready to communicate through some shared channel are synchronized. Synchronizing steps at which $w_c \wedge r_c$ holds between two processes with occurrences of $c!e$ and $c?x$, respectively, guarantees that v_c has the correct value of the e from $c!e$ at the time it is copied to the x of $c?x$. Expectedly $A_1 \mid\mid\mid_H A_2$ boils down to the usual definition (32) for A_1 and A_2 which do not communicate.

5.4. Comparison of state projection with time-step projection

Somewhat after Π was introduced in [HMM83, Mos83, MM84], another binary ITL operator was proposed in [Mos86] (see also [Mos95, KM08]) 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,

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

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 $A \mathit{\ proj\ true}$, 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 ability for *proj* to express chop-star yields that even if A and B are stutter-invariant formulas (as discussed earlier in Sect. 3.2), $A \mathit{\ proj\ } B$ might not have this property. An example is *true proj false*, which is equivalent to the formula *empty* and so not stutter-invariant (see Sect. 3.2).

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

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

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 [Mos86], whereas Π is best fit for interleaving concurrency. A variant of *proj* for projecting from real to discrete time has been studied in [He99, Gue04a]. The notation used is slightly different: a projection formula's syntax there has the form $A \backslash\backslash B$ instead of $A \mathit{\ proj\ } B$.

When its left operand is a state formula, the projection operator Π and, consequently, parallel composition $\mid\mid\mid\_$, can be expressed using *proj*:

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

Conversely, *proj* can be defined using Π and propositional quantification, which, as noted in Sect. 2, does

not add expressiveness to PITL:

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

where the propositional variable p does not occur in A or B.

6. Related work

6.1. Projection in the Duration Calculus

Dang [Dan99] proposed for the Duration Calculus (DC) [ZHR91, ZH04, OD08] a real-time version of the projection operator Π written / to reason about interleaving concurrency in hybrid systems. (Note that / reverses the two operands's order from that of other projection operators such as Π, *proj* and \\ already discussed, but the sampling operator @ mentioned shortly in Sect. 6.2 does this as well.) 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 [GD02] 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 [GD04]. In [Gue04b], / 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.

6.2. Other kinds of temporal projection

Several research groups have proposed and studied other forms of temporal projection for use with extensions of ITL such as Projection Temporal Logic [Dua96, DKH94, DYK08] and RGITL [BBN+10, BSTR11]. RGITL, which combines Jones' *Rely-Guarantee Conditions* [Jon83] with ITL, assumes interleaving with projected time. It has concurrency operators akin to ||| but defined without using an explicit projection operator. These are, as the authors acknowledge, much more complicated to handle. RGITL has been used extensively to reason about interleaved concurrent programs in the KIV proof verifier. Maybe Π can help elucidate RGITL's operators.

The originators of both RGITL and Projection Temporal Logic acknowledge that our book [Mos86] on representing sequential and concurrent programs directly in ITL has been a source of inspiration. For example, KIV proof verifier can *symbolically execute* concurrent programs expressed in RGITL in a manner partially influenced by the methods suggested in [Mos86] for executing programs suitably expressed in ITL. It is no surprise that the framework we present here for expressing concurrent programs with and without Π likewise builds on this.

Our new approach aims to avoid as much as possible the need to introduce additional primitive temporal constructs (such as RGITL's 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. [JHC15] observe 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. [NRZ+15] at Amazon Web Services with successfully specifying and verifying subtle industrial-strength concurrent algorithms using Lamport's TLA$^+$ [Lam02] 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. [EFH+03, EF14] have developed LTL$^{@}$, which adds a *clock operator* to LTL to deal with time granularities in hardware systems. This construct is included in the industrial standards Property Specification Language (PSL, IEEE Standard 1850 [IEE10]) [EF06] and SystemVerilog Assertions (SVA, in IEEE Standard 1800 [IEE12]) [CDHK15]. The clock operator adds succinctness but not expressiveness and is its own dual. It requires modifying the semantics of formulas. In particular, the definition of $\models$ in LTL$^{@}$ includes not just a state sequence but also a *clock formula*, which can be any boolean formula. The semantics of a formula A is described using $\sigma, c \models A$ instead of $\sigma \models A$, where c is such a clock formula. In what follows, we employ three notations below to clearly distinguish between the different logics LTL, LTL+Π and LTL$^{@}$:

$$\sigma \models_{\mathrm{LTL}} A \qquad \sigma \models_{\mathrm{LTL}+\Pi} A \qquad \sigma, c \models_{\mathrm{LTL}^{@}} A \ .$$

Note that for any LTL formula, the first two of these are actually equivalent. Let p and q be two propositional variables. LTL$^{@}$ defines $\sigma, p \models_{\mathrm{LTL}^{@}} q$ to hold iff p is true somewhere in σ and the first such state also satisfies q. More generally, for any purely LTL formula A (i.e., one without instances of @ or Π), the LTL$^{@}$ formula $A@p$ acts like $p \,\Pi\, A$ in LTL+Π:

$$\sigma, p \models_{\mathrm{LTL}^{@}} A \quad \text{iff} \quad \sigma \models_{\mathrm{LTL}+\Pi} p \,\Pi\, A \ .$$

However, in LTL$^{@}$ the original state sequence is still completely accessible within A, as is further explained shortly. The LTL$^{@}$ temporal formula $A@c1$ permits evaluating the subformula A with a clock $c1$ determining which states to sample:

$$\sigma, c \models_{\mathrm{LTL}^{@}} A@c1 \quad \text{iff} \quad \sigma, c1 \models_{\mathrm{LTL}^{@}} A \ .$$

For instance, if A is of the form $B@\mathit{true}$, then the following chain-reasoning holds if B itself does not contain any @-constructs:

$$\sigma, c \models_{\mathrm{LTL}^{@}} B@\mathit{true} \quad \text{iff} \quad \sigma, \mathit{true} \models_{\mathrm{LTL}^{@}} B \quad \text{iff} \quad \sigma \models_{\mathrm{LTL}} B \ .$$

Hence, the evaluation of B here ignores the clock c. It follows that the next valid LTL$^{@}$ equivalence concerning two clocks $c1$ and $c2$ holds for *any* LTL$^{@}$ formula A:

$$(A@c1)@c2 \quad \equiv \quad A@c1 \ .$$

In contrast, the PITL+Π formulas $p \,\Pi\, A$ and $q \,\Pi\, (p \,\Pi\, A)$ have quite distinct semantics. As a consequence, the next equivalence is *not* valid in PITL+Π:

$$q \,\Pi\, (p \,\Pi\, A) \quad \equiv \quad p \,\Pi\, A \ .$$

The developers of LTL$^{@}$ point out in [EFH+03, p. 858] that the use of the term 'projection' for the clock operator @ in LTL$^{@}$ and other standards which have adapted @ is imprecise since states in between the projected ones are still fully accessible (unlike for Π):

> Actually, referring to a projection of the path is not precisely correct, as we allow access to states in between consecutive states of a projection in the event of a clock switch. However, the word 'projection' conveys the intuitive function of the clock operator in the case that the formula is singly-clocked. Use of the word 'projection' when describing the clocks of Sugar2.0 and ForSpec ... is similarly imprecise.

Indeed, their book [EF06] on PSL makes no mention of the term 'projection' and instead uses 'sampling', but their forthcoming book chapter [EF14] does refer to 'projection' a few times. A similar construct called the *sampling operator* is found in *temporal 'e'* (part of IEEE Standard 1647 [IEE11] and influenced by ITL [Mor99, HMN01]).

The operator $|||\_$ is related to the shuffle operator investigated by Gischer [Gis81, Gis88] for expressing regular languages involving interleaving. Such an interleaving operator is available in the markup language *Standard Generalized Markup Language* (SGML) [ISO86] as well as in OWL-S, which is used in semantic markup for Web Services [OWL04]. Gelade [Gel10] and Peng et al. [PCM15, PC15] have recently analysed regular expressions extended with interleaving.

Katz and Peled introduced *Interleaving Set Temporal Logic* (ISTL) [KP87, KP90, KP92] to reason about distributed computations. The logic is related to branching time temporal logic and uses sets of triples to model events as branching structures capturing all interleavings. However, there is no interleaving operator such as $|||\_$.

6.3. Work relating stuttering invariance to regular expressions and interval temporal logics

Dax et al. [DKL09] have developed specification languages for stutter-invariant properties naturally associated with regular and ω-regular languages (LTL only expresses star-free languages, as noted above in Sect. 3.2). One of the specification languages is intended to be a practical variant of conventional regular expressions. Another is a variant of the core of the industrial standard Property Specification Language (PSL, IEEE Standard 1850 [IEE10, EF06]), and therefore called *Stutter-Invariant PSL*. The conventional concatenation and Kleene-star constructs for regular expressions do not ensure stutter invariance. Alternatives are used which instead *fuse* strings. For example, one of these is an analogue of the derived PITL construct *chop-plus* (A^+ denotes $A; A^*$).

Yang and Duan [YD10] discuss the use of stutter invariance in a propositional version of Projection Temporal Logic (itself already mentioned above in Sect. 6.2) and present a sublogic called *Stutter-invariant Propositional Projection Temporal Logic* ($\mathrm{PPTL}_{\mathrm{st}}$). Their notion of stutter invariance permits a finite state sequence and an infinite state sequence to be stutter-equivalent, so differs from the convention we discuss above in Definition 3.7. Hence, our convention of regarding the constructs *finite* and *inf* as well as formulas of the form *fin* w to all be stutter-invariant would not apply. Furthermore, Yang and Duan employ $\mathrm{PPTL}_{\mathrm{st}}$ as a specific syntactic class of stutter-invariant formulas for specifying properties unlike our approach of relating various naturally arising LTL formulas with $\bigcirc$ operators to equivalent formulas in LTL$-\bigcirc$. No mention is made of Peled and Wilke's Theorem 3.11 [PW97] that all stutter-invariant LTL properties can be expressed in LTL$-\bigcirc$.

Conclusions

We have explored new uses of the oldest known projection operator Π for ITL and also its relationship with other temporal constructs, in particular briefly examining Π's uncanny resemblance to the standard binary until-operator $\mathcal{U}$ in certain respects concerning expressiveness. This and the natural links to stutter invariance suggest that the projection operator Π, especially in combination with LTL, could have some practical value in model checking. In future work, we would like to further our understanding of Π's expressiveness, apply our projection-based approach to larger concurrent applications and investigate the possibility for tool support. This research could include a more extensive evaluation of the merits of the approaches presented here for formalising concurrency in either LTL or ITL with or without projection, and also with and without the assumption of fairness. A systematic evidence-based comparison of the standard transition-based style of reasoning commonly advocated for LTL and a less familiar idiom based the interaction between global time and the projected local time in which individual interleaved processes operate seems worthwhile. This requires commitment and investment but could yield long-range benefits. Our plans furthermore include exploring formal connections with RGITL as well as clocked-based logics such as $\mathrm{LTL}^{@}$ (all mentioned in Sect. 6).

Acknowledgments

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

References

[BA06] Mordechai Ben-Ari. *Principles of Concurrent and Distributed Programming.* Addison-Wesley, Harlow, England, second edition, 2006.

[BBN+10] Simon Bäumler, Michael Balser, Florian Nafz, Wolfgang Reif, and Gerhard Schellhorn. Interactive verification of concurrent systems using symbolic execution. *AI Commun.*, 23(2–3):285–307, 2010.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking.* MIT Press, Cambridge, MA, USA, 2008.

[BSTR11] Simon Bäumler, Gerhard Schellhorn, Bogdan Tofan, and Wolfgang Reif. Proving linearizability with temporal logic. *Formal Aspects of Computing*, 23(1):91–112, 2011.

[CDHK15] E. Cerny, S. Dudani, J. Havlicek, and D. Korchemny. *SVA: The Power of Assertions in SystemVerilog.* Springer, Cham, Switzerland, second edition, 2015.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[Che80] Brian F. Chellas. *Modal Logic: An Introduction*. Cambridge University Press, Cambridge, England, 1980.

[Dan99] 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.

[DG08] Volker Diekert and Paul Gastin. First-order definable languages. In Jörg Flum, Erich Grädel, and Thomas Wilke, editors, *Logic and Automata: History and Perspectives*, volume 2 of *Texts in Logic and Games*, pages 261–306. Amsterdam University Press, Amsterdam, 2008.

[DKH94] Zhenhua Duan, Maciej Koutny, and Chris Holt. Projection in temporal logic programming. In Frank Pfenning, editor, *LPAR '94*, volume 822 of *LNCS*, pages 333–344. Springer, Berlin Heidelberg, 1994.

[DKL09] Christian Dax, Felix Klaedtke, and Stefan Leue. Specification languages for stutter-invariant regular properties. In Zhiming Liu and Anders P. Ravn, editors, *7th Int'l Symp. Automated Technology for Verification and Analysis (ATVA 2009)*, volume 5799 of *LNCS*, pages 244–254. Springer, Berlin Heidelberg, 2009.

[dRdBH+01] Willem-Paul de Roever, Frank de Boer, Ulrich Hanneman, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, Cambridge, England, 2001.

[Dua96] Zhenhua Duan. *An Extended Interval Temporal Logic and a Framing Technique for Temporal Logic Programming*. PhD thesis, Dept. of Computing Science, Newcastle University, Newcastle upon Tyne, England, 1996. Tech. rep. 556.

[DYK08] Zhenhua Duan, Xiaoxiao Yang, and Maciej Koutny. Framed temporal logic programming. *Science of Computer Programming*, 70(1):31–61, 2008.

[EF06] Cindy Eisner and Dana Fisman. *A Practical Introduction to PSL*. Springer, New York, NY, USA, 2006.

[EF14] Cindy Eisner and Dana Fisman. Temporal logic made practical. `http://www.cis.upenn.edu/~fisman/documents/EF_HBMC14.pdf`, 2014. Accessed 4 June 2015. Expected to appear in 2018 in *Handbook of Model Checking*, editors: Edmund M. Clarke, Thomas A. Henzinger and Helmut Veith. Springer, Cham, Switzerland.

[EFH+03] Cindy Eisner, Dana Fisman, John Havlicek, Anthony McIsaac, and David Van Campenhout. The definition of a temporal clock operator. In Jos C.M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *ICALP 2003*, volume 2719 of *LNCS*, pages 857–870. Springer, Berlin Heidelberg, 2003.

[Eme90] E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 16, pages 995–1072. Elsevier/MIT Press, Amsterdam, 1990.

[GD02] Dimitar P. Guelev and Dang Van Hung. Prefix and projection onto state in duration calculus. *Electr. Notes Theor. Comput. Sci.*, 65(6):101–119, 2002.

[GD04] Dimitar P. Guelev and 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.

[Gel10] Wouter Gelade. Succinctness of regular expressions with interleaving, intersection and counting. *Theoretical Computer Science*, 411(31-33):2987–2998, 2010.

[Gis81] Jay Gischer. Shuffle languages, Petri nets, and context-sensitive grammars. *Communications of the ACM*, 24(9):597–605, 1981.

[Gis88] Jay Gischer. The equational theory of pomsets. *Theoretical Computer Science*, 61:199–224, 1988.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. Springer, Berlin Heidelberg, 1996.

[Gue04a] Dimitar P. Guelev. A complete proof system for first-order interval temporal logic with projection. *J. Log. Comput.*, 14(2):215–249, 2004.

[Gue04b] Dimitar P. Guelev. Logical interpolation and projection onto state in the Duration Calculus. *J. Applied Non-Classical Logics*, 14(1–2):181–208, 2004.

[GW91a] Patrice Godefroid and Pierre Wolper. A partial approach to model checking. In *Sixth Annual Symposium on Logic in Computer Science (LICS '91)*, pages 406–415. IEEE Computer Society, Los Alamitos, CA, USA, 1991.

[GW91b] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In Kim Guldstrand Larsen and Arne Skou, editors, *3rd International Workshop on Computer Aided Verification (CAV '91)*, volume 575 of *LNCS*, pages 332–342. Springer, Berlin Heidelberg, 1991.

[GW93] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, 1993.

[HC96] George E. Hughes and Max J. Cresswell. *A New Introduction to Modal Logic*. Routledge, London, 1996.

[He99] Jifeng He. A behavioral model for co-design. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99, Vol. II*, volume 1709 of *LNCS*, pages 1420–1438. Springer, Berlin Heidelberg, 1999.

[HMM83] J. Halpern, Z. Manna, and B. Moszkowski. A hardware semantics based on temporal intervals. In J. Diaz, editor, *ICALP 1983*, volume 154 of *LNCS*, pages 278–291. Springer, Berlin Heidelberg, 1983.

[HMN01] Yoav Hollander, Matthew Morley, and Amos Noy. 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*, pages 41–50. IEEE Computer Society, Los Alamitos, CA, USA, 2001.

[Hol03] Gerard Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, Boston, MA, USA, 2003.

[IEE10] IEEE. *Standard for Property Specification Language (PSL), Standard 1850-2010*. ANSI/IEEE, New York, 2010.

[IEE11] IEEE. *Standard for the Functional Verification Language e, Standard 1647-2011*. ANSI/IEEE, New York, 2011.

[IEE12] IEEE. *Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language, Standard 1800-2012*. ANSI/IEEE, New York, 2012.

[ISO86] ISO. *Standard Generalized Markup Language (SGML): ISO 8879:1986*. International Organization for Standardization, Geneva, Switzerland, 1986.

[ITL] ITL web pages. `http://www.antonio-cau.co.uk/ITL/`. Accessed 8 June 2015.

[JHC15] Cliff B. Jones, Ian J. Hayes, and Robert J. Colvin. Balancing expressiveness in formal approaches to concurrency. *Formal Asp. Comput.*, 27(3):475–497, 2015.

[Jon83] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, October 1983.

[Kel76] Robert M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976.

[KM08] Fred Kröger and Stephan Merz. *Temporal Logic and State Systems*. Texts in Theoretical Computer Science (An EATCS Series). Springer, Berlin Heidelberg, 2008.

[KP87] Shmuel Katz and Doron Peled. Interleaving Set Temporal Logic. In *Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC '87)*, pages 178–190. ACM, 1987.

[KP90] Shmuel Katz and Doron A. Peled. Interleaving Set Temporal Logic. *Theoretical Computer Science*, 75(3):263–287, 1990.

[KP92] Shmuel Katz and Doron A. Peled. Verification of distributed programs using representative interleaving sequences. *Distributed Computing*, 6(2):107–120, 1992.

[Lam83] L. Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Information Processing 83*, pages 657–668. North-Holland, 1983.

[Lam02] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, Boston, MA, USA, 2002.

[LPZ85] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In R. Parikh, editor, *Logics of Programs*, volume 193 of *LNCS*, pages 196–218. Springer, Berlin Heidelberg, 1985.

[MG15] Ben Moszkowski and Dimitar P. Guelev. An application of temporal projection to interleaving concurrency. In Xuandong Li, Zhiming Liu, and Wang Yi, editors, *Dependable Software Engineering: Theories, Tools, and Applications – First International Symposium (SETTA 2015)*, volume 9409 of *LNCS*, pages 153–167. Springer, Cham, Switzerland, 2015.

[MM84] Ben Moszkowski and Zohar Manna. Reasoning in Interval Temporal Logic. In E. M. Clarke and D. Kozen, editors, *Proc. Workshop on Logics of Programs, Pittsburgh, PA, June, 1983*, volume 164 of *LNCS*, pages 371–382. Springer, Berlin Heidelberg, 1984.

[Mor99] Matthew J. Morley. Semantics of temporal $\boldsymbol{e}$. In T. F. Melham and F. G. Moller, editors, *Banff'99 Higher Order Workshop: Formal Methods in Computation, Ullapool, Scotland, 9–11 Sept. 1999*, pages 138–142. Technical Report, Department of Computing Science, University of Glasgow, Glasgow, Scotland, 1999.

[Mos83] B. Moszkowski. *Reasoning about Digital Circuits*. PhD thesis, Department of Computer Science, Stanford University, June 1983. Tech. rep. STAN–CS–83–970.

[Mos86] B. Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, Cambridge, England, 1986.

[Mos95] Ben Moszkowski. Compositional reasoning about projected and infinite time. In *Proc. 1st IEEE Int'l Conf. on Engineering of Complex Computer Systems (ICECCS'95)*, pages 238–245. IEEE Computer Society, Los Alamitos, CA, USA, 1995.

[Mos00] Ben Moszkowski. An automata-theoretic completeness proof for Interval Temporal Logic (extended abstract). In U. Montanari, J. Rolim, and E. Welzl, editors, *Proc. 27th Int'l. Colloquium on Automata, Languages and Programming (ICALP 2000)*, volume 1853 of *LNCS*, pages 223–234. Springer, Berlin Heidelberg, 2000.

[Mos04] Ben Moszkowski. A hierarchical completeness proof for Propositional Interval Temporal Logic with finite time. *J. Applied Non-Classical Logics*, 14(1–2):55–104, 2004.

[Mos12] Ben Moszkowski. A complete axiom system for propositional Interval Temporal Logic with infinite time. *Log. Meth. Comp. Sci.*, 8(3:10):1–56, 2012.

[Mos13] Ben Moszkowski. Interconnections between classes of sequentially compositional temporal formulas. *Information Processing Letters*, 113(9):350–353, 2013.

[Mos14] Ben Moszkowski. Compositional reasoning using intervals and time reversal. *Ann. Math. Artif. Intell.*, 71(1-3):175–250, 2014.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specifications*. Springer, New York, 1992.

[NRZ+15] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services uses formal methods. *Communications of the ACM*, 58(4):66–73, 2015.

[OD08] Ernst-Rüdiger Olderog and Henning Dierks. *Real-Time Systems: Formal Specification and Automatic Verification*. Cambridge University Press, Cambridge, England, 2008.

[OH83] Ernst-Rüdiger Olderog and C. A. R. Hoare. Specification-oriented semantics for communicating processes. In Josep Díaz, editor, *Automata, Languages and Programming, 10th Colloquium (ICALP 1983)*, volume 154 of *LNCS*, pages 561–572. Springer, Berlin Heidelberg, 1983.

[OH86] Ernst-Rüdiger Olderog and C. A. R. Hoare. Specification-oriented semantics for communicating processes. *Acta Inf.*, 23(1):9–66, 1986.

[OWL04] OWL-S: Semantic markup for web services. `http://www.w3.org/Submission/OWL-S/`, 2004. Accessed 14 March 2016.

[PC15] Feifei Peng and Haiming Chen. Discovering restricted regular expressions with interleaving. In Reynold Cheng, Bin Cui, Zhenjie Zhang, Ruichu Cai, and Jia Xu, editors, *17th Asia-PacificWeb Conf. on Web Technologies and Applications (APWeb 2015)*, volume 9313 of *LNCS*, pages 104–115. Springer, Cham, Switzerland, 2015.

[PCM15] Feifei Peng, Haiming Chen, and Xiaoying Mou. Deterministic regular expressions with interleaving. In Martin Leucker, Camilo Rueda, and Frank D. Valencia, editors, *12th International Colloquium on Theoretical Aspects of Computing (ICTAC 2015)*, volume 9399 of *LNCS*, pages 203–220. Springer, Cham, Switzerland, 2015.

[Pel93] Doron Peled. All from one, one for all: on model checking using representatives. In Costas Courcoubetis, editor, *5th International Conference on Computer Aided Verification (CAV '93)*, volume 697 of *LNCS*, pages 409–423. Springer, Berlin Heidelberg, 1993.

[Pel96] Doron Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in System Design*, 8(1):39–64, 1996.

[Pet81] Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[PW97] Doron Peled and Thomas Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Information Processing Letters*, 63(5):243–246, 1997.

[STE+14] Gerhard Schellhorn, Bogdan Tofan, Gidon Ernst, Jörg Pfähler, and Wolfgang Reif. RGITL: A temporal logic framework for compositional reasoning about interleaved programs. *Ann. Math. Artif. Intell.*, 71(1-3):131–174, 2014.

[Tau06] Gadi Taubenfeld. *Synchronization Algorithms and Concurrent Programming*. Pearson/Prentice Hall, Harlow, England, 2006.

[Val91] Antti Valmari. Stubborn sets for reduced state space generation. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990*, volume 483 of *LNCS*, pages 491–515. Springer, Berlin Heidelberg, 1991.

[Val92] Antti Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1(4):297–322, 1992.

[YD10] Chen Yang and Zhenhua Duan. Compositional verification with stutter-invariant Propositional Projection Temporal Logic. In *Proceedings of the 14th WSEAS International Conference on Computers: Part of the 14th WSEAS CSCC Multiconference - Volume I*, ICCOMP'10, pages 272–280, Stevens Point, Wisconsin, USA, 2010. World Scientific and Engineering Academy and Society (WSEAS).

[ZH04] Zhou Chaochen and Michael R. Hansen. *Duration Calculus: A Formal Approach to Real-Time Systems*. Springer, Berlin Heidelberg, 2004.

[ZHR91] Zhou Chaochen, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.

A. Proofs of correctness for Peterson's algorithm

This appendix contains proofs concerning mutual exclusion and liveness for the version of Peterson's algorithm already presented in Fig. 2 in Sect. 4.7. The presentation first considers some relevant ways to reason about assignments to the program variables, then looks at mutual exclusion, and finally deals with liveness. Another variant of Peterson's algorithm without labels is then briefly examined.

A.1. Reasoning about assignments to the variables $flag_i$ and $turn$

We later show how to prove the safety (mutual exclusion) and liveness properties (40) and (45). In preparation for this, it is necessary to establish the following properties regarding the behaviour of the program variables $flag_0$, $flag_1$ and $turn$:

- Each process $Peterson_i$ never changes the value of the variable $flag_{1-i}$, that is, the other process $Peterson_{1-i}$ exercises exclusive control to alter $flag_{1-i}$.
- In contrast to the handling of $flag_0$ and $flag_1$, *both* processes can alter the variable $turn$. However, each process $Peterson_i$ always restricts itself to either leaving $turn$ unchanged or assigning it the value $1 - i$.

The following valid formulas are a first step towards establishing these properties:

$$\models \quad Peterson_i \quad \supset \quad \Box(flag'_{1-i} = flag_{1-i}) \tag{75}$$

$$\models \quad Peterson_i \quad \supset \quad \Box(turn' \in \{turn, 1-i\}) \ . \tag{76}$$

For example, the first of these expresses that process $Peterson_i$ maintains $flag'_{1-i}$ equal to $flag_{1-i}$, so that when framing is done in global time using *iframe*, the value of $flag_{1-i}$ is indeed never changed by $Peterson_i$. The implication's validity can be seen by sequentially composing similar valid formulas about the various statements in $Peterson_i$.

Implications (75) and (76) about each process operating in its *local* projected time are then readily

adapted to obtain the following valid ones concerning *global* time:

$$\models \quad (pid = i)\ \Pi\ Peterson_i \quad \supset \quad \Box(pid = i \ \supset \ flag'_{1-i} = flag_{1-i}) \tag{77}$$

$$\models \quad (pid = i)\ \Pi\ Peterson_i \quad \supset \quad \Box(pid = i \ \supset \ turn' \in \{turn, 1-i\})\ . \tag{78}$$

The next two valid implications concern framing achieved by *iframe* in global time and build on the valid implications (77) and (78), respectively:

$$\models \quad iframe \ \wedge \ (pid = i)\ \Pi\ Peterson_i \quad \supset \quad \Box\big(more \wedge pid = i \ \supset \ (\bigcirc flag_{1-i}) = flag_{1-i}\big) \tag{79}$$

$$\models \quad iframe \ \wedge \ (pid = i)\ \Pi\ Peterson_i \quad \supset \quad \Box\big(more \wedge pid = i \ \supset \ (\bigcirc turn) \in \{turn, 1-i\}\big)\ . \tag{80}$$

We now show how to prove the validity of implication (79):

Proof. Recall that the valid implication (77) given above concerns how the process $Peterson_i$ never alters the program variable $flag_{1-i}$ in global time, so the equality $flag'_{1-i} = flag_{1-i}$ holds in all global states where $pid = i$. It is not hard to then check the validity of the next implication which relates the values of $flag'_{1-i}$ and $\bigcirc flag_{1-i}$ in global time in states with $pid = i$ when framing is performed:

$$iframe \ \wedge \ \Box(pid = i \ \supset \ flag'_{1-i} = flag_{1-i}) \quad \supset \quad \Box(more \wedge pid = i \quad \supset \quad (\bigcirc flag_{1-i}) = flag_{1-i})\ .$$

The combination of implication (77) together with this one using simple propositional reasoning then yields the validity of implication (79). □

We now consider some general properties expressible in LTL+Π as valid implications and relating *global* and *local* framing, initialisation and interference (so by no means limited to Peterson's algorithm). The properties involve issues such as the following two, which themselves have associated valid implications concerning convenient sufficient conditions and presented in Propositions A.1 and A.2, respectively:

- *Noninterference with a program variable's initial value*: Here we want to ensure that under suitable circumstances a program variable's starting value in the first *global* state is preserved until the first *local* state of the appropriate process which relies on this initialisation. For example, in Peterson's algorithm the starting *global* state initialises the $flag_i$ to 0 for each process $Peterson_i$. It is necessary to show that when the first *local* projected state of the process $Peterson_i$ takes place, which can perhaps occur sometime much later, that the other process $Peterson_{1-i}$ has not in the meantime interfered with $flag_i$'s value by altering it.
- *Noninterference with a program variable's value at any time*: If a process never attempts to alter a program variable, then assignments by other processes are not subject to interference by this process. This can help permit a process to view a variable framed in *global* interval as also being framed in the *local* interval of that process. For example, as already discussed above, each process $Peterson_i$ never alters the variable $flag_{1-i}$, so the other process $Peterson_{1-i}$ can regard that variable as being framed in $Peterson_{1-i}$'s local interval. It is then easier to reason in projected time about the interference-free behaviour of $Peterson_{1-i}$ and $flag_{1-i}$.

For these purposes, it is useful at this stage to define a derived binary operator *istable* expressing that whenever the left-hand operand is true, the value of the right-hand operator does not change between the current and next states:

$$w_1\ istable\ w_2 \quad \widehat{=} \quad \Box\big((more \wedge w_1) \ \supset \ (\bigcirc w_2) \equiv w_2\big) \qquad \text{Interleaving Stability}$$

Consequently, when w_1 *istable* w_2 holds, any state with w_1 true does not interfere with w_2's value and passes it on to the next state. For example, the formula $(pid = i)$ *istable* $flag_{1-i}$ expresses that whenever $pid = i$, the value of $flag_{1-i}$ remains stable between the present state and the one immediately following it. One benefit of *istable* is that it can make formulas more concise. Another advantage is that w_1 *istable* w_2 is easily shown to be stutter-invariant by an analysis of its negation.

The next valid implication (81) states the following: Suppose the initial value of a program variable a in the first *global* state equals some constant j, and also that a is not altered between each state satisfying the state formula $\neg p$ and this state's immediate successor state. Then a's value is the same in the initial state and all states up to and including the first projected *local* state satisfying the state formula p, if the interval contains such a state.

$$a = j \ \wedge \ (\neg p)\ istable\ a \quad \supset \quad p\ \Pi^{\mathrm{u}}\ (a = j)\ . \tag{81}$$

Proof of validity of implication (81). Here is a version of the antecedent of this implication with *istable* replaced by its definition:

$$a = j \ \wedge \ \Box\big(\mathit{more} \wedge \neg p \ \supset \ (\bigcirc a) \equiv a\big) \ . \tag{82}$$

It is easy to check that if an interval σ satisfies conjunction (82), then the following is also true about σ: If p is ever true, then the first time it is true, so is $a = j$. Here is this expressed in LTL:

$$\Diamond p \quad \supset \quad \neg p \ \mathcal{U} \ (p \ \wedge \ a = j) \ . \tag{83}$$

Implication (83) is semantically equivalent to the formula $p \ \Pi^{\mathrm{u}} \ (a = j)$ because if p is ever true, then the first state of the interval obtained by projecting σ's states satisfying p itself satisfies $a = j$ as well. Hence, σ satisfies the consequent of implication (81), so the implication (81) itself is indeed valid. □

The valid implication below states that if a program variable a is framed and it is never assigned to in any of the states universally projected using the state formula p, then the global interval satisfies p *istable* a:

$$\mathit{iframe}(.a) \ \wedge \ p \ \Pi^{\mathrm{u}} \ \Box(a' = a) \quad \supset \quad p \ \mathit{istable} \ a \ . \tag{84}$$

Proof of validity of implication (84). Let σ be an interval satisfying implication (84)'s antecedent. The definition of $\mathit{iframe}(.a)$ is $\Box(\mathit{more} \supset (\bigcirc a) = a')$, and furthermore the projection formula $p \ \Pi^{\mathrm{u}} \ \Box(a' = a)$ is semantically equivalent to the following LTL formula:

$$\Box(p \ \supset \ (a' = a)) \ .$$

The combination of the two LTL formulas $\Box(\mathit{more} \supset (\bigcirc a) = a')$ and $\Box(p \supset (a' = a))$ together with the transitivity of equality ensures that σ also satisfies the LTL formula $\Box((\mathit{more} \wedge p) \supset (\bigcirc a) = a)$, which is the definition of p *istable* a, so implication (84) is valid. □

Below is valid implication (85) formalising *noninterference* with a program variable a's *initial value* by ensuring that if this variable a initially equals some constant j, is framed and never assigned to when $\neg p$ holds (i.e., $(\neg p) \ \Pi^{\mathrm{u}} \ \Box(a' = a)$, or equivalently $\Box(\neg p \ \supset \ a' = a)$), then this value j is correctly passed on to a in the first state projected using the state formula p, if such a state exists:

Proposition A.1 (Noninterference with a program variable's initial value). For any program variable a and propositional variable p, the next implication is valid:

$$a = j \ \wedge \ \mathit{iframe}(.a) \ \wedge \ (\neg p) \ \Pi^{\mathrm{u}} \ \Box(a' = a) \quad \supset \quad p \ \Pi^{\mathrm{u}} \ (a = j) \ . \tag{85}$$

Proof. The validity of implication (85) readily follows from the combination of the following substitution instance of valid implication (84) together with valid implication (81) and simple propositional reasoning:

$$\mathit{iframe}(.a) \ \wedge \ (\neg p) \ \Pi \ \Box(a' = a) \quad \supset \quad (\neg p) \ \mathit{istable} \ a \ .$$

□

For example, the following substitution instance (86) of valid implication (85) is needed in the analysis of Peterson's algorithm. The antecedent of (86) requires that the program variable flag_i is initialised to 0 in the starting global state, suitably framed and not altered by the process $\mathit{Peterson}_{1-i}$'s projected interval consisting of exactly the states satisfying $\mathit{pid} \neq i$. The consequent of (86) then ensures that the value 0 is correctly passed on to flag_i in the first state for process $\mathit{Peterson}_i$'s projected interval, which is the one containing all states satisfying $\mathit{pid} = i$:

$$\mathit{flag}_i = 0 \ \wedge \ \mathit{iframe}(.\mathit{flag}_i) \ \wedge \ (\mathit{pid} \neq i) \ \Pi \ \Box(\mathit{flag}'_i = \mathit{flag}_i) \quad \supset \quad (\mathit{pid} = i) \ \Pi^{\mathrm{u}} \ (\mathit{flag}_i = 0) \ . \tag{86}$$

The next valid implication is needed later on in the proof of Proposition A.2, which concerns *noninterference* with a program variable's value *at any time*. This implication shows how for a program variable a the operator *istable* can help to import $\mathit{iframe}(.a)$ from the global interval into a subinterval projected using the propositional variable p. It works because *istable* ensures that any processes operating in states with $\neg p$ do not alter the value of the variable a.

$$\mathit{iframe}(.a) \ \wedge \ (\neg p) \ \mathit{istable} \ a \quad \supset \quad p \ \Pi^{\mathrm{u}} \ \mathit{iframe}(.a) \ . \tag{87}$$

Proof of validity of implication (87). Suppose on the contrary that there is an interval σ with the implication's antecedent true and its consequent $p \ \Pi^{\mathrm{u}} \ \mathit{iframe}(.a)$ false. Our proof shows that σ itself falsifies

iframe(.a). Now if σ falsifies the consequent $p\ \Pi^{\mathrm{u}}$ *iframe*(.a), then there is an interval σ' obtained by projecting from σ using the propositional variable p and which does not satisfy *iframe*(.a). It follows from the definition of *iframe*(.a) that σ' satisfies the next formula:

$$\Diamond\big(\mathit{more}\ \wedge\ a' = a\ \wedge\ (\bigcirc a) \neq a\big)\ . \tag{88}$$

We will show that the original global interval σ also satisfies (88) (so falsifies *iframe*(.a) to obtain a contradiction). Now σ', which satisfies the $\Diamond$-formula (88), has a suffix subinterval σ'' satisfying the formula's operand $\mathit{more} \wedge a' = a \wedge (\bigcirc a) \neq a$, so with the following properties:

1. σ'' has at its start at least two states, denoted here as s_1 and s_2.
2. The first of these states s_1 satisfies the state formula $a' = a$.
3. The values of a in s_1 and s_2 differ.

Items 1 and 2 readily ensure that the global suffix interval of σ starting from s_1 satisfies the formulas *more* and $a' = a$. We now show that this global suffix interval of σ also satisfies the formula $(\bigcirc a) \neq a$. Observe that p is false in all of the intermediate *global* states of the interval σ after s_1 and before s_2 because these two states were projected using p. Consider the subformula $(\neg p)$ *istable* a in implication (87)'s antecedent which the global interval σ is assumed to satisfy. This subformula guarantees that a's value in each of these intermediate global states (which, if they exist, all satisfy $\neg p$) equals a's value in s_2 because $(\neg p)$ *istable* a ensures that whenever p is false, a's value cannot change between the current state and the next one. Hence, a's value in all of the intermediate global states and in s_2 are equal. Now a's values in s_1 and s_2 differ (recall item 3 above concerning the interval σ''), so a's value in s_1 likewise differs from a's value in all the intermediate global states. Therefore, the values of a in s_1 and its *adjacent global successor* state (i.e., either s_2 or the first intermediate global state after s_1) differ, so the *global* suffix subinterval of σ starting from s_1 does not satisfy the equality $(\bigcirc a) = a'$. As already noted above, this global suffix interval of σ readily satisfies the formulas *more* and $a' = a$. Therefore, this suffix of σ satisfies the conjunction $\mathit{more} \wedge a' = a \wedge (\bigcirc a) \neq a$. Hence, σ itself satisfies the formula $\Diamond(\mathit{more} \wedge a' = a \wedge (\bigcirc a) \neq a)$ and consequently falsifies *iframe*(.a), thus yielding a contradiction. □

An alternative more syntactic proof of the validity of implication (87) could show the validity of the following implication which splits the consequent's negation $p\ \Pi \neg$ *iframe*(.a) into two parts $p\ \Pi^{\mathrm{u}} \neg$ *iframe*(.a) and $\Diamond p$ (e.g., see the relevant valid equivalence in (10) about expressing Π using Π^{u} and $\Diamond$), and then ensures that the first of these guarantees the second's negation:

$$\mathit{iframe}(.a)\ \wedge\ (\neg p)\ \mathit{istable}\ a\ \wedge\ p\ \Pi^{\mathrm{u}}\ \neg\ \mathit{iframe}(.a)\quad \supset\quad \Box\neg p\ .$$

Induction over time could then be used to infer this implication's consequent $\Box\neg p$ from the antecedent by first showing the validity of the following two implications:

$$\mathit{iframe}(.a)\ \wedge\ (\neg p)\ \mathit{istable}\ a\ \wedge\ p\ \Pi^{\mathrm{u}}\ \neg\ \mathit{iframe}(.a)\quad \supset\quad \neg p$$
$$\mathit{iframe}(.a)\ \wedge\ (\neg p)\ \mathit{istable}\ a\ \wedge\ p\ \Pi^{\mathrm{u}}\ \neg\ \mathit{iframe}(.a)\ \wedge\ \mathit{more}\quad \supset$$
$$\bigcirc\big(\mathit{iframe}(.a)\ \wedge\ (\neg p)\ \mathit{istable}\ a\ \wedge\ p\ \Pi^{\mathrm{u}}\ \neg\ \mathit{iframe}(.a)\big)\ .$$

The next Proposition A.2 about a valid implication provides a convenient sufficient condition to ensure *noninterference* with a program variable a's value *at any time* by helping to import framing of a variable a from the global interval to an interval projected using the state formula p when the remaining non-projected states (i.e., those satisfying $\neg p$) do not attempt to alter a (i.e., the equality $a' = a$ is true in all of them):

Proposition A.2 (Noninterference with a program variable's value at any time). For any program variable a and propositional variable p, the implication below is valid:

$$\mathit{iframe}(.a)\ \wedge\ p\ \Pi^{\mathrm{u}}\ \Box(a' = a)\quad \supset\quad (\neg p)\ \Pi^{\mathrm{u}}\ \mathit{iframe}(.a)\ . \tag{89}$$

Proof. The validity of implication (89) readily follows by propositional reasoning combining valid implications (84) and (87). □

For example, the following instance of valid implication (89) helps import the global framing of each program variable flag_i into the corresponding process $\mathit{Peterson}_i$'s local interval because the other process never assigns to flag_i (i.e., $(\mathit{pid} \neq i)\ \Pi^{\mathrm{u}}\ \Box(\mathit{flag}_i' = \mathit{flag}_i)$, or equivalently $\Box(\mathit{pid} \neq i\ \supset\ \mathit{flag}_i' = \mathit{flag}_i)$):

$$\mathit{iframe}(.\mathit{flag}_i)\ \wedge\ (\mathit{pid} \neq i)\ \Pi^{\mathrm{u}}\ \Box(\mathit{flag}_i' = \mathit{flag}_i)\quad \supset\quad (\mathit{pid} = i)\ \Pi^{\mathrm{u}}\ \mathit{iframe}(.\mathit{flag}_i)\ .$$

The valid property below illustrates employing the operator *istable* as an alterative, somewhat more concise way to describe the global-time framing achieved by *iframe* in the previous implications (79) and (80). Here, *istable* helps express that when framing is done, each process $Peterson_i$ indeed does not at all alter the value of the variable $flag_{1-i}$ and preserves the value of *turn* whenever it equals $1 - i$:

$$iframe \ \wedge \ (pid = i) \,\Pi\, Peterson_i \quad \supset \quad (pid = i) \ istable \ flag_{1-i} \ \wedge \ (pid = i \ \wedge \ turn = 1 - i) \ istable \ turn \ . \tag{90}$$

A.2. Proof of mutual exclusion for Peterson's algorithm

The proof here of mutual exclusion ultimately shows in Theorem A.3 the validity of the earlier implication (40), and requires an analysis of the behaviour of the program variables when a process is in its critical section with $lab = l_4$. The next two valid implications deal with this and are easy to check by an examination of the definition of each process $Peterson_i$:

$$\models \quad Peterson_i \quad \supset \quad \Box\big(lab = l_4 \ \supset \ (flag_{1-i} = 0 \ \vee \ turn = i)\big) \tag{91}$$

$$\models \quad Peterson_i \quad \supset \quad \Box\big(lab = l_4 \ \supset \ flag'_0 = flag_0 \wedge flag'_1 = flag_1 \wedge turn' = turn\big) \ . \tag{92}$$

Proof of validity of implications (91) *and* (92). The first implication (91)'s validity follows from the fact that the state with label l_4 is always immediately preceded by the successful termination of the while-loop when its test $flag_{1-i} = 1 \wedge turn = 1 - i$ is false. The second implication (92)'s validity is a direct consequence of the *noop* at label l_4 not attempting to alter the value of any of the three program variables. □

The next valid implication requires the assumption that process $Peterson_i$ can regard the variable $flag_i$ as being framed not just globally but also in the local projected interval in which the process operates:

$$\models \quad iframe(.flag_i) \ \wedge \ Peterson_i \quad \supset \quad \Box(lab = l_4 \ \supset \ flag_i = 1) \ . \tag{93}$$

Proof of validity of implication (93). This can be checked by observing that whenever the assignment $flag_i := 1$ at label l_1 is executed, the equality $flag'_i = 1$ holds and the statements between labels l_1 and l_4 all ensure the equality $flag'_i = flag_i$ is true. Therefore, if framing of $flag_i$ is performed using $iframe(.flag_i)$, then the value of $flag_i$ remains unchanged until label l_4, where it still equals 1. □

The three valid implications below relating the local perspective of $Peterson_i$ when at label l_4 to global time readily follow from (91)–(93):

$$\models \quad (pid = i) \,\Pi\, Peterson_i \quad \supset \quad \Box\big(pid = i \wedge lab = l_4 \ \supset \ (flag_{1-i} = 0 \ \vee \ turn = i)\big) \tag{94}$$

$$\models \quad (pid = i) \,\Pi\, Peterson_i \quad \supset \quad \Box\big(pid = i \wedge lab = l_4 \ \supset \ flag'_0 = flag_0 \wedge flag'_1 = flag_1 \wedge turn' = turn\big) \tag{95}$$

$$\models \quad (pid = i) \,\Pi\, \big(iframe(.flag_i) \ \wedge \ Peterson_i\big) \quad \supset \quad \Box(pid = i \wedge lab = l_4 \ \supset \ flag_i = 1) \ . \tag{96}$$

For example, the validity of implication (94), which transfers information about the while-loop's test from process $Peterson_i$'s local time to global time, can be obtained from the validity of implication (91) by first projecting the antecedent and consequent of (91) using strong and weak versions of Π, respectively:

$$(pid = i) \,\Pi\, Peterson_i \quad \supset \quad (pid = i) \,\Pi^{\mathrm{u}}\, \Box\big(lab = l_4 \ \supset \ (flag_{1-i} = 0 \ \vee \ turn = i)\big)$$

The consequent can then be re-expressed by eliminating the operator Π^{u} to obtain the semantically equivalent LTL formula which serves as the consequent of implication (94). Hence, this implication is indeed valid.

Theorem A.3 (Mutual exclusion for Peterson's algorithm). Implication (40) concerning mutual exclusion for Peterson's algorithm is valid.

Proof. Let σ be an interval satisfying implication (40)'s antecedent, and let s_1 and s_2 be two adjacent states in σ with $lab = l_4$ in s_1. By symmetry, we only need to consider the case where $pid = 0$ in s_1 and $pid = 1$ in s_2 and show that $lab \neq l_4$ in s_2. Valid implications (94)–(96), which concern when $lab = l_4$ holds, respectively

ensure that state s_1 satisfies all of the following three formulas:

$$\mathit{flag}_1 = 0 \ \vee \ \mathit{turn} = 0 \tag{97}$$

$$\mathit{flag}'_0 = \mathit{flag}_0 \ \wedge \ \mathit{flag}'_1 = \mathit{flag}_1 \ \wedge \ \mathit{turn}' = \mathit{turn} \tag{98}$$

$$\mathit{flag}_0 = 1 \tag{99}$$

The global framing of the three program variables flag_0, flag_1 and turn together with the conjunction (98) guarantees that these variables do not change their values from state s_1 to state s_2. Hence, by disjunction (97), in state s_2 either $\mathit{flag}_1 = 0$ or $\mathit{turn} = 0$. Furthermore, by (99), state s_2 also has $\mathit{flag}_0 = 1$. Here is an analysis of each of the two possibilities in disjunction (97) which forces state s_1 to satisfy either $\mathit{flag}_1 = 0$ or $\mathit{turn} = 0$:

- State s_1 satisfies $\mathit{flag}_1 = 0$: It follows by framing using (98) being true in state s_1 that the equality $\mathit{flag}_1 = 0$ also holds in the successor state s_2, and hence so does $\mathit{flag}_1 \neq 1$. Now we have assumed $\mathit{pid} = 1$ in state s_2, so valid implication (96) yields from $\mathit{flag}_1 \neq 1$ that $\mathit{lab} \neq l_4$ holds in s_2, so mutual exclusion is indeed observed in this case.
- State s_1 satisfies $\mathit{turn} = 0$: Framing using (98) being true in state s_1 results in the successor state s_2 satisfying the equality $\mathit{turn} = 0$. Likewise, framing using (99) being true in state s_1 results in the successor state s_2 also satisfying $\mathit{flag}_0 = 1$. Therefore, s_2 fails to satisfy the disjunction $\mathit{flag}_0 = 0 \vee \mathit{turn} = 1$. Now pid=1 holds in s_2, so the valid implication (94) ensures in s_2 that lab cannot equal l_4. As a result, mutual exclusion is guaranteed for this case.

Hence, both of the two cases are inconsistent with state s_2 satisfying $\mathit{lab} = l_4$, so implication (40) is indeed valid, and mutual exclusion is ensured. □

A.3. Proof of liveness for Peterson's algorithm

We now present a proof of validity for implication (45) concerning liveness by adapting Peterson's original liveness proof [Pet81]. Here is a theorem regarding this:

Theorem A.4 (Liveness for Peterson's algorithm). Implication (45) concerning liveness for Peterson's algorithm is valid.

Much of the analysis involves formulas in the logic LTL+Π. Let σ be an interval satisfying implication (45)'s antecedent:

$$\mathit{init} \ \wedge \ \mathit{iframe} \ \wedge \ (0 :: \mathit{Peterson}_0) \ ||| \ (1 :: \mathit{Peterson}_1) \ .$$

Therefore, this global interval σ includes the states of both processes. By symmetry, we only need to ensure liveness for process $\mathit{Peterson}_0$ and show that σ satisfies the first conjunct in implication (45)'s consequent:

$$(\mathit{pid} = 0) \ \Pi \ \Box(\mathit{lab} = l_0 \ \supset \ \Diamond \mathit{lab} = l_4) \ .$$

Assume on the contrary that σ fails to satisfy this. The main loop of each process $\mathit{Peterson}_i$ can behave in one of three modes. Here is an informal summary of these, which are considered shortly in more detail:

1. *Mode a – finite iterations, each having finite length*: Process $\mathit{Peterson}_i$ eventually terminates at label l_7.
2. *Mode b – finite iterations but with the last one having infinite length*: The process eventually gets stuck infinitely long at label l_3 waiting in vain to enter its critical section.
3. *Mode c – infinite iterations and infinite length (chop-omega)*: The process infinitely often enters and exits its critical section, each time executing along the way the assignment statement $\mathit{turn} := 1 - i$ at label l_2. As in the previous Mode b, $\mathit{Peterson}_i$ does not terminate, but in Mode c the process visits each of its labels l_0, l_1, ..., l_6 infinitely often (although perhaps sometimes skipping label l_3).

The definition of each process $\mathit{Peterson}_i$ in Fig. 2 ensures that whenever the process is at label l_0, it eventually reaches label l_4 or gets stuck infinitely long at label l_3 executing the statement *noop*. Therefore, if process $\mathit{Peterson}_0$ ever gets stuck waiting to enter its critical section, the global interval σ satisfies the next formula:

$$(\mathit{pid} = 0) \ \Pi \ \big(\mathit{inf} \wedge \Diamond\Box(\mathit{lab} = l_3)\big) \ .$$

The rest of the proof shows that this behaviour is inconsistent with each of the three ways the second process $\mathit{Peterson}_1$ can execute its own main loop. Here is a summary of why for each mode:

1. *For Mode a*: If the second process $Peterson_1$ terminates in finite time, then the value of the variable $flag_1$ equals 0 from then on because $Peterson_1$ ensures $flag_1 = 0$ before terminating and the other process $Peterson_0$ never alters $flag_1$. Therefore, if $Peterson_0$ tries later on to enter its critical section, it succeeds because its own while-loop's test $flag_1 = 1 \wedge turn = 1$ is false.
2. *For Mode b*: If *both* processes eventually get stuck waiting infinitely long at label l_3 to enter their respective critical sections, then the while-loop's test $flag_0 = 1 \wedge turn = 0$ for the second process $Peterson_1$ is true infinitely long. Furthermore, $flag_1 = 1$ holds at this label. The two processes when simultaneously at their respective *noop* statements at label l_3 combine to leave the value of the variable *turn* unchanged from then on since the equality $turn' = turn$ persists in being true in all of the processes' projected local states, so in all global ones as well. Therefore, global framing of *turn* ensures that its subsequent value is stable, so either always 0 or always 1 from then on. Hence, one of the processes must succeed in entering its critical section. This contradicts the assumption of both processes getting stuck in this mode, so such a situation cannot in fact occur.
3. *For Mode c*: Suppose the second process $Peterson_1$ successfully iterates through its main loop infinitely often in and out of its critical section. Then each time an iteration is performed, the assignment statement $turn := 0$ at label l_2 is executed prior to entering the critical section. It follows that when this process, after leaving its critical section, tries the next time to enter it, the while-loop's test $flag_0 = 1 \wedge turn = 0$ is true because the first process $Peterson_0$ is stuck with $flag_0 = 1$ and cannot alter *turn*'s value to instead be 1. Hence, the second process likewise gets stuck in vain waiting. Therefore, this situation, like the previous one, cannot happen.

The next valid implication describes how every execution of a process $Peterson_i$ conforms to the behaviour of one of the three modes already discussed:

$$Peterson_i \quad \supset \quad (finite \ \wedge \ fin\, lab = l_7) \ \vee \ (inf \ \wedge \ \Diamond\Box lab = l_3) \ \vee \ (inf \ \wedge \ \Box\Diamond lab = l_2) \ . \tag{100}$$

Proof of validity of implication (100). This can be easily shown by analysing the three ways the main loop of either process $Peterson_i$ is executed. We start with the valid implication below having a consequent which summarises the process flow-of-control just concerning *lab*'s relevant behaviour when it equals one of the labels l_2, l_3 or l_7:

$$Peterson_i \quad \supset \quad \big((finite \wedge \Diamond lab = l_2) \ \vee \ (inf \wedge \Diamond\Box lab = l_3)\big)^*; (empty \wedge lab = l_7) \ . \tag{101}$$

This can be demonstrated to be valid by simply analysing the sequential parts of the definition of an individual process $Peterson_i$.

It follows that an individual iteration of the skeletal loop for a process $Peterson_i$ in implication (101)'s consequent satisfies the next disjunction:

$$(finite \wedge \Diamond lab = l_2) \ \vee \ (inf \wedge \Diamond\Box lab = l_3) \ . \tag{102}$$

It is not hard to check that if this is sequentially repeated some finite or infinite number of times in an interval, then the interval must either (a) have finite length, (b) be infinitely long with $lab = l_3$ eventually true in all states of some suffix subinterval, or (c) be infinitely long and have $lab = l_2$ true infinitely often. Hence, the interval must satisfy the next disjunction:

$$finite \ \vee \ (inf \ \wedge \ \Diamond\Box lab = l_3) \ \vee \ (inf \ \wedge \ \Box\Diamond lab = l_2) \ . \tag{103}$$

Here is a valid implication expressing that an interval exhibiting the looping with iterations described by the disjunction (102) satisfies the second disjunction (103):

$$\begin{aligned} &\big((finite \wedge \Diamond lab = l_2) \ \vee \ (inf \wedge \Diamond\Box lab = l_3)\big)^* \quad \supset \\ &\quad finite \ \vee \ (inf \ \wedge \ \Diamond\Box lab = l_3) \ \vee \ (inf \ \wedge \ \Box\Diamond lab = l_2) \ . \end{aligned} \tag{104}$$

The following valid implication is a modification of the previous one (104) to take into account that in finite intervals the last state of process $Peterson_i$ has $lab = l_7$:

$$\begin{aligned} &\big((finite \wedge \Diamond lab = l_2) \ \vee \ (inf \wedge \Diamond\Box lab = l_3)\big)^*; (empty \wedge lab = l_7) \quad \supset \\ &\quad (finite \wedge fin\, lab = l_7) \ \vee \ (inf \ \wedge \ \Diamond\Box lab = l_3) \ \vee \ (inf \ \wedge \ \Box\Diamond lab = l_2) \ . \end{aligned} \tag{105}$$

Only the leftmost disjunct is changed to now include that $lab = l_7$ is true in the final state. The other two disjunctions concern infinite intervals, so never involve a state where $lab = l_7$ is true.

The chain of the two valid implications (101) and (105) then yields the validity of implication (100). □

The consequent of implication (100) is stutter-invariant because its first disjunct can be equivalently expressed as $\mathit{finite} \wedge \Diamond\Box(\mathit{lab} = l_7)$, and the remaining two disjuncts can together be propositional rearranged into the equivalent conjunction $\mathit{inf} \wedge (\Diamond\Box \mathit{lab} = l_3 \vee \Box\Diamond \mathit{lab} = l_2)$.

The next property Inv_i is defined to be a state formula expressing an invariant which process $\mathit{Peterson}_i$ is shown to maintain in all local states:

$$\begin{array}{lll} \mathit{Inv}_i \;\widehat{=}\; & \mathit{flag}'_{1-i} = \mathit{flag}_{1-i} & \\ & \wedge\ (\mathit{lab} = l_2 \;\supset\; & \mathit{turn}' = 1 - i) \\ & \wedge\ (\mathit{lab} = l_3 \;\supset\; & \mathit{flag}_{1-i} = 1 \ \wedge\ \mathit{turn} = 1 - i \ \wedge\ \mathit{turn}' = \mathit{turn}) \\ & \wedge\ (\mathit{lab} = l_7 \;\supset\; & \mathit{flag}_i = 0 \ \wedge\ \mathit{flag}'_i = 0) \ . \end{array}$$

The first conjunct states that the program variable flag_{1-i} is never altered in a state in which process $\mathit{Peterson}_i$ is active. The remaining conjuncts concern relevant behaviour of specific program variables just when $\mathit{lab} \in \{l_2, l_3, l_7\}$.

In order to prove that a process $\mathit{Peterson}_i$ implies its invariant Inv_i in each local state, it is first necessary to check that the process never attempts to alter the program variable flag_{1-i}. The next easily checked valid implication expresses this (and was already presented and justified above as (75)):

$$\mathit{Peterson}_i \;\supset\; \Box(\mathit{flag}'_{1-i} = \mathit{flag}_{1-i}) \ .$$

Here is a valid implication concerning how a process $\mathit{Peterson}_i$ with suitable initialisation and framing of the program variable flag_i ensures its invariant holds in all local states:

$$\mathit{flag}_i = 0 \ \wedge\ \mathit{iframe}(.\mathit{flag}_i) \ \wedge\ \mathit{Peterson}_i \;\supset\; \Box \mathit{Inv}_i \tag{106}$$

We need to show for each process $\mathit{Peterson}_i$ that the initialised value of the program variable flag_i in the first *global* state is correctly passed on to the first *local* state. Here is a valid implication concerning this:

$$\begin{array}{l} \mathit{flag}_i = 0 \ \wedge\ \mathit{iframe}(.\mathit{flag}_i) \ \wedge\ (\mathit{pid} \neq i)\ \Pi^{\mathrm{u}}\ \Box(\mathit{flag}'_i = \mathit{flag}_i) \;\supset \\ \quad (\mathit{pid} = i)\ \Pi^{\mathrm{u}}\ (\mathit{flag}_i = 0 \wedge \mathit{iframe}(.\mathit{flag}_i)) \ . \end{array} \tag{107}$$

This can be obtained adapting instances of the two earlier valid implications (85) and (89) for any framed program variable a initialised to some constant value j and not altered in any global states with $\neg p$. We reproduce these implications below for the convenience of readers:

$$\begin{array}{l} a = j \ \wedge\ \mathit{iframe}(.a) \ \wedge\ p\ \Pi^{\mathrm{u}}\ \Box(a' = a) \;\supset\; (\neg p)\ \Pi^{\mathrm{u}}\ (a = j) \\ \mathit{iframe}(.a) \ \wedge\ p\ \Pi^{\mathrm{u}}\ \Box(a' = a) \;\supset\; (\neg p)\ \Pi^{\mathrm{u}}\ \mathit{iframe}(.a) \ . \end{array}$$

Concluding proof of Theorem A.4 concerning liveness. Valid properties discussed above are now combined to obtain the following proof of Theorem A.4:

Proof. We will obtain a contradiction by examining how the second process $\mathit{Peterson}_1$, when operating in any of its three possible modes, influences the first process $\mathit{Peterson}_0$, which itself is assumed here to operate solely in Mode b. The analysis below only considers *Mode a*, which occurs when the main loop of $\mathit{Peterson}_1$ terminates with $\mathit{lab} = l_7$, but the analysis for the other two modes is similar. As before, let σ be the global interval in which both processes concurrently execute within their respective local projected intervals. Assume that σ satisfies implication (45)'s antecedent, which is as follows:

$$\mathit{init} \ \wedge\ \mathit{iframe} \ \wedge\ (0 :: \mathit{Peterson}_0) \ ||| \ (1 :: \mathit{Peterson}_1) \ . \tag{108}$$

If $\mathit{Peterson}_1$ operates in *Mode a*, then the global interval σ satisfies all of the formulas below of particular relevance to the analysis of $\mathit{Peterson}_1$:

$$\begin{array}{l} \mathit{iframe} \\ (\mathit{pid} = 0)\ \Pi\ \big(\mathit{inf} \ \wedge\ \Box(\mathit{flag}'_1 = \mathit{flag}_1)\big) \\ (\mathit{pid} = 1)\ \Pi\ \big(\mathit{finite} \ \wedge\ \Box \mathit{Inv}_1 \ \wedge\ \mathit{fin}\ \mathit{lab} = l_7\big) \ . \end{array}$$

Here are justifications for the global interval σ satisfying these three formulas:

1. The first one *iframe* holds because implication (45)'s antecedent (108) explicitly includes global framing of the program variables.
2. The second holds because, first of all, implication (45)'s antecedent (108) includes the formula $(0 :: \mathit{Peterson}_0) \,|||\, (1 :: \mathit{Peterson}_1)$, which is itself semantically equivalent to the conjunction

   $$(\mathit{pid} = 0) \;\Pi\; \mathit{Peterson}_0 \;\wedge\; (\mathit{pid} = 1) \;\Pi\; \mathit{Peterson}_1 \;\; .$$

   Hence, the global interval σ satisfies the formula $(\mathit{pid} = 0) \;\Pi\; \mathit{Peterson}_0$. Also process $\mathit{Peterson}_0$ is assumed to be operating in Mode b, so runs infinitely long in its projected interval. Furthermore, the process never assigns to the program variable flag_1 in the course of this (see valid implication (77) above).
3. The third formula holds because interval σ satisfies the formula $(\mathit{pid} = 1) \;\Pi\; \mathit{Peterson}_1$ (by reasoning analogous to that above for the second formula about process $\mathit{Peterson}_0$), and when process $\mathit{Peterson}_1$ operates in Mode a, then its local projected interval is finite and it terminates at label l_7. We already ensured earlier that the *global* initialisation and framing of flag_1 can be imported into $\mathit{Peterson}_1$'s *local* projected time (see valid implication (107) above). Furthermore, when the program variable flag_1 equals 0 in the first local state and is also locally framed, the invariant Inv_1 is true in every one of process $\mathit{Peterson}_1$'s local states (see valid implication (106) above).

Now the invariant Inv_1 and $\mathit{lab} = l_7$ are both true in the last state of $\mathit{Peterson}_1$'s projected interval. Therefore, so is the conjunction $\mathit{flag}_1 = 0 \wedge \mathit{flag}'_1 = \mathit{flag}_1$. The global interval consequently satisfies the next projection formula:

$$(\mathit{pid} = 1) \;\Pi\; \big(\mathit{finite} \wedge \mathit{fin}(\mathit{flag}_1 = 0 \wedge \mathit{flag}'_1 = \mathit{flag}_1)\big) \;\; .$$

Therefore, this and the projection formula $(\mathit{pid} = 0) \;\Pi\; \big(\mathit{inf} \;\wedge\; \Box(\mathit{flag}'_1 = \mathit{flag}_1)\big)$ guarantee that both $\mathit{Peterson}_0$'s projected interval and the global interval are in contrast infinite. Furthermore, it follows that $\mathit{flag}'_1 = \mathit{flag}_1$ in all the infinite number of global states after the last state for $\mathit{Peterson}_1$. Here is a valid implication about this:

$$\begin{array}{l}(\mathit{pid} = 0) \;\Pi\; \big(\mathit{inf} \wedge \Box(\mathit{flag}'_1 = \mathit{flag}_1)\big) \;\wedge\; (\mathit{pid} = 1) \;\Pi\; \big(\mathit{finite} \wedge \mathit{fin}(\mathit{flag}_1 = 0 \wedge \mathit{flag}'_1 = \mathit{flag}_1)\big) \quad \supset \\ \qquad \mathit{inf} \;\wedge\; \Diamond\big(\mathit{flag}_1 = 0 \wedge \Box(\mathit{flag}'_1 = \mathit{flag}_1)\big) \;\; .\end{array} \tag{109}$$

This is a substitution instance of the following valid implication:

$$p \;\Pi\; (\mathit{inf} \wedge \Box r) \;\wedge\; (\neg p) \;\Pi\; \big(\mathit{finite} \wedge \mathit{fin}(q \wedge r)\big) \quad \supset \quad \mathit{inf} \;\wedge\; \Diamond(q \wedge \Box r) \;\; .$$

The combination of implication (109)'s consequent together assumption that *iframe* holds in the global interval then yields that the global interval has an infinite suffix satisfying $\Box(\mathit{flag}_1 = 0)$. Therefore, process $\mathit{Peterson}_0$ cannot get stuck because the test of its while-loop will eventually fail, and so the process will enter its critical section. The valid implication below encapsulates the reasoning just discussed regarding how when the second process $\mathit{Peterson}_1$ operates in *Mode a*, the first process $\mathit{Peterson}_0$ cannot get stuck infinitely long waiting because it will eventually detect $\mathit{flag}_1 = 0$.

$$\begin{array}{l}\mathit{iframe} \;\wedge\; (\mathit{pid} = 1) \;\Pi\; \big(\mathit{finite} \wedge \mathit{fin}(\mathit{flag}_1 = 0 \wedge \mathit{flag}'_1 = \mathit{flag}_1)\big) \quad \supset \\ \qquad (\mathit{pid} = 0) \;\Pi^{\mathrm{u}}\; \big((\mathit{inf} \;\wedge\; \Box(\mathit{flag}'_1 = \mathit{flag}_1)) \;\supset\; \Diamond\Box(\mathit{flag}_1 = 0)\big) \;\; .\end{array}$$

□

Consider now the alternative version of Peterson's algorithm in Fig. 3. This differs only by the inclusion of auxiliary program variables cs_0 and cs_1. Each such variable cs_i is set to true when process $\mathit{Peterson}'_i$ enters its critical section. There is in fact a delay of one local step required for this to happen. We present the two versions of the algorithm because they vary regarding how the safety and liveness properties are specified and analysed. The alternative version uses a more conventional and straightforward safety property of the form $\Box w$ that is clearly stutter-invariant but requires the auxiliary variables cs_0 and cs_1. In principle, this version can be verified even without program labels (as is done with our compositional analysis in [Mos14]). The reasoning concerning the first version works directly with a different variant of the safety property which seems closer to the essence of interleaving with projection. For example, the first version's key safety property (40), which is expressed using label constants and was already discussed above, relates the two processes at adjacent projected states. It includes $\bigcirc$-formulas, but we showed above that it is nevertheless stutter-invariant.

Process $Peterson'_i$, for $i \in \{0,1\}$
for some times do (
l_0: *noop*;
l_1: $\mathit{flag}_i := 1$;
l_2: $\mathit{turn} := 1 - i$;
while $(\mathit{flag}_{1-i} = 1 \ \wedge \ \mathit{turn} = 1 - i)$ *do*
l_3: *noop*;
l_4: $cs_i := \mathit{true}$; /* Critical section */
l_5: $\mathit{flag}_i, cs_i := 0, \mathit{false}$; /* Leave critical section */
l_6: *noop*
);
l_7: *enoop*

Let $\mathit{dom}(\mathit{nval}_{Peterson'}) = \{.\mathit{flag}_0, .\mathit{flag}_1, .\mathit{turn}, .cs_0, .cs_1\}$.
Initially $\mathit{flag}_0 = \mathit{flag}_1 = 0$, $\neg cs_0$ and $\neg cs_1$, but *turn* can start as either 0 or 1.

Fig. 3. Alternative version of Peterson's algorithm with processes $Peterson'_0$ and $Peterson'_1$

Here are some temporal properties for this second version of the algorithm:

$$\models \quad \mathit{init} \ \wedge \ \mathit{iframe} \ \wedge \ (0 :: Peterson'_0) \ ||| \ (1 :: Peterson'_1) \quad \supset \quad \Box\neg(cs_0 \ \wedge \ cs_1) \tag{110}$$

$$\models \quad \mathit{init} \ \wedge \ \mathit{iframe} \ \wedge \ (0 :: Peterson'_0) \ ||| \ (1 :: Peterson'_1) \tag{111}$$
$$\supset \quad \Box\big((\mathit{flag}_0 = 1 \wedge \neg cs_0) \ \supset \ \Diamond cs_0\big) \ \wedge \ \Box\big((\mathit{flag}_1 = 1 \wedge \neg cs_1) \ \supset \ \Diamond cs_1\big)$$

$$\models \quad \mathit{init} \ \wedge \ \mathit{iframe} \ \wedge \ (\mathit{inf} \wedge 0 :: Peterson'_0) \ ||| \ (\mathit{inf} \wedge 1 :: Peterson'_1) \tag{112}$$
$$\supset \quad \Box\Diamond(\mathit{flag}_0 = 0 \wedge \neg cs_0) \ \wedge \ \Box\Diamond(\mathit{flag}_0 = 1 \wedge cs_0)$$
$$\wedge \ \Box\Diamond(\mathit{flag}_1 = 0 \wedge \neg cs_1) \ \wedge \ \Box\Diamond(\mathit{flag}_1 = 1 \wedge cs_1) \ .$$

Implication (110) expresses the safety property for mutual exclusion. We already mentioned its consequent $\Box\neg(cs_0 \ \wedge \ cs_1)$ does not have any $\bigcirc$-operators so is clearly stutter-invariant. This is in contrast with the consequents of the initial two safety properties (39)–(40) for mutual exclusion in the first version of Peterson's algorithm, where (39) is not stutter-invariant and (40) is but requires a proof showing equivalence to a formula in LTL$-\bigcirc$.

This author-produced version was formatted on 25 April 2017.