Synthesising Classic and Interval Temporal Logic

Sven Schewe
Department of Computer Science
University of Liverpool
Liverpool, United Kingdom
sven.schewe@liverpool.ac.uk

Cong Tian
ICTT and ISN Lab
Xidian University
Xi'an, 710071, P.R.China
c.tian.xdu@gmail.com

***Abstract*—*Linear-Time Temporal Logic* (LTL) is one of the most influential logics for the specification and verification of reactive systems. An important selling point of LTL is its striking simplicity, which might be a reason why none of the many extensions suggested to LTL have gained the same influence. Interval based temporal logics like *Interval Temporal Logic* (ITL) are a more recent branch of temporal logics with their own niche of interesting applications. On first glance, interval based temporal logics very little resemble LTL and the spread of these logics beyond their niche is hampered by a seeming structural incompatibility with LTL. When competing for being applied on a larger scale, interval based temporal logics would fight a losing battle against a more established competitor with better complexity and mature tools. In this paper, we suggest to extend ITL to *Pop Logic* (PL) by introducing a simple *pop* operator that revokes the binding of the *chop* operation—very much like the popping operation in a stack—and show that LTL can be viewed as a syntactic subset of PL. This is a surprising twist: by strengthening the comparably exotic logic ITL slightly and by using the new *pop* and the old *chop* operator as primitive constructs, we obtain a logic for which LTL is a de-facto syntactic fragment. The power of this extension is that it can, by subsuming both interval and classic temporal logics, synthesise both concepts to a common framework. The charm of this extension is that PL does not sacrifice the simplicity that makes its sub-logics attractive.**

I. Introduction

Temporal logics are popular formalisations that can express properties about the temporal order of events. The family of temporal logics has grown over the years, containing linear [15] and branching time logics [4], [2], and, more recently, game, alternating time, and coordination logics [1], [9]. While linear time temporal logics are concerned with properties of paths, branching time logics describe properties that depend on the branching of computational tree structures.

There has been a long debate between linear time and branching time temporal logics, but three decades worth experiences have shown that branching time logics are hard to be understood and error-prone [16], [20]. In contrast, linear time temporal logics seem to lend themselves to the system designers. This makes *Linear-time Temporal Logic* (LTL) [15], the original linear time temporal logic, one of the most influential logics in specification and verification. From a theoretical point of view, one can show that the expressive power of LTL is restricted to star-free expressions. To overcome this limitation, several extensions and variations have been proposed. Notably, Quantified Linear Time Temporal Logic (QLTL) [17] and Extended Temporal Logic (ETL) [22], [21] are extensions of LTL for the expressiveness of full ω-regular language. Temporal Logic of Actions (TLA) is a variation of LTL where state changes can be easily handled through actions [12]. However, these extensions (or variations) affect the simplicity of LTL and make it less intuitive. This might be one of the reasons why none of them has gained the same influence as LTL itself.

Interval based temporal logics are a more recent branch of temporal logics with their own niche of interesting applications. The characteristic operator of these logics is the *chop* operator, often denoted by the symbol ';'. Different from the traditional temporal operators $\Box$ (always) and $\mathcal{U}$ (until), a *chop* construct, $p\,;q$, holds over a path (or an interval) if, and only if, the path can be split into two parts, such that p holds over the first part and q holds over the second part. The *chop* operator was first used as a temporal construct by Harel, Kozen and Parikh [11] and studied in more depth by Chandra, Halpern, Meyer and Parikh [3]. Halpern, Manna, and Moszkowski showed that *chop* is a useful operator when reasoning about time-dependent digital hardware [10], which triggered the development *Interval Temporal Logic* (ITL), a temporal logic based on *chop*, *chop star*, *next*, and *projection* operations, by Moszkowski [13], [14]. Initially, ITL is confined within finite models. Projection Temporal Logic (PTL) [5], [6], [7] is an extension of ITL with infinite models and a new projection construct, $(P_1,\ldots,P_m)\ prj\ Q$.

Compared to classic temporal logics, interval based temporal logics greatly simplify the formulation of certain correctness properties [8], which underlines the usefulness of these logics for specification and formal reasoning about concurrent systems. Interval based temporal logics lend themselves particularly well to reasoning about properties with a 'scope'; such properties are very common in most programming languages. Further, with *chop* operations, sequential behaviours can be described elegantly and succinctly; and full regular expressiveness can easily be achieved by the introduction of a *chop star* operator, which can be intuitively understood as the multi-time implementation of a *chop* operation.

Interval based temporal logics very little resemble LTL and the spread of these logics beyond their niche is hampered by a seemingly structural incompatibility with LTL. When competing for being applied on a larger scale, interval based temporal logics would fight a losing battle against a more established competitor with better complexity and well developed tools. This leads to the question of whether or not the differences between these logics can be bridged *without* affecting their simplicity. A first approach would be to consider a simple merge of the operators. Why not enrich LTL by a *chop* operator? Or, likewise, ITL with an until? The disadvantage of such a solution is that the strongest advantages of these logics is their simplicity: they are build around a single intuitive concept. In this sense, each extension comes to the cost of elegance and effects the intuitive access to these logics.

In this paper, we discuss an alternative approach. We introduce a natural extension of ITL, *Pop Logic* (PL), by introducing a simple operator *pop* operation, denoted as $\uparrow$, that revokes the binding of a *chop* operation. (The name is inspired by the popping operation in a stack.) The pop operation is interesting in itself: it can be used as a pseudo inverse of a chop operation, as $(\uparrow \varphi)\,; \mathit{true}$ is logically equivalent to φ, and it provides a fresh view on the interval temporal logics. In PL, we view the scoping implied by the chop operator as the scoping invoked by a call. The *pop* operation provides access to lower levels of the call tree, which implies a recognition of the call structure in the semantics of the logic. This entails a semantics that accurately reflects this call structure. But while it clearly provides some insight into the relation of calls and interval logic, the main advancement is the link it establishes with LTL: We show that LTL operators can be viewed as snippets of PL operators. LTL can therefore justly be viewed as a de-facto syntactic subset of PL. This, in turn, provides another insight to interval temporal logics: Using the extension, we get—with LTL—a meaningful—and even popular—a PSPACE-complete sub-logic of PL. We use this observation to define more general structural restrictions of PL that preserve this low complexity.

The remainder of the paper is organised as follows. The following section presents the syntax and semantics of Pop Logic and we discuss its relation to ITL in Section III. We then discuss the embedding of LTL into PL in Section IV and demonstrate the decidability of PL by embedding it in QLTL in Section V. (This proof simplifies, as a small side result, the known decidability proof of ITL.) Finally, we study PSPACE-complete subsets of PL in Section VI.

II. Pop Logic

In this section, we introduce Pop Logic (PL), a syntactical extension of ITL with a *pop operator* '$\uparrow$' that revokes the most recent effect (interval restriction) imposed by a *chop* operator. Intuitively, a *chop* operator pushes a new interval into an interval stack, while a pop-operator pops the top element of this stack (hence the name). Just like ITL, its extension PL has a natural semantics for finite and infinite words.

A. Syntax of Pop Logic

Pop Logic is interpreted over finite and infinite words over a countable set Π of atomic propositions, which includes a special proposition $\top$ (true) that holds in every position of the word. The syntax of Pop Logic is given by the grammar

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \varphi;\varphi \mid \uparrow\varphi,$$

where $p \in \Pi$ is an atomic proposition. PL extends the syntax of ITL (see the following section) by the unary pop operator $\uparrow$.

B. Semantics of Pop Logic

We call a PL formula φ *well formed* if there exists no node in the formula tree of φ such that strictly more *pop* operators than *chop* operators occur on the way from the root of the formula tree to this node. We define the PL semantics only for well formed formulas. The intuition for the exclusion of ill formed formulas is that we would otherwise try to pop the bottom element of an interval stack.

For $l,u \in \omega$ and $u' \in \omega \uplus \{\infty\}$, $[l,u] = \{k \in \omega \mid l \le k \le u\}$ and $[l,u'[= \{k \in \omega \mid l \le k < u'\}$ are (integer) intervals.

The semantics of a well formed PL formula φ leans on the semantics of ITL, but as we have the power to revoke the effect of previous *chop* operators, we have to keep track of a stack of intervals, comparable to the call structure in recursive procedures. Consequently, a sub-formula ψ of φ needs to be interpreted in the context of this stack of intervals at the time where ψ is evaluated.

PL is interpreted over a (finite or infinite) word $\sigma \in (2^\Pi)^* \uplus (2^\Pi)^\omega$. When convenient, we interpret σ as a function from $[0,|\sigma|[$ to 2^Π, $\sigma : [0,|\sigma|[\to 2^\Pi$. In the following, $I_0,\dots,I_n$ (with $n \in \omega$) is a non-empty interval stack with $[0,|\sigma|[= I_0 \supseteq I_1 \supseteq \dots \supseteq I_n$. We first define the interpretation of a word σ in an interval stack $I_0,\dots,I_n$ at a position k.

$$\begin{array}{lll}
\sigma; I_0,\dots,I_n; k \models p & \text{iff} & k \in I_n \text{ and } p \in \sigma(k),\\
\sigma; I_0,\dots,I_n; k \models \neg\varphi & \text{iff} & \sigma; I_0,\dots,I_n; k \not\models \varphi,\\
\sigma; I_0,\dots,I_n; k \models \varphi \vee \psi & \text{iff} & \sigma; I_0,\dots,I_n; k \models \varphi\\
 & \text{or} & \sigma; I_0,\dots,I_n; k \models \psi,\\
\sigma; I_0,\dots,I_n; k \models \bigcirc\varphi & \text{iff} & \sigma; I_0,\dots,I_n; k+1 \models \varphi,\\
\sigma; I_0,\dots,I_n,[b,e[; k \models \varphi;\psi & \text{iff} & k \in [b,e[\text{ and}\\
\multicolumn{3}{r}{\exists l \in [k,e[.\ \sigma; I_0,\dots,I_n,[b,e[,[b,l]; k \models \varphi \text{ and}}\\
\multicolumn{3}{r}{\sigma; I_0,\dots,I_n,[b,e[,[l,e[; l \models \psi, \text{ and}}\\
\sigma; I_0,\dots,I_n,I_{n+1}; k \models \uparrow\varphi & \text{iff} & \sigma; I_0,\dots,I_n; k \models \varphi.
\end{array}$$

A finite of infinite word σ is a *model of* φ, denoted $\sigma \models \varphi$, if, and only if, φ holds initially on the complete word, that is, if, and only if, $\sigma; [0,|\sigma|[; 0 \models \varphi$ holds.

Note that, as usual, for all finite words σ and all $k \geq |\sigma|$, $\sigma;k \not\models \top$.

No pop on the singleton stack.: If the interval stack contains only one element, then the semantics of the pop operator is not properly defined. Note that we could easily avoid this by defining '$\sigma;I_0;k \models \uparrow \varphi$ if, and only if, $\sigma;I_0;k \models \varphi$'. Choosing to do so would also allow us to treat ill formed formulas by a convention that could intuitively be phrased as 'an attempt to *pop* the bottom element of an interval stack is ignored'. However, we consider it more natural to simply disallow such operations.

III. Relation of Pop Logic to ITL

In order to obtain a simple logic, we have extended a basic version of ITL, ITL without star [5], [7], whose syntax simply represents the PL syntax without *pop*. The *semantics* of ITL, however, is traditionally 'flat' in the sense that the semantics of ITL is normally not defined using an interval stack: Without the introduced pop operation, there is no requirement for it because preserving the top-most interval is unnecessary.

The traditional ITL semantics for *chop* is

$$\begin{aligned}\sigma;[b,e[;k \models \varphi;\psi \text{ iff } & k \in [b,e[\text{ and} \\ & \exists l \in [k,e[.\ \sigma;[k,l];k \models \varphi \text{ and } \sigma;[l,e[;l \models \psi.\end{aligned}$$

This has no effect on the evaluation of $\sigma;[0,|\sigma|[;0 \models \varphi$, because without a *pop* operation we would never refer to any other than the top interval on the interval stack.

IV. Embedding LTL in Pop Logic

In the following, we briefly present the related temporal logic LTL and discuss its embedding in PL. The syntax of LTL is given by the following grammar:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \varphi\,\mathcal{U}\varphi$$

The semantics of LTL formulas is only defined on infinite words. Like with ITL, the concept of interval stacks does not exist in LTL and the interpretation of a formula φ at a position k is defined inductively by:

$$\begin{array}{lll}\sigma;k \models p & \text{iff} & p \in \sigma(k), \\ \sigma;k \models \neg\varphi & \text{iff} & \sigma;k \not\models \varphi, \\ \sigma;k \models \varphi \vee \psi & \text{iff} & \sigma;k \models \varphi \text{ or } \sigma;k \models \psi, \\ \sigma;k \models \bigcirc\varphi & \text{iff} & \sigma;k+1 \models \varphi, \\ \sigma;k \models \varphi\,\mathcal{U}\psi & \text{iff} & \exists i \geq k.\sigma;i \models \psi \text{ and } \forall j \in [k,i[.\ \sigma;j \models \varphi.\end{array}$$

A central advantage of Pop Logic is that it provides us with a very simple embedding of LTL, while preserving the scoping allowed for by ITL. The LTL operators can be viewed as abbreviations of small snippets of PL, just as $\Box$ and $\Diamond$ operators in LTL can be viewed as snippets of LTL syntax.

In PL, we have the following abbreviations:

$$\begin{array}{lrcl}\text{PUSH} & \psi : \vartheta & = & \psi \,;\, \uparrow\vartheta \\ \text{EVENTUALLY} & \Diamond\psi & = & \top : \psi \\ \text{ALWAYS} & \Box\psi & = & \neg\Diamond\neg\psi \\ \text{UNTIL} & \psi\,\mathcal{U}\vartheta & = & \big(\Box\uparrow(\psi\vee\vartheta)\big) : \vartheta\end{array}$$

The first abbreviation introduces a *push* operator ':', which is closely related to the *chop*. Different to *chop*, a *push* operation only increases the call stack on the left. There is little difference in the effect of *push* and *chop*, as the evaluation of the truth of the formula on the right is started at the beginning of the second interval. In our view, the *push* operation is the more useful, as it reflects a call and return situation; our intuition is that we first satisfy the formula on the right (pushing down one level) and then come back (pop) and satisfy the right side.

On infinite intervals, the eventually and always operators '$\Diamond$' and '$\Box$' have the same semantics as in LTL, with the natural extension for finite intervals (holds somewhere within the interval and holds throughout the interval, respectively). A similar claim holds for until.

Using only the LTL operators, it is plain to see that we maintain the semantics of LTL. Note that, in the LTL fragment, a *chop* (or *pop*) operator can only occur in the abbreviations for a temporal operator, and there it is 'guarded' in the sense that the stack cannot grow to a size of more than three.

V. Decidability of Pop Logic

There are two obvious approaches to show the decidability of PL: A direct translation to automata and an embedding into QLTL. We describe an embedding into QLTL, because it is a much simpler transformation. It also simplifies the decidability proofs for the sub-logic ITL with finite [13] and infinite [5] word semantics. The main reason for our choice, however, is that it again outlines the connection between PL (or ITL) and classic temporal logics. The embedding of PL into QLTL is also a counter position to the embedding of LTL into PL discussed in the previous section.

QLTL: extends LTL slightly by introducing quantification [17], yielding the full power of ω-regular expressions. QLTL formulas are described by the following grammar:

$$\varphi ::= p \mid \neg\varphi \mid \varphi\vee\varphi \mid \bigcirc\varphi \mid \varphi\,\mathcal{U}\varphi \mid \exists x.\ \varphi$$

Likewise, the semantics of QLTL extends the semantics of LTL by introducing a rule for quantification,

$$\sigma;k \models \exists x.\varphi \text{ iff } \exists\sigma_x : \omega \to 2^{\{x\}}.\ \sigma\oplus\sigma_x;k \models \varphi,$$

where $\sigma\oplus\sigma_x : n \mapsto \sigma(n) \uplus \sigma_x(n) \forall n \in \omega$, to the inductive definition of the semantics of a formula.

While the quantifiers extend the expressiveness, quantification does not seem to be a concept that lends itself to human system analysts. We freely confess that we would not trust ourselves when writing a QLTL specification, in particular with nested quantification under the scope of temporal operators.

A. Embedding Pop Logic into QLTL

The embedding of PL into QLTL builds on a simple observation: The truth of a sub-formula depends on the interval stack. In the interval stack, each interval but the interval at the bottom of the interval stack is introduced by a particular *chop* operator.

Each *chop* operator introduces two different intervals for its left and right sub-formulas and we index the interval in the interval stack with the respective sub-formula. That is, for a sub-formula of a specification φ that is a *chop* formula $\psi;\vartheta$, we index with ψ and ϑ, respectively. We index the interval at the bottom of the stack with φ.

An interval can be encoded by the existence of a proper assignment of truth values to *interval propositions* that are evaluated to true within the interval and to false outside of the interval. We use p_ψ to denote the truth values of an interval I_ψ. This way, the existence of a suitable *chop* of an interval can be translated to a QLTL formula that intuitively says 'there is a chopping point for the given interval'. The respective encoding requires that (1) the new intervals are proper intervals, (2) the last position of the left interval coincides with the first position of the right interval, and (3) the union of these intervals coincides with the chopped interval.

As the interpretation of a sub-formula depends—through the interval stack—on its position in the formula tree, its translation needs to take this position into account. We call the complete formula φ and all direct sub-formulas ϑ and ϑ' of a *chop* sub-formula $\vartheta;\vartheta'$ of φ *interval identifiers*. φ can be used to identify the interval $[0,|\sigma|[$ at the bottom of our interval stack, while ϑ and ϑ' can be used to identify the left and right interval, respectively, which is added by the *chop* operation in $\vartheta;\vartheta'$ to the interval stack. Note that ϑ and ϑ' are defined by their positions in the formula tree; syntactically identical sub-formulas of φ have to be distinguished.

For a sub-formula ψ of φ, we first look for the relevant interval identifier, which is the index of the top interval in the stack under which ψ is interpreted. The *relevant interval identifier* of ψ can be found by, starting at ψ, walking the formula tree upwards towards its root. While walking up, we maintain a counter, which is initially 0. The counter is incremented each time we pass a *pop* operator '$\uparrow$' and decremented each time we pass a *chop* operator '$;$', reflecting the pushing and popping of the interval stack. The first interval identifier we pass (including ψ itself) with counter value 0 is the *relevant interval identifier* of ψ, denoted $\widehat{\psi}$.

In other words: if, for a true sub-formula of a specification φ, $\mathsf{par}(\psi)$ denotes the sub-formula of φ whose direct sub-formula ψ is, then the following holds. For a specification φ, we set $\widehat{\varphi}=\varphi$, for a *chop* formula $\psi=\vartheta;\vartheta'$, we set $\widehat{\psi}=\widehat{\mathsf{par}(\psi)}$, $\widehat{\vartheta}=\vartheta$, and $\widehat{\vartheta'}=\vartheta'$, and for a formula ψ that appears as a direct sub-formula of $\psi'=\uparrow\psi$, we set $\widehat{\psi}=\widehat{\mathsf{par}(\psi')}$. Otherwise, if ψ is a direct sub-formula of ψ', we set $\widehat{\psi}=\widehat{\psi'}$.

Using these terms, we define our translation κ from PL to QLTL as follows:

- leaves p (including $\top$) of the formula tree are strengthened by a claim that the current position is within the top interval of the interval stack, reflected by the respective interval proposition: a sub-formula $\psi=\top$ is translated to $\kappa(\psi)=p_{\widehat{\psi}}$ and an atomic proposition $\psi=q$ is translated to $\kappa(\psi)=p_{\widehat{\psi}}\wedge q$;
- a sub-formula $\psi=\neg\vartheta$ is translated to $\kappa(\psi)=\neg\kappa(\vartheta)$ and a sub-formula $\psi=\vartheta\vee\vartheta'$ is translated to $\kappa(\psi)=\kappa(\vartheta)\vee\kappa(\vartheta')$ (all boolean connectives can be translated in this simple manner, note that $\widehat{\vartheta}=\widehat{\vartheta'}=\widehat{\psi}$);
- a sub-formula $\psi=\bigcirc\vartheta$ is translated to $\kappa(\psi)=\bigcirc(\kappa(\vartheta))$,
- a sub-formula $\psi=\uparrow\vartheta$ is translated to[1] $\kappa(\vartheta)$, and
- a sub-formula $\psi=\vartheta\ ;\ \vartheta'$ is translated to $\exists p_\vartheta,p_{\vartheta'}.\ \kappa(\vartheta)\wedge\gamma(p_\vartheta,p_{\vartheta'},p_{\widehat{\psi}})\wedge\Diamond(\kappa(\vartheta')\wedge p_\vartheta\wedge p_{\vartheta'})$, where $\gamma(p,q,r)\ =\ p\wedge\big(\Box((p\vee q)\leftrightarrow r)\big)\wedge\big(\Diamond(p\wedge q)\big)\wedge\Box(q\rightarrow\bigcirc\neg\Diamond p)$ simply checks if p and q define two intervals that properly chop the interval defined by r.

In order to ease the proof that a PL formula φ and its translation $\kappa(\varphi)$ to QLTL are semantically equivalent (that is, have the same models), we introduce a natural extension of the QLTL semantics to interval stacks. Assuming without loss of generality that $p_0,p_1,\dots,p_n$ are fresh propositional variables, we denote with $\Pi_{I_0,\dots,I_n}=\Pi\uplus\{p_0,\dots,p_n\}$ an extended set of atomic propositions. For a word $\sigma=[0,|\sigma|[\mapsto 2^\Pi$, we denote by $\overline{\sigma;I_0,\dots,I_n}$ the ω-word with

- $\overline{\sigma;I_0,\dots,I_n}(k)\cap\Pi=\sigma(k)$ and
- $p_i\in\overline{\sigma;I_0,\dots,I_n}(k)$ if, and only if, $k\in I_i$.

Finally, we use $\overline{\sigma}\models\varphi$ as an abbreviation for $\overline{\sigma;[0,|\sigma|[};0\models\varphi$.

Note that there is a difference between the bottom element of the stack and the remaining elements: While the bottom element—$[0,|\sigma|[$—reflects the true length of a finite or infinite input word σ, the other elements of the stack are introduced during the interpretation of the φ on σ. Thus, the bottom element and the atomic proposition describing it (no matter if named p_0, p_φ, or $p_{\widehat{\varphi}}$) are assumed to be explicitly given.

Theorem 5.1: For a PL formula φ and a word σ, $\sigma\models\varphi$ holds if, and only if, $\overline{\sigma}\models\kappa(\varphi)$.

Proof: By induction over the structure of the formula, we show that $\sigma;I_\varphi,\dots,I_{\widehat{\psi}};k\models\psi$ holds for every sub-formula ψ of φ if, and only if, $\overline{\sigma;I_\varphi,\dots,I_{\widehat{\psi}}};k\models\kappa(\psi)$.

[1]This translation might look a bit surprising at first glance, because it may convey the impression that the $\uparrow$ operator has no effect. However, it is reflected in the $\widehat{\theta}$ operations that identify the relevant interval identifiers—and hence the interval propositions occurring in the translation.

The *induction basis* is trivial, as the claim holds by definition for the atomic propositions (including $\top$).

The *induction step* for boolean connectives is also trivial. For the next operation, we have

$$\begin{aligned}
&\sigma; I_\varphi, \ldots, I_{\widehat{\psi}}; k \models \bigcirc \psi \\
\Leftrightarrow^{PL} \quad & \sigma; I_\varphi, \ldots, I_{\widehat{\psi}}; k+1 \models \psi \\
\Leftrightarrow^{IH} \quad & \overline{\sigma; I_\varphi, \ldots, I_{\widehat{\psi}}}; k+1 \models \kappa(\psi) \\
\Leftrightarrow^{QLTL} \quad & \overline{\sigma; I_\varphi, \ldots, I_{\widehat{\psi}}}; k \models \bigcirc \kappa(\psi) =_\kappa^{def} \kappa(\bigcirc \psi).
\end{aligned}$$

The induction step for the pop operator is also simple:

$$\begin{aligned}
&\sigma; I_\varphi, \ldots, I_{\widehat{\psi}}, I_{\widehat{\uparrow\psi}}; k \models \uparrow \psi \\
\Leftrightarrow^{PL} \quad & \sigma; I_\varphi, \ldots, I_{\widehat{\psi}}; k \models \psi \\
\Leftrightarrow^{IH} \quad & \overline{\sigma; I_\varphi, \ldots, I_{\widehat{\psi}}}; k \models \kappa(\psi) \\
\Leftrightarrow_\kappa^{def} \quad & \overline{\sigma; I_\varphi, \ldots, I_{\widehat{\psi}}, I_{\widehat{\uparrow\psi}}}; k \models \kappa(\uparrow \psi)
\end{aligned}$$

For the last equivalence, note that the leading $\uparrow$ operator prevents that the variable $p_{\widehat{\uparrow\psi}}$ occurs in $\kappa(\widehat{\psi})$. (Hence, its valuation does not matter.)

We conclude the inductive proof with the induction step for the *chop* operator, using the abbreviations $I_{\widehat{\psi;\vartheta}} = [b, e[$, $I_\psi = [k, c]$, and $I_\vartheta = [c, e[$. We then get:

$$\begin{aligned}
&\sigma; I_\varphi, \ldots, I_{\widehat{\psi;\vartheta}}; k \models \psi; \vartheta \\
\Leftrightarrow^{PL} \quad & \exists c \in [k, e[.\ \sigma; I_\varphi, \ldots, I_{\widehat{\psi;\vartheta}}, I_\psi; k \models \psi \text{ and} \\
& \qquad \sigma; I_\varphi, \ldots, I_{\widehat{\psi;\vartheta}}, I_\vartheta; c \models \vartheta \\
\Leftrightarrow^{IH} \quad & \exists c \in [k, e[.\ \overline{\sigma; I_\varphi, \ldots, I_{\widehat{\psi;\vartheta}}, I_\psi}; k \models \kappa(\psi) \text{ and} \\
& \qquad \overline{\sigma; I_\varphi, \ldots, I_{\widehat{\psi;\vartheta}}, I_\vartheta}; c \models \kappa(\vartheta) \\
\Leftrightarrow^{QLTL} \quad & \overline{\sigma; I_\varphi, \ldots, I_{\widehat{\psi;\vartheta}}}; k \models \exists p_\psi, p_\vartheta.\ \kappa(\psi) \wedge \\
& \qquad \gamma(p_\psi, p_\vartheta, p_{\widehat{\psi;\vartheta}}) \wedge \Diamond(\kappa(\vartheta) \wedge p_\psi \wedge p_\vartheta) \\
\Leftrightarrow_\kappa^{def} \quad & \overline{\sigma; I_\varphi, \ldots, I_{\widehat{\psi;\vartheta}}}; k \models \kappa(\psi; \vartheta).
\end{aligned}$$

■

Note that we have to adjust $\kappa(\varphi)$ if we are interested in the satisfiability or validity problem as there is no word σ to start with. If we are only interested in infinite models, we can replace all occurrences of p_φ by $\top$, if are interested only in finite models, we can use $\kappa(\varphi) \wedge (p_\varphi\, \mathcal{U} \Box \neg p_\varphi)$, and if we want to allow for both we can use their disjunction.

B. Complexity of Pop Logic

Having established a linear translation of well formed PL formulas to equivalent QLTL formulas, we can re-use the decision procedures of QLTL to decide PL specifications. We thus inherit the non-elementary decision procedures for the satisfiability and word problem of QLTL from [17]. Matching hardness results are inherited from the syntactic sub-logic ITL of PL.

Theorem 5.2: The satisfiability, word, and model checking problems for PL are non-elementary decidable, and hard for this class even if restricted to finite or infinite words.

Proof: With κ, we have established a linear translation from PL to equivalent QLTL specifications. We therefore inherit the non-elementary decision procedures of QLTL for the satisfiability problem and the word problem for finite and ω-regular words.

For the lower bounds, we can use the matching hardness results of the syntactic sub-logic ITL for both finite and infinite [18] words. ■

VI. A PSPACE-Complete Subset of PL

While Section V establishes the decidability and complexity of PL, the non-elementary complexity of PL is not appealing. But we have also seen that the popular and inexpensive temporal logic LTL can be viewed as a de-facto syntactic sub-logic of PL, which implies that relevant sub-logics of PL are in PSPACE.

In this section, we define a wider fragment of PL—which includes LTL—that is still decidable in polynomial space. Depending on personal preferences, this sub-logic can be considered as a restriction of PL or as an extension of LTL.

To get an intuition for the extension, we adjust our translation κ with complexity considerations in mind. We approach finding a PSPACE fragment by looking at the translation of LTL, starting with the sub-logic of LTL that uses the *eventually* operator instead of *until*.

In LTL, $\Diamond \psi$ is an abbreviation for $\top; \uparrow \psi$, and this sub-formula translates to $\exists p_\top, p_{\uparrow\psi} \kappa(\top) \wedge \gamma(p_\top, p_{\uparrow\psi}, p_{\widehat{\top;\uparrow\psi}}) \wedge \Diamond(p_{\uparrow\psi} \wedge \kappa(\uparrow \psi))$. γ states that $p_\top$ and $p_{\uparrow\psi}$ do chop $p_{\widehat{\top;\uparrow\psi}}$, and neither $p_\top$ nor $p_{\uparrow\psi}$ occur in $\kappa(\uparrow \psi)$. In such a situation, we can avoid the existential quantification and simply translate $\kappa(\top; \uparrow \psi)$ to $p_{\widehat{\top;\uparrow\psi}} \wedge \Diamond(p_{\widehat{\top;\uparrow\psi}} \wedge \kappa(\uparrow \psi))$.

Adjusting κ.: Avoiding quantification is a very useful tool, because without quantification the target language is LTL instead of QLTL. In the remainder of the paragraph we therefore discuss properties of sub-formulas which allow either avoiding quantification, or at least to use it in a monotone and inexpensive way. This raises the question of whether or not we can avoid the introduction of quantification. For this we adjust our translation κ from PL to QLTL to $\overline{\kappa}$, where we only touch the rules for *chop* sub-formula $\psi; \vartheta$. For them we give two rules for cases, where the use of quantification can be avoided completely, and a fall-back rule that only avoids the introduction of the p_ϑ.

This *fall-back rule*, which is only used if neither of the two rules introduced later in this section apply, translates $\psi; \vartheta$ to $\exists p_\psi . \overline{\kappa}(\psi) \wedge \big((p_\psi \wedge p_{\widehat{\psi;\vartheta}})\, \mathcal{U} (p_\psi \wedge p_{\widehat{\psi;\vartheta}} \wedge \overline{\kappa}(\vartheta)[p_\vartheta \mapsto p_{\widehat{\psi;\vartheta}}] \wedge \bigcirc \Box \neg p_\psi)\big)$.

This fall-back rule is also a preparation for the more powerful rules in the special cases where quantification over p_ψ can also be avoided, as it uses the concept of implicitly expressing the existence of a reasonable representation of the second interval. The right side of the until marks the situation at the chopping point: p_ψ holds, but ceases to do so in the next position, and the chopping point needs to be within the interval identified by $p_{\widehat{\psi;\vartheta}}$. p_ϑ has to coincide with $p_{\widehat{\psi;\vartheta}}$ from the chopping point onwards (and cannot appear in $\overline{\kappa}(\psi)$). We can therefore check the correctness of $\overline{\kappa}(\vartheta)[p_\vartheta \mapsto p_{\widehat{\psi;\vartheta}}]$ instead of $\overline{\kappa}(\vartheta)$ at the chopping point and avoid the introduction of p_ϑ.

Unfortunately, no similarly general technique for the left side of a chop operation can exist, because this would apply a lower complexity of Pop Logic. But the observation that the introduction of p_ϑ can be avoided allows us to concentrate on p_ψ when seeking sub-languages in PSPACE. Our focus is on cases where the chopping point itself can be guessed.

We can avoid the introduction of (and quantification over) p_ψ if we can identify a suitable chopping point within the interval identified by $p_{\widehat{\psi;\vartheta}}$. The simplest case in which this is possible is, of course, if p_ψ does not occur in $\overline{\kappa}(\psi)$. This is, for example, the case in the de-facto sub-language that resembles LTL. But we can do more.

We call a sub-formulas ψ *long for an interval proposition* p if it holds that, if ψ holds if p identifies the non-empty interval $[b,e[$, then it holds if p identifies the interval $[b,e'[$ for all $e' \geq e$. Likewise, we call a sub-formula ψ *short for an interval proposition* p if it holds that, if ψ holds if p identifies the non-empty interval interval $[b,e[$, then it holds if p identifies the interval $[b,e'[$ for all $b < e' \leq e$. We call a sub-formula ψ *finite* if the interval stack $I_\varphi,\ldots,I_{\widehat{\psi}}$ under which ψ is evaluated contains an interval I_ϑ stemming from the left side of a chop sub-formula $\vartheta;\vartheta'$ of φ. (Finite simply guarantees that the respective interval is finite; in the finite semantics, all sub-formulas are finite.)

There are simple sufficient conditions for formulas to be long or short. In particular, a sub-formula is both long and short for p_ψ if p_ψ does not occur in the translation $\overline{\kappa}(\psi)$. This holds, for example, for $\psi = \uparrow \psi'$. Also, atomic propositions are short and long, longness is preserved by next operations, and shortness and longness are preserved by positive boolean combination, and shortness and longness are toggled by negation. (That is, $\neg\psi$ is long if ψ is short and $\neg\psi$ is short if ψ is long.)

We can use a simplifying translation for $\psi;\vartheta$ in two cases: if ψ is both short and long, and if ψ is finite and long. If we can identify the first case (to which we give preference to have an unambiguous rule) by the rules above we can define $\overline{\kappa}(\psi;\vartheta)$ to

$$\overline{\kappa}(\psi)[p_\psi \mapsto p_{\widehat{\psi;\vartheta}}] \wedge \Diamond \left(p_{\widehat{\psi;\vartheta}} \wedge \kappa(\vartheta)[p_\vartheta \mapsto p_{\widehat{\psi;\vartheta}}]\right).$$

The right conjunct requires that there is a chopping point such that ϑ holds in the right interval. (Where, again, p_ϑ and $p_{\widehat{\psi;\vartheta}}$ coincide on that interval.) Shortness and longness together imply that ψ holds on the left interval if, and only if, it holds on the interval identified by $p_{\widehat{\psi;\vartheta}}$; it is therefore safe to use this interval instead.

If $\overline{\kappa}(\psi)$ is long for p_ψ and $\psi;\vartheta$ finite, we can use the well defined last possible chopping point—the last point where $\alpha = \Diamond(p_{\widehat{\psi;\vartheta}} \wedge \overline{\kappa}(\vartheta)[p_\vartheta \mapsto p_{\widehat{\psi;\vartheta}}])$ holds. We set $\overline{\kappa}(\psi;\vartheta) = \kappa(\psi)[p_\psi \mapsto \alpha] \wedge \alpha$.

Finally, we call *chop* operators *safe* if the sub-formula they govern is not translated to a quantified formula by $\overline{\kappa}$, and left if, on the path from the root to this formula, only left turns were taken at every *chop* operator. As usual, we call these formulas positive if they are bound by an even number of negations, and negative otherwise.

A first observation is that, provided all *chop* operators are safe, the discussed translation goes to LTL, providing for a PSPACE complexity.

Theorem 6.1: The PL subset where all occurring chops are safe is a sub-logic of PL that contains LTL as a de-facto syntactic subset. It has a PSPACE-complete satisfiability, validity, word, and model checking problem.

Proof: In the formulas from the LTL fragment of PL all *chop* operations are safe and the translation of PL formulas where all chop operations are safe go to LTL. (Which also implies that this subset is no more expressive than LTL.)

Note that the formula tree of the target formula is not necessarily polynomial in the size of the source specification, because the ‘long & finite’ case adds multiple occurrences of $\Diamond(p_{\widehat{\psi;\vartheta}} \wedge \overline{\kappa}(\vartheta)[p_\vartheta \mapsto p_{\widehat{\psi;\vartheta}}])$ in the translation of a *chop* sub-formula $\psi;\vartheta$. However, if we represent the formula as a directed acyclic graph (DAG), then this blow-up does not occur: there are merely multiple edges pointing to the same node. Hence, the number of *sub-formulas* stays linear in the size of the PL formula. ■

This also outlines a difference between PL and LTL that cannot be bridged: While the valuation of a sub-formula of an LTL formula is independent of its position in the formula tree, the valuation of sub-formulas in PL depend on the interval stack under which they are interpreted. We therefore cannot safely assume that the valuation is similar unless they must refer to the same interval stack when evaluated. (Which is the case for two sub-formulas ψ_1 and ψ_2 if $\widehat{\psi_1} = \widehat{\psi_2}$.)

We can extend the relative tractability to larger classes of languages: There is no need for the target formula to be in LTL. If we are interested in the satisfiability or word problem, then it suffices to reside in prenex QLTL with only existential quantification.

Theorem 6.2: The PL subset where all occurring unsafe chops are positive left formulas is a sub-logic with a PSPACE-complete satisfiability and word problem.

Proof: It is easy to show by induction that, provided all unsafe *chop* operations in a PL formula φ are positive left formulas, the target formula $\overline{\kappa}(\varphi)$ has a formula DAG where no quantifier is bound by a temporal operator and all quantifiers occur positively. We can hence re-write $\overline{\kappa}(\varphi)$ to prenex normal form by simply moving all quantifiers to the front, which does not affect the size of the formula DAG.

For formulas of this type, the satisfiability and word problem are the same as for LTL, because leading existential quantification does not affect them. ■

Consequently, the co-problems of the negation of these formulas are also in PSPACE.

Corollary 6.3: The PL subset where all occurring unsafe chops are negative left formulas is a sub-logic with a PSPACE-complete validity, word, and model-checking problem.

VII. Extended Pop Logic

The logic PL introduced in Section II extends the most basic version of ITL. In order to reach the full expressiveness of regular and ω regular expressions, we extend the widespread version of ITL with *star* to *Extended Pop Logic* (EPL) in this section. This provides us with the following grammar:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \varphi;\varphi \mid \uparrow\varphi \mid \varphi^*$$

Like the *chop* operator, the *star* operator pushes new intervals in the stack. We extend the well formedness to: there exists no node in the formula tree of φ such that strictly more *pop* operators than *chop* and *star* operators (counted together) occur on the way from the root of the formula tree to this node. For the *star* operator, we use the following semantics:

$\sigma; I_0, \ldots, I_n, [b,e[; k \models \varphi^*$ iff
$k \in [b,e[$ and there exist $m \in \mathbb{N}$ integers
$k = r_0 < r_1 < \ldots < r_m = e$ such that
$\forall i \in [0,m[.\ \sigma; I_0, \ldots, I_n, [b,e[, [r_i, r_{i+1}+1[; r_i \models \varphi$.

Lemma 7.1: EPL can be embedded in QLTL.

Proof: To prove this, it suffices to extend the function κ and Theorem 5.1 accordingly. The extension is a straight forward generalization of the rules (and proof) for the *chop*, with the difference that we mark finally many intervals instead of marking exactly two. Technically this can be done by guessing the fringes of the intervals (existential quantification) and showing that all intervals defined by these fringes are suitable (universal quantification). ■

Theorem 7.2: EPL is decidable with non-elementary complexity. It can express exactly the regular and ω-regular languages.

Proof: The non-elementary lower bound and the expressiveness of EPL can be inferred from the respective lower bounds and expressiveness of ITL [18], [19]. The decidability and upper bound on the complexity are implied by Lemma 7.1. ■

VIII. Discussion

We have introduced Pop Logic, an interval temporal logic that synthesises the concepts of classic and interval temporal logic. Pop Logic extends ITL only very modestly by introducing a *pop* operation that revokes the scoping of a previous *chop* operation.

This modest extension preserves the most important property of ITL: Pop Logic remains simple and intuitive. But while the extension is sufficiently modest to preserve this important property, it is powerful enough to include not only ITL, but also *the* classic temporal logic LTL, as syntactic sub-logics.

As a result, we think that PL provides a useful bridge between classic and interval temporal logics: It allows for using both logics individually for sub-problems, and yet provides a simple framework to treat these specifications. It therefore allows for naturally integrating aspects expressed by interval temporal logic in a predominantly classic LTL specification, or, less often, LTL aspects in a predominantly ITL specification.

In addition, we have characterised a relatively inexpensive (PSPACE-complete) fragment of PL, which includes and extends LTL. We consider the identification of such sub-logics useful, as they can guide a system analyst in devising her specification: When the expensive steps can easily be identified, the decision to keep the specification in the current form—and pay the price in form of increased complexity—or to invest more work in reformulating the specification can be made informed.

Acknowledgement

Sven Schewe is supported by the Engineering and Physical Science Research Council (EPSRC) through the grant EP/H046623/1 'Synthesis and Verification in Markov Game Structures'. Cong Tian is supported by the NSFC Grant No. 61003078, 91018010, and 60910004, 973 Program Grant No. 2010CB328102, and ISN Lab Grant No. ISN1102001. Both Sven Schewe and Cong Tian are corresponding authors.

References

[1] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-Time Temporal Logic. Journal of the ACM 49(5): pages 672–713, 2002.

[2] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. Acta Informatica 20: pages 207–226, 1983.

[3] A. Chandra, J. Halpern, A. Meyer, and R. Parikh. Equations between regular terms and an application to process logic. Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing (STOC 1981). pages 384–390, 1981.

[4] E. M. Clarke and E. A. Emerson. Design and syntesis of synchronization skeletons using branching time temporal logic. In Proceedings of the IBM Workshop on Logics of Programs (LP 1981). pages 52–71, 1982.

[5] Z. Duan. An Extended Interval Temporal Logic and A Framing Technique for Temporal Logic Programming. PhD thesis, University of Newcastle Upon Tyne, May 1996.

[6] Z. Duan, M. Koutny, and C. Holt. Projection in Temporal Logic Programming. In Proceedings of Logic Programming and Automated Reasoning (LPAR 94). pages 333–344, 1994.

[7] Z. Duan, C. Tian, and L. Zhang. A Decision Procedure for Propositional Projection Temporal Logic with Infinite Models. Acta Informatica, 45(1): pages 43–78, 2008.

[8] E. A. Emerson. Temporal and Modal Logic. Computer Science Department, University of Texas at Austin, USA, 1995.

[9] B. Finkbeiner and S. Schewe. Coordination Logic. In Proceedings of the 19th Annual Conference of the European Association for Computer Science Logic (CSL 2010). pages 305–319, 2010.

[10] J. Halpern, Z. Manna, and B. Moszkowski. A hardware semantics based on temporal intervals. In Proceedings of the tenth International Colloquium on Automata, Languages and Programming (ICALP 1983). pages 278–291, 1983.

[11] D. Harel, D. Kozen, and R. Parikh. Process logic: expressiveness, decidability, completeness. Journal of Computer and System Sciences 2: pages 144–170, 1982.

[12] L. Lamport. The Temporal Logic of Actions. ACM Transactions on Programming Languages and Systems 16(3): pages 872–923, 1994.

[13] B. Moszkowski. Reasoning about digital circuits. Ph.D Thesis, Department of Computer Science, Stanford University. TRSTAN-CS-83-970, 1983.

[14] B. Moszkowski. Compositional reasoning about projected and infinite time. In Proceeding of the First IEEE International Conference on Enginneering of Complex Computer Systems (ICECCS 1995), pages 238–245, 1995.

[15] A. Pnueli. The temporal logic of programs. In Proceedings of the 18th IEEE Symposium on Foundations of Computer Science (FOCS 1977). pages 46–57, 1977.

[16] T. Schlipf, T. Buechner, R. Fritz, M. Helms, and J. Koehl. Formal verification made easy. IBM Journal of Research and Development, 41(4-5): pages 567–576, 1997.

[17] A. P. Sistla. Theoretical issues in the design and verification of distributed systems. PhD thesis, Harvard University, 1983.

[18] C. Tian and Z. Duan. Complexity of propositional projection temporal logic with star. Mathematical Structures in Computer Science 19(1): pages 73–100, 2009.

[19] C. Tian and Z. Duan: Expressiveness of propositional projection temporal logic with star. Theoretical Computer Science 412(18): pages 1729–1744, 2011.

[20] M. Y. Vardi: Branching vs. Linear Time: Final Showdown. In Proceedings of the Seventh International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001). pages 1–22, 2001.

[21] M. Vardi and P. Wolper. Automata Theoretic Techniques for Modal Logics of Programs, Journal of Computer and System Sciences 32(2): pages 183–221, 1986.

[22] P. L. Wolper. Temporal logic can be more expressive. Information and Control, 56(1/2): pages 72–99, 1983.

Appendix: Embedding LTL in Pop Logic

In this appendix, we show the semantic equivalence of the LTL fragment of PL in the PL semantics and the common LTL semantics. Note that, for this comparison, the bottom element of the stack is always $\omega = [0, \infty[$.

The inductive proof below makes use of the flatness of the translation.

As *induction basis*, the following obviously holds for all atomic propositions (including $\top$) $p \in \Pi$:

$$\sigma;\omega;k \models_{PL} p \Leftrightarrow^{PL} p \in \sigma(k) \Leftrightarrow^{LTL} \sigma;k \models_{LTL} p.$$

For the *induction step*, we have to cover boolean connectives, next, and until. Boolean connectives and next are trivial:

$$\begin{array}{lll}
\sigma;\omega;k & \models_{PL} & \varphi \vee \psi \\
 & \Leftrightarrow^{PL} & \sigma;\omega;k \models_{PL} \varphi \text{ or } \sigma;\omega;k \models_{PL} \psi \\
 & \Leftrightarrow^{IH} & \sigma;k \models_{LTL} \varphi \text{ or } \sigma;k \models_{LTL} \psi \\
 & \Leftrightarrow^{LTL} & \sigma;k \models_{LTL} \varphi \vee \psi, \\
\sigma;\omega;k & \models_{PL} & \neg\varphi \\
 & \Leftrightarrow^{PL} & \sigma;\omega;k \not\models_{PL} \varphi \\
 & \Leftrightarrow^{IH} & \sigma;k \not\models_{LTL} \varphi \\
 & \Leftrightarrow^{LTL} & \sigma;k \models_{LTL} \neg\varphi, \text{ and} \\
\sigma;\omega;k & \models_{PL} & \bigcirc\varphi \\
 & \Leftrightarrow^{PL} & \sigma;\omega;k+1 \models_{PL} \varphi \\
 & \Leftrightarrow^{IH} & \sigma;k+1 \models_{LTL} \varphi \\
 & \Leftrightarrow^{LTL} & \sigma;k \models_{LTL} \bigcirc\varphi,
\end{array}$$

Finally, for the until we have:

$$\begin{array}{l}
\sigma;\omega;k \models_{PL} \psi\,\mathcal{U}\vartheta \\
\quad \Leftrightarrow^{def}_{\mathcal{U}} \sigma;\omega;k \models_{PL} \Big(\neg\big(\top;\uparrow\neg\uparrow(\psi\vee\vartheta)\big)\Big);\uparrow\vartheta \\
\quad \Leftrightarrow^{PL} \exists l \geq k.\sigma;\omega,[k,l];k \models_{PL} \neg\big(\top;\uparrow\neg\uparrow(\psi\vee\vartheta)\big) \\
\qquad\quad \text{and } \sigma;\omega,[l,\infty[;l \models_{PL}\uparrow\vartheta \\
\quad \Leftrightarrow^{PL} \exists l \geq k.\sigma;\omega,[k,l];k \not\models_{PL} \big(\top;\uparrow\neg\uparrow(\psi\vee\vartheta)\big) \\
\qquad\quad \text{and } \sigma;\omega;l \models_{PL} \vartheta \\
\quad \Leftrightarrow^{PL} \exists l \geq k.\ \nexists m \in [k,l].\ \sigma;\omega,[k,l],[k,m];k \models_{PL} \top \\
\qquad\quad \text{and } \sigma;\omega,[k,l],[m,l];m \models_{PL}\uparrow\neg\uparrow(\psi\vee\vartheta) \\
\qquad\quad \text{and } \sigma;\omega;l \models_{PL} \vartheta \\
\quad \Leftrightarrow^{PL}_{\top} \exists l \geq k.\ \nexists m \in [k,l].\ \sigma;\omega,[k,l],[m,l];m \models_{PL} \\
\qquad\quad \uparrow\neg\uparrow(\psi\vee\vartheta)\sigma;\omega;l \models_{PL} \vartheta \\
\quad \Leftrightarrow^{PL} \exists l \geq k.\ \nexists m \in [k,l].\ \sigma;\omega,[k,l];m \models_{PL} \neg\uparrow(\psi\vee\vartheta) \\
\qquad\quad \text{and } \sigma;\omega;l \models_{PL} \vartheta \\
\quad \Leftrightarrow^{PL} \exists l \geq k.\ \nexists m \in [k,l].\ \sigma;\omega,[k,l];m \not\models_{PL}\uparrow(\psi\vee\vartheta) \\
\qquad\quad \text{and } \sigma;\omega;l \models_{PL} \vartheta \\
\quad \Leftrightarrow \exists l \geq k.\forall m \in [k,l].\ \sigma;\omega,[k,l];m \models_{PL}\uparrow(\psi\vee\vartheta) \\
\qquad\quad \text{and } \sigma;\omega;l \models_{PL} \vartheta \\
\quad \Leftrightarrow^{PL} \exists l \geq k.\forall m \in [k,l].\ \sigma;\omega;m \models_{PL} \psi\vee\vartheta \\
\qquad\quad \text{and } \sigma;\omega;l \models_{PL} \vartheta \\
\quad \Leftrightarrow^{PL} \exists l \geq k.\forall m \in [k,l].\ \sigma;\omega;m \models_{PL} \psi \\
\qquad\quad \text{or } \sigma;\omega;m \models_{PL} \vartheta \text{ and } \sigma;\omega;l \models_{PL} \vartheta \\
\quad \Leftrightarrow^{IH} \exists l \geq k.\forall m \in [k,l].\ \sigma;m \models_{LTL} \psi \\
\qquad\quad \text{or } \sigma;m \models_{LTL} \vartheta \text{ and } \sigma;l \models_{LTL} \vartheta \\
\quad \Leftrightarrow \exists l \geq k.\sigma;l \models_{LTL} \vartheta \text{ and } \forall m \in [k,l[.\ \sigma;m \models_{LTL} \psi \\
\quad \Leftrightarrow^{LTL} \sigma;k \models_{LTL} \psi\,\mathcal{U}\vartheta.
\end{array}$$