Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 70 (2008) 31–61
Science of Computer Programming
www.elsevier.com/locate/scico

Framed temporal logic programming☆

Zhenhua Duan[a,*], Xiaoxiao Yang[a], Maciej Koutny[b]

[a] *Institute of Computing Theory and Technology, Xidian University, Xi'an 710071, PR China*
[b] *School of Computing Science, University of Newcastle, Newcastle upon Tyne NE1 7RU, UK*

Received 4 June 2006; received in revised form 16 May 2007; accepted 3 September 2007
Available online 7 October 2007

Abstract

A Projection Temporal Logic is discussed and some of its laws are given. After that, an executable temporal logic programming language, called Framed Tempura, is formalized. A minimal model-based approach for framing in temporal logic programming is presented. Since framing destroys monotonicity, canonical models – used to define the semantics of non-framed programs – are no longer appropriate. To deal with this, a minimal model theory is developed, using which the temporal semantics of framed programs is captured. The existence of a minimal model for a given framed program is demonstrated. A synchronous communication mechanism for concurrent programs is provided by means of the framing technique and minimal model semantics.

Keywords: Temporal logic programming; Framing; Minimal model; Monotonicity; Synchronization; Communication; Semantics

1. Introduction

Temporal logic [12,19,20,23,21] has been proposed for the purpose of verifying properties of programs. Before that, verification suffered from a disadvantage that different languages (and thus different semantic domains) had been used for writing programs, for writing about their properties and for writing about whether and how a program satisfies a given property [20]. One way to address this problem is to use the same language as much as possible. It was therefore suggested that a subset of temporal logic be used as the foundational basis for a programming language [1,13,14,23,26]. This means that writing programs, specifying their properties and verifying those properties, could all be treated within the same notation. There are some aspects of programming in temporal logics that have not yet received sufficient attention, notably the problems of framing, synchronization and communication. As synchronization and communication are closely related to framing, we will first discuss some general issues regarding framing techniques in Temporal Logic Programming (TLP).

☆ This research was supported by NSFC Grant 60433010 and 60373103, and Defence Pre-Research Project of China NO. 51315050105.
* Corresponding author.
E-mail addresses: zhhduan@mail.xidian.edu.cn, zhenhua_d@yahoo.com (Z. Duan), xxyang@mail.xidian.edu.cn (X. Yang), maciej.koutny@newcastle.ac.uk (M. Koutny).

0167-6423/$ - see front matter
doi:10.1016/j.scico.2007.09.001

1.1. Motivation

Introducing framing to TLP is motivated by both practical and theoretical considerations; in particular, improving the efficiency of a program and synchronizing communication between parallel processes. Let us begin with an example in Tempura [23]:

$$\text{PROG I}: \quad (x = 1) \wedge ((y := 2)\,;\ (y := x + y))$$

where "**;**" (chop) is a sequential operator, "=" assigns a value to a variable at the current state, and ":=" does the same at the next state.

The program only tells us that $x = 1$ at the initial state of an interval over which the program is interpreted, and that $y = 2$ at the second state. One might expect (or even require) that $y = 3$ (the sum of the values of x and y) at the third state, but the program does not guarantee this. The reason is that x is unspecified at the second state, and so y is unspecified at the third state (as well as in the initial state). There are several ways to achieve the desired effect, and an ad hoc fix to the problem is to make the stability of variables' values explicit. The above example would then be rewritten as

$$\text{PROG II}: \quad (x = 1) \wedge \Box(\mathsf{more} \rightarrow (\bigcirc x = x)) \wedge ((y := 2)\,;\ (y := x + y))$$

where $\Box$ is the modal operator associated with the temporal "always", $\bigcirc$ is the "next" operator, and more means that the current interval is not yet over.

Now x is assigned its current value repeatedly, from one state to another, so that its value is inherited. But these additional assignments are tedious and may decrease the efficiency of the program. Although for a small program repeated assignments may be tolerable, in some cases they may be unacceptable. For instance, maintaining in such a way a large array with changing only a few its entries at different times, would lead to a disaster in terms of performance. Moreover, the verification and transformation of programs would also suffer in such a case. It is therefore important to have an efficient method (a *framing technique*) allowing one to carry forward variables' values, from the current state to the next.

Another problem one needs to consider is that of communication between concurrent processes. In TLP languages, such as XYZ/E [26,27] and Tempura [6,23], such a communication is based on shared variables.[1] To effect communication between parallel processes in a shared variables model, a synchronization construct $\mathsf{await}(c)$, or some equivalent, is required [21]. The meaning of $\mathsf{await}(c)$ is simple: it changes no variables, but waits until the condition c becomes true, at which point it terminates.

How could the $\mathsf{await}(c)$ construct be implemented in a TLP like Tempura? To start with, the $\mathsf{halt}(c)$ statement might play a role similar to that of $\mathsf{await}(c)$. However, $\mathsf{halt}(c)$ requires that c become true only at the end of an interval, and it does not prevent the variables from being changed. Thus problems arise whether we adopt repeated or unrepeated assignments, when we attempt to synchronize parallel components. For instance, the program:

$$\text{PROG III}: \quad (x = 0) \wedge \mathsf{halt}(x = 1) \wedge \mathsf{len}(2)$$

where len specifies the temporal length of an interval, is satisfied by any interval comprising three states such that $x = 0$ and $x = 1$ at the first and third state, respectively. On the other hand, if we used repeated assignment, the program:

$$\text{PROG IV}: \quad (x = 0) \wedge \Box(\mathsf{more} \rightarrow (\bigcirc x = x)) \wedge \mathsf{halt}(x = 1) \wedge \mathsf{len}(2)$$

is obviously unsatisfiable. As another example, the program:

$$\text{PROG V}: \quad (x = 0) \wedge \mathsf{halt}(x = 1)$$

is satisfiable over an interval such that $x = 0$ in its first state, and $x = 1$ in the last one. It terminates at some indefinite state where $x = 1$ because x is only defined at the initial state. On the other hand, if we used repeated assignment, the program

$$\text{PROG VI}: \quad (x = 0) \wedge \Box(\mathsf{more} \rightarrow (\bigcirc x = x)) \wedge \mathsf{halt}(x = 1)$$

[1] This should be contrasted with communication by message passing used by, e.g., CCS [22] and CSP [18].

would be forever waiting for $(x = 1)$ to become true. No process acting in parallel can set x to 1, since such an assignment would conflict with $x = 0$.

The above problem is due to the fact that, unlike in conventional programming languages, the values of variables are not inherited automatically from one state to another, and the assignments are not destructive in TLP languages. Modelling an $\mathsf{await}(c)$ construct in TLP requires, on the one hand, a kind of indefinite stability, since it cannot be known at the point of use how long the waiting will be. On the other hand, one must also allow variables to change their values so that an external process can change the boolean guard c. Solving this problem can also be attempted by a suitable framing technique.

1.2. Logic with framing

Framing is concerned with how the value of a variable can be carried from one state to the next. It has been employed by the conventional imperative languages for years. TLP offers no ready-made solution in this respect as variables' values are not assumed to be carried forward.

Considerable attention has been given to framing in the past two decades [6,9,16,17,20,25,27]. However, no consensus has yet emerged as to what is the best underlying semantics of framing operators. For example, in [9] a framing technique based on an explicit frame operator was formalized, together with a specification (modelling) language FTLL. In this paper, we describe a similar approach but at a much greater depth of details. Based on the proposed framing technique, we introduce an executable version of a framed programming language, called Framed Tempura. We investigate the properties of the frame operator and, in particular, the behaviour of concurrent programs under a specific model of framing in which framed and non-framed variables can be mixed.

In our discussion, we use Projection Temporal Logic (PTL), which is an extension of Interval Temporal Logic (ITL) [23]. The key operator used is the new projection operator, $(p_1, \ldots, p_m)$ prj q, which can be thought of as a combination of the original parallel and projection operators in ITL [5,23].[2] Within the PTL framework, we define a new assignment operator ($\Leftarrow$) and an assignment flag (af), and then formalize a framing operator $\mathsf{frame}(x)$. These new constructs are interpreted within a minimal model semantics.

1.3. Framed programs

To make the framing technique useful in practice, we develop an extension of Tempura [23], called Framed Tempura. Using its framing operator, one can specify the framing status of variables throughout an interval in a flexible manner, and to verify properties of a reactive system in a manageable way.

When a framing technique is introduced to TLP, the semantics of a program needs to be carefully reconsidered. The semantics of an imperative program can be captured in an operational or denotational or axiomatic way. In TLP, these kinds of semantics can also be investigated. Since a TLP language, e.g., Tempura, is a subset of the corresponding logic, and this logic has its model theory and axiom system, the semantics of a TLP program can be captured by both the model theory and axiomatic theory. When executed, of course, a program can also be interpreted in an operational way. Actually, the model theory, in some sense, plays a similar role in TLP languages as the denotational semantics theory in imperative programming languages.

To capture the temporal semantics of nonframed Tempura programs, canonical models can be used [6]. However, since framing destroys monotonicity, a program can have different meanings under different canonical models, and so a canonical model may no longer capture the intended meaning of a program. In logic programming languages such as Prolog, negation by failure has been used in programs, and a program is interpreted by the minimal model semantics, or the fixpoint semantics [2]. This has led to the introduction of a similar idea in TLP, and to interpret framed programs faithfully, we will use minimal models. We discuss the semantics of framed programs; in particular, we define the normal form of framed programs and show that a framed program can be transformed into its normal form. We also show that a satisfiable framed program has at least one minimal model.

The synchronous communication construct, $\mathsf{await}(c)$, can easily be defined using the framing operator. Note that an interpreter for the Framed Tempura has been developed using the framing technique presented in this paper; however, it will not be presented here.

[2] PTL subsumes ITL since it can express the chop and projection operators of ITL.

The next section introduces PTL, and Section 3 describes Framed Tempura. The semantics of framed programs is discussed in Section 4. After that we present some programming examples in Section 5. Conclusions are drawn in Section 6, and the Appendix contains proofs of selected results.

2. Projection temporal logic

PTL (Projection Temporal Logic) – an extension of ITL [23] – is a first order temporal logic [19,21] with projection [6,7,10,9].

2.1. Syntax

Let Π be a countable set of *propositions*, and V be a countable set of typed static and dynamic *variables*. The *terms* e and *formulas* p of the logic are defined as follows:

$$
\begin{array}{lcl}
e & ::= & v \mid \bigcirc e \mid \ominus e \mid \mathsf{beg}(e) \mid \mathsf{end}(e) \mid f(e_1, \ldots, e_m) \\
p & ::= & \pi \mid e_1 = e_2 \mid P(e_1, \ldots, e_m) \mid \neg p \mid p_1 \wedge p_2 \mid \exists v : p \mid \\
 & & \bigcirc p \mid \ominus p \mid (p_1, \ldots, p_m)\ \mathsf{prj}\ p \mid p^+
\end{array}
$$

where v is a static or dynamic variable, and π is a proposition. In $f(e_1, \ldots, e_m)$ and $P(e_1, \ldots, e_m)$, where f is a function and P is a predicate, it is assumed that the types of the terms are compatible with those of the arguments of f and P.

A formula (term) is called a *state* formula (term) if it does not contain any temporal operators, i.e. *next* ($\bigcirc$), *previous* ($\ominus$), *beginning value* (beg), *ending value* (end), *projection* (prj) and *chop-plus* ($^+$); otherwise it is a *temporal* formula (term). Temporal operators *previous* ($\ominus$) and *beginning value* (beg) are called past temporal operators, whereas *next* ($\bigcirc$), *ending value* (end), *projection* (prj) and *chop-plus* ($^+$) are future temporal operators. A formula is called a *non-past* (*non-future*) formula if it does not contain any past (future) temporal operators.

The derived logic connectives, $\vee$, $\rightarrow$ and $\leftrightarrow$, as well as the logic formulas, true and false, are defined as usual.

2.2. Semantics

A *state* $\mathfrak{s}$ is a pair of assignments (I_{var}, I_{prop}) which, for each variable $v \in V$ gives $\mathfrak{s}[v] \stackrel{\text{df}}{=} I_{var}[v]$, and for each proposition $\pi \in \Pi$ gives $\mathfrak{s}[\pi] \stackrel{\text{df}}{=} I_{prop}[\pi]$. Each $I_{var}[v]$ is a value in data domain D with the appropriate type or *nil* (undefined), whereas $I_{prop}[\pi]$ is true or false. In the examples, the assignment I_{var} can be specified as the set of pairs of the form $v : I_{var}[v]$, where $I_{var}[v] \neq nil$. Similarly, I_{prop} can be specified as the set of those π for which $I_{prop}[\pi] = \mathsf{true}$. Sometimes only the relevant elements may be listed.

An *interval* $\sigma \stackrel{\text{df}}{=} \langle \mathfrak{s}_0, \mathfrak{s}_1, \ldots \rangle$ is a non-empty (possibly infinite) sequence of states. It is assumed that each static variable is assigned the same value in all the states in σ. The length of σ, denoted by $|\sigma|$, is defined as ω if σ is infinite; otherwise it is the number of the states in σ minus one.

To have a uniform notation for both finite and infinite intervals, we will use *extended integers* as indices. That is, we consider the set $\mathbb{N}_0$ of non-negative integers with added ω, $\mathbb{N}_\omega \stackrel{\text{df}}{=} \mathbb{N}_0 \cup \{\omega\}$, and extend the standard arithmetic comparison operators ($=$, $<$ and $\leq$) to $\mathbb{N}_\omega$, by setting $\omega = \omega$ and $n < \omega$, for all $n \in \mathbb{N}_0$. Furthermore, we define $\preceq$ as $\leq - \{(\omega, \omega)\}$. To simplify definitions, we will denote σ as $\langle \mathfrak{s}_0, \ldots, \mathfrak{s}_{|\sigma|} \rangle$, where $\mathfrak{s}_{|\sigma|}$ is undefined if σ is infinite. With such a notation, $\sigma_{(i..j)}$ (for $0 \leq i \preceq j \preceq |\sigma|$) denotes the sub-interval $\langle \mathfrak{s}_i, \ldots, \mathfrak{s}_j \rangle$ and $\sigma^{(k)}$ (for $0 \leq k \preceq |\sigma|$) denotes $\langle \mathfrak{s}_k, \ldots, \mathfrak{s}_{|\sigma|} \rangle$.[3] For a variable v, we will denote $\sigma' \stackrel{v}{=} \sigma$ whenever σ' is an interval which is the same as σ except that different values can be assigned to v. That is, $|\sigma| = |\sigma'|$, $I^h_{var}[y] = I'^h_{var}[y]$ for all $y \in V - v$, and $I^h_{prop}[p] = I'^h_{prop}[p]$ for all $p \in \Pi$ ($0 \leq h \preceq |\sigma|$). The concatenation of a finite σ with another interval (or the empty sequence) σ' is denoted by $\sigma \cdot \sigma'$.

[3] When $i > j$, $\sigma_{(i..j)}$ is the empty sequence.

To define the semantics of the projection operator we need an auxiliary operator. Let $\sigma = \langle \mathfrak{s}_0, \mathfrak{s}_1, \ldots \rangle$ be an interval and $r_1, \ldots, r_h$ be integers ($h \geq 1$) such that $0 \leq r_1 \leq r_2 \leq \cdots \leq r_h \leq |\sigma|$. The *projection* of σ onto $r_1, \ldots, r_h$ is the interval (called projected interval)

$$\sigma\downarrow(r_1, \ldots, r_h) \stackrel{\text{df}}{=} \langle \mathfrak{s}_{t_1}, \mathfrak{s}_{t_2} \ldots, \mathfrak{s}_{t_l} \rangle, (t_1 < t_2 < \cdots < t_l)$$

where $t_1, \ldots, t_l$ are obtained from $r_1, \ldots, r_h$ by deleting all duplicates. In other words, $t_1, \ldots, t_l$ is the longest strictly increasing subsequence of $r_1, \ldots, r_h$. For example,

$$\langle \mathfrak{s}_0, \mathfrak{s}_1, \mathfrak{s}_2, \mathfrak{s}_3, \mathfrak{s}_4, \mathfrak{s}_5 \rangle\downarrow(0, 2, 2, 2, 3, 4, 4, 5) = \langle \mathfrak{s}_0, \mathfrak{s}_2, \mathfrak{s}_3, \mathfrak{s}_4, \mathfrak{s}_5 \rangle.$$

An *interpretation* for a PTL term or formula is a tuple $\mathfrak{I} \stackrel{\text{df}}{=} (\sigma, i, k, j)$, where $\sigma = \langle \mathfrak{s}_0, \mathfrak{s}_1, \ldots \rangle$ is an interval and $i, j, k \in \mathbb{N}_\omega$ are such that $i \leq k \preceq j \leq |\sigma|$. Intuitively, we use (σ, i, k, j) to mean that a term or formula is interpreted over a subinterval $\sigma_{(i..j)}$ with the current state being $\mathfrak{s}_k$. Then, for every term e, the evaluation of e relative to $\mathfrak{I}$ is defined as $\mathfrak{I}[e]$, by induction on the structure of the term, as shown in Fig. 1, and the satisfaction relation for formulas, $\models$, is defined as the least relation satisfying the rules in Fig. 2.

It can be shown that $\mathfrak{I} \models p$ if and only if $(\sigma_{(i..j)}, 0, k - i, j - i) \models p$. That is, the relevant part of σ in $\mathfrak{I} = (\sigma, i, k, j)$ is $\sigma_{(i..j)}$. In particular, the valuations of variables and predicates outside the bounds given by i and j do not matter. Furthermore, if p is a formula which does not use the previous operator then $\mathfrak{I} \models p$ if and only if $(\sigma_{(k..j)}, 0, 0, j - k) \models p$.

If there is an interpretation $\mathfrak{I}$ such that $\mathfrak{I} \models p$ then a formula p is *satisfiable*. We also define the satisfaction relation for an interval σ and formula p, by stating that $\sigma \models p$ if $(\sigma, 0, 0, |\sigma|) \models p$. Furthermore, we denote $\models p$ if $\sigma \models p$, for all intervals σ.

Example 2.1. As our first example, consider the formula

$$p \stackrel{\text{df}}{=} (x = 1) \wedge ((\bigcirc \bigcirc x) = ((\bigcirc x) + x))$$

and the interval $\sigma = \langle (I^0_{var}, I^0_{prop}), \ldots \rangle$ such that $I^0_{var} = \{x : 1\}$, $I^1_{var} = \{x : 2\}$ and $I^2_{var} = \{x : 3\}$. One can show that $\mathfrak{I} \models p$, where $\mathfrak{I} \stackrel{\text{df}}{=} (\sigma, 0, 0, 2)$, in the following way:

$$\begin{array}{lll}
\mathfrak{I} \models p & \Longleftrightarrow \mathfrak{I} \models (x = 1) & \text{and} \quad \mathfrak{I} \models (\bigcirc \bigcirc x) = ((\bigcirc x) + x) \\
& \Longleftrightarrow \mathfrak{I}[x] = \mathfrak{I}[1] & \text{and} \quad \mathfrak{I}[\bigcirc \bigcirc x] = \mathfrak{I}[\bigcirc x] + \mathfrak{I}[x] \\
& \Longleftrightarrow I^0_{var}[x] = I^0_{var}[1] & \text{and} \quad (\sigma, 0, 1, 2)[\bigcirc x] = (\sigma, 0, 1, 2)[x] + I^0_{var}[x] \\
& \Longleftrightarrow 1 = 1 & \text{and} \quad (\sigma, 0, 2, 2)[x] = I^1_{var}[x] + 1 \\
& \Longleftrightarrow 1 = 1 & \text{and} \quad I^2_{var}[x] = 2 + 1 \\
& \Longleftrightarrow 1 = 1 & \text{and} \quad 3 = 2 + 1 \\
& \Longleftrightarrow \mathsf{true} &
\end{array}$$

Note that integer 1 is treated as a static variable, so we can write $I^0_{var}[1]$. □

In $(p_1, \ldots, p_m)$ prj q, to ensure smooth synchronization between $p_1, \ldots, p_m$ and q, the previous operator is not allowed within q; however, it can still be used in the p_i's. The projection operator is executable. In programming language terms, the interpretation of $(p_1, \ldots, p_m)$ prj q is somewhat sophisticated as we need *two* sequences of clocks (states) running according to two different time scales: one is a local state sequence, over which $p_1, \ldots, p_m$ are executed, and the other is a global state sequence over which q is executed in parallel with the sequence of processes $p_1, \ldots, p_m$, as follows (see Fig. 3). We start q and p_1 at the first global state and p_1 is executed over a sequence of local states until its termination. Then (the remaining part of) q and p_2 are executed at the second global state, and p_2 is executed over a sequence of local states until its termination, and so on. Although q and p_1 start at the same time, $p_1, \ldots, p_m$ and q may terminate at different time points. If q terminates before some p_{h+1}, then, subsequently, $p_{h+1}, \ldots, p_m$ are executed sequentially. If $p_1, \ldots, p_m$ are finished before q, then the execution of q is continued until its termination.

Two formulas, p and q, are *weakly* (*strongly*) equivalent if $\models p \leftrightarrow q$ (resp. $\models \Box(p \leftrightarrow q)$), and we denote this by $p \approx q$ (resp. $p \equiv q$). Similarly, we denote $p \hookrightarrow q$ (*weak implication*) and $p \supset q$ (*strong implication*), if $\models p \rightarrow q$ and $\models \Box(p \rightarrow q)$, respectively.

IVAR	$\mathcal{I}[v]$	$\stackrel{\text{df}}{=}$	$\mathcal{S}_k[v] = I^k_{var}[v]$	
IFUN	$\mathcal{I}[f(e_1, \ldots, e_m)]$	$\stackrel{\text{df}}{=}$	$\begin{cases} f(\mathcal{I}[e_1], \ldots, \mathcal{I}[e_m]) \\ nil \end{cases}$	if $\mathcal{I}[e_h] \neq nil$ for all $1 \leq h \leq m$ otherwise
IENEXT	$\mathcal{I}[\bigcirc e]$	$\stackrel{\text{df}}{=}$	$\begin{cases} (\sigma, i, k+1, j)[e] \\ nil \end{cases}$	if $k < j$ otherwise
IEPREV	$\mathcal{I}[\ominus e]$	$\stackrel{\text{df}}{=}$	$\begin{cases} (\sigma, i, k-1, j)[e] \\ nil \end{cases}$	if $i < k$ otherwise
IBEG	$\mathcal{I}[\mathsf{beg}(e)]$	$\stackrel{\text{df}}{=}$	$(\sigma, i, i, j)[e]$	
IEND	$\mathcal{I}[\mathsf{end}(e)]$	$\stackrel{\text{df}}{=}$	$\begin{cases} (\sigma, i, j, j)[e] \\ nil \end{cases}$	if $j \neq \omega$ otherwise

Fig. 1. Interpreting PTL terms, where v is a static or dynamic variable, and $e, e_1, \ldots, e_m$ are terms. Note that $\mathcal{I} = (\sigma, i, k, j)$ is an interpretation.

IPROP	$\mathcal{I}$	$\models$	π	if $\mathcal{S}_k[\pi] = I^k_{prop}[\pi] = \mathsf{true}$
IPRED	$\mathcal{I}$	$\models$	$P(e_1, \ldots, e_m)$	if $P(\mathcal{I}[e_1], \ldots, \mathcal{I}[e_m]) = \mathsf{true}$, and $\mathcal{I}[e_h] \neq nil$ for all $1 \leq h \leq m$
IEQUAL	$\mathcal{I}$	$\models$	$e_1 = e_2$	if $\mathcal{I}[e_1] = \mathcal{I}[e_2]$
INEG	$\mathcal{I}$	$\models$	$\neg p$	if $\mathcal{I} \not\models p$
IAND	$\mathcal{I}$	$\models$	$p \wedge q$	if $\mathcal{I} \models p$ and $\mathcal{I} \models q$
INEXT	$\mathcal{I}$	$\models$	$\bigcirc p$	if $k < j$ and $(\sigma, i, k+1, j) \models p$
IPREV	$\mathcal{I}$	$\models$	$\ominus p$	if $i < k$ and $(\sigma, i, k-1, j) \models p$
IEXISTS	$\mathcal{I}$	$\models$	$\exists v : p$	if $(\sigma', i, k, j) \models p$ for some $\sigma' \stackrel{v}{=} \sigma$
IPRJ	$\mathcal{I}$	$\models$	$(p_1, \ldots, p_m)$ prj q	if there are $k = r_0 \leq r_1 \leq \cdots \leq r_m \leq j$ such that $(\sigma, i, r_0, r_1) \models p_1$ and $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p_l$ for all $1 < l \leq m$ and $(\sigma', 0, 0, \lvert\sigma'\rvert) \models q$ for σ' given by : – if $r_m < j$ then $\sigma' = \sigma\downarrow(r_0, \ldots, r_m)\cdot\sigma_{(r_m+1..j)}$ – if $r_m = j$ then $\sigma' = \sigma\downarrow(r_0, \ldots, r_h)$ for some $0 \leq h \leq m$
IPLUS	$\mathcal{I}$	$\models$	p^+	if there are $k = r_0 \leq r_1 \leq \cdots \leq r_{n-1} \preceq r_n = j$ $(n \geq 1)$ such that $(\sigma, i, r_0, r_1) \models p$ and $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p$ for all $1 < l \leq n$

Fig. 2. Interpreting PTL formulas, where v is a variable, $e_1, \ldots, e_m$ are terms, and $p, q, p_1, \ldots, p_m$ are formulas.

Note that weakly equivalent formulas have the same truth value in the first state of every model (i.e., interval), while strongly equivalent formulas have the same truth value in every state of every model. And, similarly, for the weak and the strong implication relations.

It should be clear that the strong equivalence (implication) implies the weak equivalence (resp. implication) but the converse does not hold.

Theorem 2.1. *Let p be a* PTL *formula, and q be a sub-formula of p. If r is a formula such that $q \equiv r$, then $p \equiv p[r/q]$, where $p[r/q]$ is p with q replaced by r.*

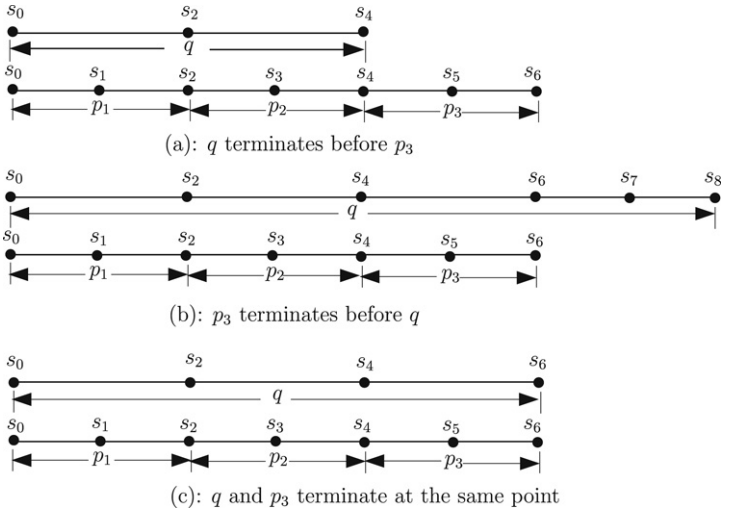

Fig. 3. Possible executions of (p_1, p_2, p_3) prj q.

```
t0      t2              t6                      t12 t13
|-------|---------------|-----------------------|---|
|-----------------------q-------------------------->|
t0  t1  t2  t3  t4  t5  t6  t7  t8  t9  t10 t11 t12
|---|---|---|---|---|---|---|---|---|---|---|---|
|<--p1->|<-----p2------>|<--------- p3--------->|
i=2  4   6   9   12  15  18  22  26  30  34  38  42
j=0      2               8                      26  68
```

Fig. 4. Computing using the projection.

The proof of Theorem 2.1 can be found in the Appendix. This law is the common substitution law and useful in the reduction of programs.

2.3. Derived formulas and logic laws

Table 1 introduces a number of useful derived formulas. Note that: empty represents the final state and first represents the first state of the current interval; len(n) specifies the distance n from the current state to the final state of an interval; *sometimes*($\Diamond p$) means that p holds eventually in the future including the current state; *always* ($\Box p$) means that p holds always in the future from now on; *weak next* ($\odot p$) tells us that either the current state is the final one or p holds at the next state of the present interval; Prj($p_1, \ldots, p_m$) represents a *sequential* computation of $p_1, \ldots, p_m$ since the projected interval is a singleton; and *chop* (p ; q) represents a computation of p followed by q; and *chop-star* (p^*) holds over an interval if it can be partitioned into finitely many subintervals and p holds over each of these.

Example 2.2. As already mentioned, projection can be thought of as a special parallel computation which is executed on two different time scales. Consider the following formulas:

$$
\begin{aligned}
p_1 &\stackrel{\text{df}}{=} \mathsf{len}(2) \wedge \Box(\mathsf{more} \rightarrow ((\bigcirc i) = i + 2)) \\
p_2 &\stackrel{\text{df}}{=} \mathsf{len}(4) \wedge \Box(\mathsf{more} \rightarrow ((\bigcirc i) = i + 3)) \\
p_3 &\stackrel{\text{df}}{=} \mathsf{len}(6) \wedge \Box(\mathsf{more} \rightarrow ((\bigcirc i) = i + 4)) \\
q &\stackrel{\text{df}}{=} \mathsf{len}(4) \wedge (i = 2) \wedge (j = 0) \wedge \Box(\mathsf{more} \rightarrow ((\bigcirc j) = j + i)).
\end{aligned}
$$

Then executing (p_1, p_2, p_3) prj q yields the result shown in Fig. 4. □

Table 1
Derived PTL formulas

AEMPTY	empty	$\stackrel{df}{\equiv}$	$\neg \bigcirc \mathsf{true}$
AMORE	more	$\stackrel{df}{\equiv}$	$\neg\mathsf{empty}$
AFIRST	first	$\stackrel{df}{\equiv}$	$\neg\ominus\mathsf{true}$
APRJ	$\mathsf{Prj}(p_1, \ldots, p_m)$	$\stackrel{df}{\equiv}$	$(p_1, \ldots, p_m)$ prj empty
ACHOP	$p\,;\,q$	$\stackrel{df}{\equiv}$	$\mathsf{Prj}(p, q)$
ASOME	$\Diamond p$	$\stackrel{df}{\equiv}$	$\mathsf{Prj}(\mathsf{true}, p)$
AALWS	$\Box p$	$\stackrel{df}{\equiv}$	$\neg\Diamond\neg p$
AWNEXT	$\odot p$	$\stackrel{df}{\equiv}$	$\mathsf{empty} \vee \bigcirc p$
ALEN	$\mathsf{len}(n)$	$\stackrel{df}{\equiv}$	$\begin{cases}\mathsf{empty} & \text{if } n = 0\\ \bigcirc\mathsf{len}(n-1) & \text{if } n > 1\end{cases}$
ASKIP	skip	$\stackrel{df}{\equiv}$	$\mathsf{len}(1)$
ACHOP-STAR	p^*	$\stackrel{df}{\equiv}$	$\mathsf{empty} \vee p^+$

A PTL formula p is *left end closed* (lec-formula) if $(\sigma, k, k, j) \models p$ if and only if $(\sigma, i, k, j) \models p$, for any interpretation (σ, i, k, j). Similarly, q is *right end closed* (rec-formula) if $(\sigma, i, k, k) \models q$ if and only if $(\sigma, i, k, j) \models q$ for any interpretation (σ, i, k, j). Intuitively, being lec-formula means that if p holds over a subinterval $\sigma_{(k..j)}$ resulting from $\sigma_{(i..j)}$ by chopping it at the state $\mathfrak{s}_k$, then p does not refer to any state to the left of $\mathfrak{s}_k$, and similarly for a rec-formula. For instance, $\Box(\mathsf{more} \rightarrow \bigcirc(p_1 \leftrightarrow \ominus p_2))$ is a lec-formula, and first is a *rec*-formula.

A formula p is *non-local* (or *terminable*) if $\sigma \models p$ implies $|\sigma| \geq 1$ (resp. $|\sigma| = 0$).

Theorem 2.2. *The laws in Tables 2 and 3 all hold.*

The proof of Theorem 2.2 can be found in the Appendix. These logic laws are basic laws and useful for the reduction of programs. In particular, they play an important role in transforming a program into its normal form.

To reduce programs with existential quantification, we will use renaming. Given a formula $\exists x : p(x)$ with a bound variable x, we can remove the existential quantification to obtain a formula $p(y)$ with a free variable y by renaming x as y. To do so, we require that: (i) y does not occur free or bound in $\exists x : p(x)$; (ii) y and x are both either dynamic or static; and (iii) y is substituted only for those occurrences of x which are bound by the outer $\exists x$. We will call $p(y)$ a *renamed formula* of $\exists x : p(x)$.

Proposition 2.3. *Let $p(y)$ be a renamed formula of $\exists x : p(x)$. Then $\exists x : p(x)$ is satisfiable iff $p(y)$ is satisfiable. Moreover, any model of $p(y)$ is also a model of $\exists x : p(x)$.*

Note that projection as defined here and chop-plus are related operators, with p^+ being equivalent to

$$\exists m : m \geq 1 \wedge \mathsf{Prj}(\underbrace{p, \ldots, p}_{m \text{ times}}).$$

Note that the chop operator is central to ITL [23] as well as chop-star and projection [23,24]. Moreover, [3–5] further develop the projection construct of [23].

In what follows, in order to avoid excessive use of parentheses, we will sometimes use the following priority levels of PTL (and Framed Tempura) operators:

LEVEL 1 $\neg$	LEVEL 2 $\bigcirc\ \odot\ \ominus\ \Diamond\ \Box$	LEVEL 3 $\exists$	LEVEL 4 $:=\ =$
LEVEL 5 $\wedge\ \vee\ \|$	LEVEL 6 $\rightarrow\ \leftrightarrow$	LEVEL 7 ; prj	

Table 2
PTL laws I

Law1	$\mathsf{empty};\ p$	$\equiv$	p	if $lec(p)$
Law2	$(p \wedge \mathsf{empty});\ q$	$\equiv$	$p \wedge q$	if $rec(p)$ and $lec(q)$
Law3	$p \wedge \mathsf{empty};\ q$	$\supset$	q	if $lec(q)$
Law4	$p;\ \mathsf{empty}$	$\equiv$	$p \wedge \Diamond\mathsf{empty}$	
Law5	$p;\ q \wedge \mathsf{empty}$	$\equiv$	$p \wedge \Box(\mathsf{empty} \rightarrow q)$	if $lec(q)$
Law6	$\neg \odot p$	$\equiv$	$\bigcirc\neg p$	
Law7	$\neg \bigcirc p$	$\equiv$	$\odot\neg p$	
Law8	$\bigcirc(p \wedge q)$	$\equiv$	$\bigcirc p \wedge \bigcirc q$	
Law9	$\bigcirc(p \vee q)$	$\equiv$	$\bigcirc p \vee \bigcirc q$	
Law10	$\bigcirc(p \rightarrow q)$	$\equiv$	$\odot p \rightarrow \bigcirc q$	
Law11	$r;\ (p \vee q)$	$\equiv$	$(r;\ p) \vee (r;\ q)$	
Law12	$(p \vee q;\ r)$	$\equiv$	$(p;\ r) \vee (q;\ r)$	
Law13	$\bigcirc p$	$\equiv$	$\odot p \wedge \mathsf{more}$	
Law14	$\bigcirc(\exists x : p)$	$\equiv$	$\exists x : \bigcirc p$	
Law15	$(\bigcirc p);\ q$	$\equiv$	$\bigcirc(p;\ q)$	
Law16	$w \wedge (p;\ q)$	$\equiv$	$(w \wedge p);\ q$	
Law17	more	$\supset$	$\bigcirc e_1 = \bigcirc e_2 \leftrightarrow \bigcirc(e_1 = e_2)$	
Law18	$\neg\mathsf{first} \wedge \mathsf{more}$	$\supset$	$\bigcirc\ominus e = \ominus \bigcirc e$	
Law19	$(p \wedge \mathsf{empty});\ \mathsf{empty}$	$\equiv$	$p \wedge \mathsf{empty}$	
Law20	p^+	$\equiv$	$p \vee (p;\ p^+)$	
Law21	p^+	$\equiv$	$p \vee ((p \wedge \mathsf{more});\ p^+)$	if $lec(p)$
Law22	p	$\equiv$	$(p;\ \mathsf{empty}) \vee (p \wedge \Box\mathsf{more})$	
Law23	$\Diamond p$	$\equiv$	$p \vee \bigcirc\Diamond p$	
Law24	$\Box p$	$\equiv$	$p \wedge \odot\Box p$	
Law25	$\bigcirc p$	$\supset$	more	
Law26	p^*	$\equiv$	$\mathsf{empty} \vee (p;\ p^*) \vee (p \wedge \Box\mathsf{more})$	
Law27	p^*	$\equiv$	$\mathsf{empty} \vee (p \wedge \mathsf{more};\ p^*) \vee (p \wedge \Box\mathsf{more})$	if $lec(p)$
Law28	$\neg\mathsf{first} \wedge \mathsf{more}$	$\supset$	$(\bigcirc\ominus p) \leftrightarrow (\ominus \bigcirc p)$	
Law29	$\bigcirc e_1 + \bigcirc e_2$	$=$	$\bigcirc(e_1 + e_2)$	
Law30	$(p \wedge \mathsf{empty});\ (q \wedge \mathsf{empty})$	$\equiv$	$p \wedge q \wedge \mathsf{empty}$	if $rec(p)$ and $lec(q)$
Law31	$p^+ \wedge \mathsf{empty}$	$\equiv$	$p \wedge \mathsf{empty}$	
Law32	$\exists x : p(x)$	$\equiv$	$\exists y : p(y)$	
Law33	$\forall x : p(x)$	$\equiv$	$\forall y : p(y)$	
Law34	$\exists x : p(x)$	$\equiv$	$\exists x : q(x)$	if $p(x) \equiv q(x)$
Law35	$\forall x : p(x)$	$\equiv$	$\forall x : q(x)$	if $p(x) \equiv q(x)$
Law36	$\bigcirc p\ \mathsf{prj}\ \bigcirc q$	$\equiv$	$\bigcirc(p;\ q)$	
Law37	$\mathsf{empty}\ \mathsf{prj}\ q$	$\equiv$	q	
Law38	$q\ \mathsf{prj}\ \mathsf{empty}$	$\equiv$	q	

It is assumed that in Law32–Law35 variables x and y are both either static or dynamic, and neither does x appear in $p(y)$ nor does y appear in $p(x)$. Note that $lec(p)$ means that p is a lec-formula, and $rec(p)$ means that p is a rec-formula and w is a state formula.

3. An executable TLP language

The programming language we use – a subset of PTL – extends Tempura with framing, projection, and await operators [6–10]. In addition, program variables can refer to their previous values. After introducing the basic constructs of Tempura and the previous operator, we will formalize the frame operator and the await construct.

3.1. Syntax

Generally speaking, Tempura programs are deterministic and so disjunction is usually unavailable. Hence, the negation – being fundamentally non-deterministic – is not a primitive operator of the language. Instead, a conditional statement and empty, both defined in terms of negation, are taken as primitives. Programs are constructed from the operators described below, together with a suitable choice of expressions which may involve the previous operator.

Table 3
PTL laws II

LAW39	$\mathbf{p}$ prj empty	$\equiv$	$p_1 ; \ldots ; p_m$
LAW40	$(\mathbf{p}, \text{empty}, \mathbf{q})$ prj q	$\equiv$	$(\mathbf{p}, \mathbf{q})$ prj q
LAW41	$(p \wedge \text{empty}) ; (\mathbf{p}$ prj $q)$	$\supset$	$(p, \mathbf{p})$ prj q
LAW42	$(\mathbf{p}, \text{empty} \wedge w, p, \mathbf{q})$ prj q	$\equiv$	$(\mathbf{p}, w \wedge p, \mathbf{q})$ prj q
LAW43	$(\mathbf{p}, p \vee w, \mathbf{q})$ prj q	$\equiv$	$((\mathbf{p}, p, \mathbf{q})$ prj $q) \vee ((\mathbf{p}, w, \mathbf{q})$ prj $q)$
LAW44	$\mathbf{p}$ prj $(p \vee q)$	$\equiv$	$(\mathbf{p}$ prj $p) \vee (\mathbf{p}$ prj $q)$
LAW45	p prj $\bigcirc q$	$\equiv$	$((p \wedge \text{more}) ; q) \vee ((p \wedge \text{empty}) ; \bigcirc q)$
LAW46	$(p \wedge \text{more}) ; (\mathbf{p}$ prj $q)$	$\supset$	$(p, \mathbf{p})$ prj $\bigcirc q$
LAW47	$(p, \mathbf{p})$ prj $\bigcirc q$	$\equiv$	$((p \wedge \text{more}) ; (\mathbf{p})$ prj $q)$ $\vee ((p \wedge \text{empty}) ; (\mathbf{p})$ prj $\bigcirc q)$
LAW48	$(\bigcirc p_1, \ldots, p_m)$ prj $\bigcirc q$	$\equiv$	$\bigcirc(p_1 ; (p_2, \ldots, p_m)$ prj $q)$
LAW49	p prj q	$\equiv$	$p \wedge q$
LAW50	$(w \wedge p, \mathbf{p})$ prj q	$\equiv$	$w \wedge ((p, \mathbf{p})$ prj $q)$
LAW51	$\mathbf{p}$ prj $(w \wedge q)$	$\equiv$	$w \wedge (\mathbf{p}$ prj $q)$
LAW52	$\text{fin}(w) \wedge (p ; q)$	$\equiv$	$p ; (\text{fin}(w) \wedge q)$
LAW53	skip prj q	$\equiv$	q
LAW54	q prj skip	$\supset$	q
LAW55	(p, q) prj skip	$\supset$	$p ; q$
LAW56	Assuming $p_0 \equiv p_{m+1} \equiv \text{empty}$:		
	$\mathbf{p}$ prj $\bigcirc q \equiv \bigvee_{t=0}^{m-1} ((p_0 \wedge \cdots \wedge p_t) \wedge \text{empty} ; p_{t+1} \wedge \text{more} ; (p_{t+2}, \ldots, p_{m+1})$ prj $q)$ $\vee ((p_0 \wedge \cdots \wedge p_m) ; \bigcirc q)$		

Note that $\mathbf{p}$ and $\mathbf{q}$ stand for (possibly empty) sequences of formulas: $p_1, \ldots, p_m$ and $q_1, \ldots, q_n$. It is assumed that: (i) in LAW42, LAW50, LAW51 and LAW52 w is a state formula; (ii) in LAW53 and LAW54 q is nonlocal; (iii) in LAW55 p or q is a nonlocal formula; and (iv) in LAW49 p and q are state formulas.

Altogether, there are eleven elementary statements. Five of them are basic constructs taken directly from PTL: the equality ($=$), conjunction ($\wedge$), existential quantification ($\exists x : p$), next ($\bigcirc$) and projection (prj). The remaining statements are: the conditional, always[4] ($\Box$), chop (;), while, parallel and empty. From the point of view of the programming language, the eleven statements are all primitives since the negation ($\neg$), sometimes ($\Diamond$), disjunction ($\vee$) and chop-plus ($+$) are absent in the definition of the extended Tempura. This also implies that some operators cannot be derived as before. Note that in an induction proof of a property of programs, the assignment and empty statements can be thought of as basic statements and the others can be treated as composite statements. In what follows, x is a variable, e is an arbitrary arithmetic expression, b is a state boolean expression (consisting of propositions, variables and boolean connectives), and p and q are programs.

ASSIGNMENT (UNIFICATION)	$x = e$
CONJUNCTION	$p \wedge q$
CONDITIONAL	`if` b `then` p `else` $q \stackrel{\text{df}}{=} (b \rightarrow p) \wedge (\neg b \rightarrow q)$
EXISTS	$\exists x : p$
NEXT	$\bigcirc p$
ALWAYS	$\Box p$
SEQUENCE	$p ; q$
WHILE LOOP	`while` b `do` $p \stackrel{\text{df}}{=} (p \wedge b)^* \wedge \Box(\text{empty} \rightarrow \neg b)$
PROJECTION	$(p_1, \ldots, p_m)$ prj q
PARALLEL	$p \parallel q \stackrel{\text{df}}{=} p \wedge (q ; \text{true}) \vee q \wedge (p ; \text{true})$
TERMINATION	empty

[4] Note that $\Diamond$ is not permitted.

Table 4
While laws

LAW57	`while` b `do` p	$\equiv$	`if` b `then` $(p$ `;` `while` b `do` $p)$ `else` empty
LAW58	`while` b `do` p	$\equiv$	`if` b `then` $(p \wedge$ more `;` `while` b `do` $p)$ `else` empty
LAW59	`while` b `do` p	$\equiv$	$((\neg b \wedge$ empty$) \vee (b \wedge p \wedge$ more `;` `while` b `do` $p)) \vee b \wedge p \wedge \Box$more
LAW60	`while` b `do` p	$\equiv$	$((\neg b \wedge$ empty$) \vee (b \wedge p$ `;` `while` b `do` $p)) \vee b \wedge p \wedge \Box$more

It is assumed that in LAW58–LAW59 p is a lec-formula, and in LAW57–LAW58 p is terminable.

The assignment $(x = e)$ means that the value of variable x is equal to the value of the expression e, and its interpretation is subject to IEQUAL. Whenever such an assignment is encountered, we evaluate x and e by $\mathfrak{I}[x]$ and $\mathfrak{I}[e]$ to see whether $\mathfrak{I}[x] = \mathfrak{I}[e]$. Therefore, if e is evaluated to a constant in D and x has not been specified before (or it was specified to have the same value as e), then we say e is assigned to x. In this case, the equality $(x = e)$ is satisfied otherwise it is false. In other words, x is unified with e as, e.g., in Prolog.

Theorem 3.1. *The laws in Table 4 all hold.*

The proof of Theorem 3.1 can be found in the Appendix. These logic laws are concerned with *while* statement and useful for reducing programs with *while* construct.

Note that the equality in Tempura has two functions: assignment and comparison. The former is a statement in a program while the latter is in a condition (boolean expression) associated with conditional or iterative statements. An assignment is true as long as it is satisfiable whereas a condition is true if all the variables involved in it are specified and the condition is evaluated to true.

As in the conventional programming languages, the conditional statement `if` b `then` p `else` q means that if b is evaluated to true then the process (i.e. sub-program) p is executed; otherwise, q is executed. The 'next' statement $\bigcirc p$ means that p holds at the next state, while $\Box p$ means that p holds in all the states from now on. The terminal statement empty simply means that the current state is the final state of the interval over which a program is executed. The sequential statement p `;` q means that p holds from now until some point in the future and from that time point q holds. Intuitively, p is a program which is executed from the current state until its termination, and then q is executed. The conjunction $p \wedge q$ is executed in a parallel manner. The processes p and q start at the same state but may terminate at different states. They share all the states and variables during the simultaneous execution. The iteration `while` b `do` p allows process p to be repeatedly executed a finite (or infinite) number of times over a finite (resp. infinite) interval as long as the condition b holds at the beginning of each execution. If b becomes false, then the `while` statement terminates; otherwise, p is executed. For instance, over an infinite interval, the statement `while` true `do` p allows p to be executed a finite or infinite number of times, each time on a finite subinterval (or on an infinite subinterval for the last execution). This statement is obviously false within any finite interval if the execution of p requires a non-singleton interval. Another special case is `while` b `do` empty which is simply equivalent to $\neg b \wedge$ empty.

The projection $(p_1, \ldots, p_m)$ prj q means that q is executed in parallel with p_1 `;` $\ldots$ `;` p_m over an interval obtained by taking the endpoints of the intervals over which the p_i's are executed. The construct permits the processes $p_1, \ldots p_m, q$ to be autonomous, each process having the right to specify the interval over which it is executed. In particular, the sequence of p_i's and q may terminate at different time points. The parallel computation presented here proceeds synchronously, and may be modelled by true concurrency. It is weaker than the asynchronous parallel computation modelled by interleaving, and is close to conjunction. The basic difference between $p \parallel q$ and $p \wedge q$ is that the former allows both processes p and q to specify their own intervals while the latter does not. For instance, len(2) $\parallel$ len(3) is satisfiable but len(2) $\wedge$ len(3) is not. The existential quantification $(\exists x : p)$ intends to hide the variable x within the process p. It may permit p to use a local variable x, and this idea can be realized in an operational semantics. However, within the temporal semantics, the concept of a local variable is not effective.

3.2. Derived statements

The first group of derived statements are various assignment operators. Below, x is a variable, u static variable, e expression (term), and *op* any of the binary assignment operators.

NEXT	$x \circ\!= e$	$\stackrel{\text{df}}{=}$	$\bigcirc x = e$
UNIT	$x := e$	$\stackrel{\text{df}}{=}$	$\mathsf{skip} \wedge (x \circ\!= e)$
MULTIPLE	$(x_1, \ldots, x_n)\ op\ (e_1, \ldots, e_n)$	$\stackrel{\text{df}}{=}$	$(x_1\ op\ e_1) \wedge \cdots \wedge (x_n\ op\ e_n)$

The 'next' assignment specifies the value of a variable at the next state, while the unit assignment additionally specifies the length of the interval over which the assignment takes place to be 1.

The next group of derived operators are concerned with termination and the final state.

FINAL	$\mathsf{fin}(p)$	$\stackrel{\text{df}}{=}$	$\Box(\mathsf{empty} \rightarrow p)$
KEEP	$\mathsf{keep}(p)$	$\stackrel{\text{df}}{=}$	$\Box(\neg\mathsf{empty} \rightarrow p)$
HALT	$\mathsf{halt}(p)$	$\stackrel{\text{df}}{=}$	$\Box(\mathsf{empty} \leftrightarrow p)$

Note that: $\mathsf{fin}(p)$ holds over an interval as long as p holds at the final state; $\mathsf{keep}(p)$ holds over an interval if p holds at every non-final state; and $\mathsf{halt}(p)$ holds over an interval if and only if p holds at the final state.

Although the language does not include disjunction, negation and universal quantification as basic statements, it is not guaranteed that programs are deterministic. The problem is that the immediate assignment $(x = e)$ is non-deterministic since

$$(x = e) \equiv (((x = e) \wedge \mathsf{empty}) \vee ((x = e) \wedge \mathsf{more}))$$

and so are, for instance, $\bigcirc(x = e)$ and $\Box(x = e)$. If the unit assignment $(x := e)$ rather than an immediate assignment was used as a primitive statement, the language would become deterministic. However, an immediate assignment has its advantages, e.g., it can easily be used to initialize variables and it corresponds to the equality taken directly from the underlying logic. To write a deterministic program, constructs like $\mathsf{len}(k)$ or $\mathsf{halt}(b)$ or $\Box\mathsf{more}$ (for non-terminating program) are needed in order to specify a definite interval.

The third group of derived statements are iterative operators (below b is a state boolean expression).

FOR LOOP	`for` 0 `times do` p	$\stackrel{\text{df}}{=}$	empty
	`for` $n+1$ `times do` p	$\stackrel{\text{df}}{=}$	(`for` n `times do` p)`;` p
REPEAT LOOP	`repeat` p `until` b	$\stackrel{\text{df}}{=}$	p`; while` $\neg b$ `do` p

In what follows, we will use the term 'program' to mean any program belonging to the extended Tempura. Typically, it will be a deterministic and terminating program which involves the previous operator only in expressions. However, some results are discussed in a broader scope in which non-deterministic programs and/or infinite intervals are considered. To avoid ambiguity, whenever only a deterministic program or only a finite interval is involved, we state this explicitly.

3.3. Expressions and data structures

The choice of permissible expressions is wider but only constants, variables, arrays, strings and list expressions, as well as restricted temporal expressions are mentioned in this paper. There are integer and boolean constants, and variables are divided into simple variables (x) and structured variables ($x[1]$). One may refer to the value of a variable at the previous and next state of an interval. Arithmetic expressions (e) and boolean expressions (b) are defined as follows (below n is an integer variable):

$$
\begin{array}{lcl}
e & ::= & n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1/e_2 \mid e_1 \, mod \, e_2 \mid \bigcirc e \mid \ominus e \\
b & ::= & \mathsf{true} \mid \mathsf{false} \mid e_1 > e_2 \mid e_1 \geq e_2 \mid e_1 < e_2 \mid e_1 \leq e_2 \mid e_1 = e_2 \mid e_1 \neq e_2 \mid \\
 & & \neg b_1 \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid b_1 \rightarrow b_2 \mid b_1 \leftrightarrow b_2
\end{array}
$$

Note that $\bigcirc e$ and $\ominus e$ are only meaningful in a state different from the final and first one, respectively.

A list is a finite or infinite sequence $l = \langle a_0, a_1, \ldots \rangle$ of elements separated by commas and enclosed within the angle brackets. Its length $|l|$ is the number of the elements in l minus 1; the i-th element is denoted by $l[i]$, and the sublist from the i-th element to the j-th element by $l(i..j)$. The empty list is denoted by ϵ. To manipulate lists, we use the following operators: concatenation ($\cdot$); fusion ($\circ$); head (hd) returning the first element of l; last (lt) returning the last element of l; and tail (tl) returning the sublist $l(1..|l|)$. The first two operators are defined thus:

$$
l_1 \cdot l_2 \stackrel{\text{df}}{=} \begin{cases} l_1 & \text{if } |l_1| = \omega \text{ or } l_2 = \epsilon \\ l_2 & \text{if } l_1 = \epsilon \\ \langle a_0, \ldots, a_i, a_{i+1}, \ldots \rangle & \text{if } l_1 = \langle a_0, \ldots, a_i \rangle \text{ and } l_2 = \langle a_{i+1}, a_{i+2} \ldots \rangle \end{cases}
$$

$$
l_1 \circ l_2 \stackrel{\text{df}}{=} \begin{cases} l_1 & \text{if } |l_1| = \omega \text{ or } l_2 = \epsilon \\ l_2 & \text{if } l_1 = \epsilon \\ \langle a_0, \ldots, a_i, a_{i+1}, \ldots \rangle & \text{if } l_1 = \langle a_0, \ldots, a_i \rangle \text{ and } l_2 = \langle a_i, a_{i+1} \ldots \rangle \\ \textit{undefined} & \text{otherwise} \end{cases}
$$

Text strings are surrounded by the double quotation marks. They may be indexed and manipulated in the same way as lists. Finally, arrays are sets of indexed elements of the same type. A two-dimensional array can be expressed as a list with the row-first implementation.

3.4. Framing

As mentioned in the introduction, framing is concerned with the persistence of the values of variables from one state to another. There are at least two realistic ways to go about framing. For example, [17] defines assignment as

$$(x := e) \stackrel{\text{df}}{=} (x' = e) \wedge (y_1' = y_1) \wedge \cdots \wedge (y_m' = y_m)$$

where the apostrophes represent the new values of variables, and $y_1, \ldots, y_m$ are all the remaining variables of a program. Intuitively, whenever a variable is assigned a value, all the other variables remain stable. However, this method can only manage framing in a limited case in which all variables are framed, and the conjunction of assignments is forbidden since $(x := e) \wedge (y := f)$ is false whenever e (or f) does not evaluate to the current value of x (resp. y). Since in Tempura parallel composition is based on conjunction, adopting the above strategy for framing would rule out parallel assignments.

Another way to introduce framing is through an explicit operator, enabling one to establish a flexible framed environment in which framed and non-framed variables can be mixed, with frame operators being used in sequential, conjunctive and parallel manner. The key characteristic of a frame operator could be stated thus:

$\mathsf{frame}(x)$ *means that variable x keeps its old value over an interval if no assignment to x has been encountered.*

The crux is how to perceive the assignment of a value to x, and we will use for this purpose a dedicated *assignment flag* $\mathsf{af}(x)$ which cannot be used freely in a program but only in the context of assignment and frame operator. This predicate is true whenever an assignment of a value to x is encountered, and false otherwise. The assignment flag is easy to understand but rather difficult to formalize in a logic framework. The problem is that a program provides only positive information about explicit assignments whereas what we need is a negative information about those variables which are not assigned values at the current state. One might search for the positive information syntactically and obtain the negative information by complement, but doing this in a logic framework is hard.

Let $SV \subset V$ be the set of variables which can be subjected to the framing technique. Since the assignment operators defined so far cannot manage framing in a desired way, we introduce new operators. First, for each variable $x \in SV$ we introduce a special proposition $aflag_x$, and then define the new assignments, in the following way (below $e, e_1, \ldots, e_n \neq nil$, and op is any of the three new assignments):

POSITIVE IMMEDIATE	$x \Leftarrow e$	$\stackrel{\text{df}}{=}$	$(x = e) \wedge aflag_x$
NEXT	$x \circ\!\!=^{+} e$	$\stackrel{\text{df}}{=}$	$(\bigcirc x = e) \wedge \bigcirc aflag_x$
UNIT	$x :=^{+} e$	$\stackrel{\text{df}}{=}$	$(x \circ\!\!=^{+} e) \wedge \mathsf{skip}$
MULTIPLE	$(x_1, \ldots x_n)\ op\ (e_1, \ldots, e_n)$	$\stackrel{\text{df}}{=}$	$(x_1\ op\ e_1) \wedge \cdots \wedge (x_n\ op\ e_n)$

The meaning of these assignments is similar to those defined in Section 3.2.

The definition of the assignment flag is simple: $\mathsf{af}(x) \stackrel{\text{df}}{=} aflag_x$, for every variable $x \in SV$ (recall that $aflag_x$ cannot be used for any other purpose). The predicate $\mathsf{af}(x)$ is associated with some assignment operator and can be used to assert whether or not such an assignment has taken place to x in the execution of a program. Whenever such an assignment is encountered, $\mathsf{af}(x)$ should be true. Conversely, when $\mathsf{af}(x)$ is true, such an assignment should have been perceived in the execution of the program. As expected, when $x \Leftarrow e$ is encountered, $aflag_x$ is set to true, hence $\mathsf{af}(x)$ is true.

Finally, we can define our *looking back* framing operators:

$\mathsf{lbf}(x)$	$\stackrel{\text{df}}{=}$	$\neg\mathsf{af}(x) \rightarrow \exists b : (\ominus x = b \wedge x = b)$
$\mathsf{frame}(x)$	$\stackrel{\text{df}}{=}$	$\Box(\mathsf{more} \rightarrow \bigcirc\mathsf{lbf}(x))$
$\mathsf{frame}(x_1, \ldots, x_n)$	$\stackrel{\text{df}}{=}$	$\mathsf{frame}(x_1) \wedge \cdots \wedge \mathsf{frame}(x_n)$

4. Semantics of framed programs

We will interpret a framed program P using minimal (canonical) models. To this end, we assume that P contains a finite set V_P of variables and a finite set Π_P of propositions, respectively interpreted over

$$\mathbf{D} \stackrel{\text{df}}{=} D \cup \{nil\} \quad \text{and} \quad \mathbf{B} \stackrel{\text{df}}{=} \{\mathsf{true}, \mathsf{false}\}.$$

There are three main ways of interpreting propositions contained in P, namely the canonical, complete and partial, as in the semantics of logic programming languages [2]. Here we will use the canonical one. That is, each I^k_{var} in a model $\sigma = \langle (I^0_{var}, I^0_{prop}), (I^1_{var}, I^1_{prop}), \ldots \rangle$ is used as in the underlying logic, but I^k_{prop} is changed to the canonical interpretation.

A *canonical interpretation* on propositions is a set $I_{prop} \subseteq \Pi_P$ and, implicitly, all propositions not in I_{prop} are false. Note that in the logic framework I^k_{prop} is an assignment of a truth value in $\mathbf{B}$ to each proposition $p \in \Pi$ at state $\mathfrak{s}_k$; whereas in a canonical interpretation, I^k_{prop} is a set of propositions of Π_P which are true at $\mathfrak{s}_k$. Clearly, these two definitions are equivalent except that they refer to different sets of variables and propositions.

Let $\sigma = \langle (I^0_{var}, I^0_{prop}), (I^1_{var}, I^1_{prop}), \ldots \rangle$ be a model. We denote the sequence of interpretation on propositions of σ by $\sigma_{prop} = \langle I^0_{prop}, I^1_{prop}, \ldots \rangle$, and call σ_{prop} canonical if each I^i_{prop} is a canonical interpretation on propositions. Moreover, if $\sigma' = \langle (I'^0_{var}, I'^0_{prop}), (I'^1_{var}, I'^1_{prop}), \ldots \rangle$ is another canonical model, then we denote:

- $\sigma_{prop} \sqsubseteq \sigma'_{prop}$ if $|\sigma| = |\sigma'|$ and $I^i_{prop} \subseteq I'^i_{prop}$, for all $0 \leq i \leq |\sigma|$.
- $\sigma \sqsubseteq \sigma'$ if $\sigma_{prop} \sqsubseteq \sigma'_{prop}$.
- $\sigma \doteq \sigma'$ if $\sigma \sqsubseteq \sigma'$ and $\sigma' \sqsubseteq \sigma$.
- $\sigma \sqsubset \sigma'$ if $\sigma \sqsubseteq \sigma'$ and $\sigma' \not\sqsubseteq \sigma$.

For example, $\langle(\{x{:}1\}, \varnothing)\rangle \doteq \langle(\{x{:}2\}, \varnothing)\rangle$ and $\langle(\{x{:}1\}, \varnothing)\rangle \sqsubset \langle(\varnothing, \{aflag_x\})\rangle$.

If there exists a model σ with σ_{prop} being canonical and $\sigma \models P$ as in the logic, then the program P is *satisfiable* under the canonical interpretation on propositions. We denote this by $\sigma \models_{\mathbf{c}} P$, and call σ_{prop} a canonical interpretation sequence (on propositions) of P. If $\sigma \models P$, for all σ with canonical σ_{prop}, then P is *valid* under the canonical interpretation on propositions. We denote this by $\models_{\mathbf{c}} P$. Note that the definition of a canonical interpretation of a program is independent of its syntax in the sense that it does not refer to the program's structure. Hence it can also be applied to temporal formulas.

Since a program can be satisfied by several different canonical models, one needs to carefully choose a model which reflects its intended meaning. We now formulate a central definition of this paper.

Definition 4.1. Let P be a program, and $\mathfrak{I} = (\sigma, i, k, j)$ be a canonical interpretation. Then $\mathfrak{I}$ is a *minimal interpretation* of P if $\mathfrak{I} \models_{\mathbf{c}} P$, and there is no σ' such that $\sigma' \sqsubset \sigma$ and $(\sigma', i, k, j) \models_{\mathbf{c}} p$. We denote this by $\mathfrak{I} \models_{\mathbf{m}} p$.

A *minimal model* of a program P is any canonical model σ such that $(\sigma, 0, 0, |\sigma|) \models_{\mathbf{m}} p$. We denote this by $\sigma \models_{\mathbf{m}} p$. Moreover, the equivalence relations $\equiv_{\mathbf{m}}$ and $\approx_{\mathbf{m}}$ as well as the strong implication relation $\supset_{\mathbf{m}}$ can be defined similarly as the relations $\equiv$, $\approx$ and $\supset$.

Example 4.1. Consider the program

PROG VII : $\mathsf{frame}(x) \wedge (x = 1) \wedge \mathsf{len}(1)$.

One can see that it has the following four kinds of canonical models:

$\sigma_1 = \langle(\{x{:}1\}, \{aflag_x\}), (\varnothing, \{aflag_x\})\rangle$ $\quad$ $\sigma_2 = \langle(\{x{:}1\}, \{aflag_x\}), (\{x{:}1\}, \varnothing)\rangle$
$\sigma_3 = \langle(\{x{:}1\}, \varnothing), (\varnothing, \{aflag_x\})\rangle$ $\quad$ $\sigma_4 = \langle(\{x{:}1\}, \varnothing), (\{x{:}1\}, \varnothing)\rangle$.

The intended meaning of PROG VII is captured by its minimal model, in this case σ_4. For this model, x is defined in both states of the interval and its value is equal to 1.

Using framing, program PROG I from the introduction can be amended as follows:

PROG VIII : $\mathsf{frame}(x) \wedge (x = 1) \wedge ((y := 2)\,;\ (y := x + y))$.

Note that PROG VIII has the same meaning as program PROG II, but it is more concise and has simpler implementation.

Theorem 4.1. *The laws in Table 5 all hold.*

The proof of Theorem 4.1 can be found in the Appendix. These logic laws are basically algebraic properties of framing operators including equivalent, distributive, absorptive and idempotent laws. Note that the properties of the framing operator in Table 5 are useful, in particular, in the reduction of framed programs. Another useful result is that the strong equivalence between programs also holds under the minimal model semantics.

Theorem 4.2. *Let p and q be framed programs. We have,*

1. $p \equiv q$ *implies* $p \equiv_{\mathbf{m}} q$
2. $\mathfrak{I} \models_{\mathbf{m}} p \vee q$ *implies* $\mathfrak{I} \models_{\mathbf{m}} p$ *or* $\mathfrak{I} \models_{\mathbf{m}} q$.

Proof. (1) Suppose that $\mathfrak{I} = (\sigma, 0, r, |\sigma|) \models_{\mathbf{m}} p$. Then $\mathfrak{I} \models p$. Since $p \equiv q$, we have $\mathfrak{I} \models q$. If $\mathfrak{I} \not\models_{\mathbf{m}} q$, then there is $\mathfrak{I}' = (\sigma', 0, r, |\sigma'|)$ such that $\mathfrak{I}' \models q$ and $\sigma' \sqsubset \sigma$. By $p \equiv q$, we have $\mathfrak{I}' \models p$ and $\sigma' \sqsubset \sigma$.

(2) Let $\mathfrak{I} = (\sigma, 0, k, |\sigma|)$. We have $\mathfrak{I} \models p$ or $\mathfrak{I} \models q$, so suppose that the former holds. If $\mathfrak{I} \not\models_{\mathbf{m}} p$, then there exists $\mathfrak{I}' = (\sigma', 0, k, |\sigma'|)$ such that $\mathfrak{I}' \models p$ and $\sigma' \sqsubset \sigma$, leading to $\mathfrak{I}' \models p \vee q$ and $\sigma' \sqsubset \sigma$. This contradicts $\mathfrak{I} \models_{\mathbf{m}} p \vee q$. □

In [6] it was shown that if P is a (possibly non-deterministic or non-terminable) satisfiable framed program which has a finite model, or has finitely many models, then it also has at least one minimal model. However, for a program which has infinitely many infinite models and no finite model, the construction used in [6] does not work. In this paper we extend the result of [6] by showing that

Table 5
Framing laws

LAW61	$\mathsf{frame}(x)$	$\equiv$	$\mathsf{frame}(x) \parallel \mathsf{frame}(x)$
		$\equiv$	$\mathsf{frame}(x)\,;\,\mathsf{frame}(x)$
		$\equiv$	$\mathsf{frame}(x) \wedge \mathsf{frame}(x)$
		$\equiv$	$\Box(\neg\mathsf{first} \rightarrow \mathsf{lbf}(x))$
LAW62	$\mathsf{frame}(x) \wedge \mathsf{more}$	$\equiv$	$\bigcirc(\mathsf{lbf}(x) \wedge \mathsf{frame}(x))$
LAW63	$\mathsf{frame}(x) \wedge \mathsf{empty}$	$\equiv$	empty
LAW64	$\mathsf{frame}(x) \wedge (p \vee q)$	$\equiv$	$\mathsf{frame}(x) \wedge (\mathsf{frame}(x) \wedge p \vee q)$
		$\equiv$	$\mathsf{frame}(x) \wedge (p \vee \mathsf{frame}(x) \wedge q)$
		$\equiv$	$\mathsf{frame}(x) \wedge (\mathsf{frame}(x) \wedge p \vee \mathsf{frame}(x) \wedge q)$
		$\equiv$	$\mathsf{frame}(x) \wedge p \vee \mathsf{frame}(x) \wedge q$
LAW65	$\mathsf{frame}(x) \wedge (p\,;\,q)$	$\equiv$	$\mathsf{frame}(x) \wedge (\mathsf{frame}(x) \wedge p\,;\,q)$
		$\equiv$	$\mathsf{frame}(x) \wedge (p\,;\,\mathsf{frame}(x) \wedge q)$
		$\equiv$	$\mathsf{frame}(x) \wedge (\mathsf{frame}(x) \wedge p\,;\,\mathsf{frame}(x) \wedge q)$
		$\equiv$	$\mathsf{frame}(x) \wedge p\,;\,\mathsf{frame}(x) \wedge q$
LAW66	$\mathsf{frame}(x) \wedge (p \parallel q)$	$\equiv$	$\mathsf{frame}(x) \wedge (\mathsf{frame}(x) \wedge p \parallel q)$
		$\equiv$	$\mathsf{frame}(x) \wedge (p \parallel \mathsf{frame}(x) \wedge q)$
		$\equiv$	$\mathsf{frame}(x) \wedge (\mathsf{frame}(x) \wedge p \parallel \mathsf{frame}(x) \wedge q)$
		$\equiv$	$\mathsf{frame}(x) \wedge p \parallel \mathsf{frame}(x) \wedge q$

Theorem 4.3. *Each satisfiable framed program has at least one minimal model on propositions.*

The proof of Theorem 4.3 can be found in the Appendix. This theorem asserts the existence of minimal models for a given framed program. The proof of the theorem requires more sophisticated technique than those used previously, and is based on the notion of a *normal form graph* (NFG) of a framed program introduced in the Appendix. Note that the proof itself provides a method for *constructing* a minimal model of a framed program.

It is worth noting that adding frame operators to PTL shifts the underlying semantics from a monotonic [15] to non-monotonic one, i.e. $\{w\} \vdash z$ does not necessarily imply $\{w, u\} \vdash z$. Intuitively, adding a new positive fact, i.e., an explicit assignment, to a set of positive facts within a framing context can make it impossible to infer the negation of the fact which was previously possible.

Remark 4.4. The equivalence relation $\equiv_{\mathfrak{m}}$ on framed programs is not preserved through conjunction, sequence, parallel composition, projection and chop plus operators. For example, if we take

$$p \stackrel{\text{df}}{=} (y = 1) \wedge \bigcirc(\neg\mathsf{af}(y) \rightarrow y = \ominus y) \qquad p' \stackrel{\text{df}}{=} (y = 1) \wedge \bigcirc(y = \ominus y)$$
$$q \stackrel{\text{df}}{=} (y = 1) \wedge \bigcirc(\neg\mathsf{af}(y) \rightarrow y = \ominus y) \wedge \bigcirc(y \Leftarrow 9) \qquad q' \stackrel{\text{df}}{=} (y = 1) \wedge \bigcirc(y \Leftarrow 9)$$

then we have $p \equiv_{\mathfrak{m}} p'$ and $q \equiv_{\mathfrak{m}} q'$. On the other hand, $p \wedge q \equiv_{\mathfrak{m}} q$ and $p' \wedge q' \equiv_{\mathfrak{m}} \mathsf{false}$. □

4.1. Normal form of framed programs

We now introduce a normal form of framed programs and establish a fundamental property that each framed program can be transformed into a normal form. Note that the normal form is similar to that of non-framed programs given in [6] except that it also involves the assignment flags.

Definition 4.2. A framed program q is in *normal form* if

$$q \;=\; \left(\bigvee_{i=1}^{k} \mathfrak{h}_i \wedge \mathsf{empty}\right) \vee \left(\bigvee_{j=1}^{h} \mathfrak{b}_j \wedge \bigcirc \mathfrak{f}_j\right)$$

where $k + h \geq 1$ and the following hold:

- Each $\bigcirc \mathfrak{f}_j$ is a lec-formula and $\mathfrak{f}_j$ is an *internal program*; the latter means variables may refer to the previous states but not beyond the first state of the current interval.[5]
- Each $\mathfrak{b}_j$ and $\mathfrak{h}_i$ is either true or a state formula of the form $p_1 \wedge \cdots \wedge p_m$ ($m \geq 1$) such that each p_z is either $(x = e)$ with $e \in \mathbf{D}$, or $aflag_x$, or $\neg aflag_x$. □

The conjuncts $\mathfrak{h}_i \wedge \mathsf{empty}$ and $\mathfrak{b}_j \wedge \bigcirc \mathfrak{f}_j$ are called *basic terminal products* and *basic future products*, respectively. Moreover, $\mathfrak{b}_j$ ($\mathfrak{h}_i$) and $\bigcirc \mathfrak{f}_j$ are called *present components* and *future components*, respectively. Note that we may always assume that all the present components are different.

Theorem 4.5. *For each program prog there is a program prog′ in normal form satisfying prog ≡ prog′.*

The proof of Theorem 4.5 can be found in the Appendix. It tells us that for each framed program there is an equivalent program in normal form. To take advantage of the normal form of framed programs, we need results allowing us to reduce programs in a convenient way.

Proposition 4.6. *The following are satisfied:*

1. $(x = e) \equiv (aflag_x \wedge (x = e)) \vee (\neg aflag_x \wedge (x = e))$.
2. $\mathsf{lbf}(x) \equiv aflag_x \vee (\neg aflag_x \wedge (x = \ominus x))$.

Proof. Follows from the definitions. □

We can use the above proposition in the reduction of framed programs, as shown below.

Example 4.2. For the framed program

$$\textsc{Prog IX}: \quad \mathsf{frame}(x) \wedge (x \Leftarrow 1) \wedge \bigcirc(x = 2) \wedge \mathsf{len}(1)$$

we have the following:

$$\begin{aligned}
& \mathsf{frame}(x) \wedge (x \Leftarrow 1) \wedge \bigcirc(x = 2) \wedge \mathsf{len}(1) \\
\equiv\ & \Box(\mathsf{more} \rightarrow \bigcirc \mathsf{lbf}(x)) \wedge (x \Leftarrow 1) \wedge \bigcirc(x = 2) \wedge \bigcirc(\mathsf{empty}) \\
\equiv\ & (\mathsf{more} \rightarrow \bigcirc \mathsf{lbf}(x)) \wedge \odot\Box(\mathsf{more} \rightarrow \bigcirc \mathsf{lbf}(x)) \wedge (x \Leftarrow 1) \wedge \bigcirc(x = 2) \wedge \mathsf{more} \wedge \bigcirc(\mathsf{empty}) \\
\equiv\ & \bigcirc \mathsf{lbf}(x) \wedge \bigcirc\Box(\mathsf{more} \rightarrow \bigcirc \mathsf{lbf}(x)) \wedge (x \Leftarrow 1) \wedge \bigcirc(x = 2) \wedge \bigcirc(\mathsf{empty}) \\
\equiv\ & (x \Leftarrow 1) \wedge \bigcirc(\mathsf{lbf}(x) \wedge \Box(\mathsf{more} \rightarrow \bigcirc \mathsf{lbf}(x)) \wedge (x = 2) \wedge \mathsf{empty}) \\
\equiv\ & (x = 1) \wedge aflag_x \wedge \bigcirc(\mathsf{lbf}(x) \wedge (x = 2) \wedge \mathsf{empty}).
\end{aligned}$$

Thus the normal form of PROG IX is

$$\left(\bigvee_{i=1}^{0} \mathfrak{h}_i \wedge \mathsf{empty}\right) \vee \left(\bigvee_{j=1}^{1} \mathfrak{b}_j \wedge \bigcirc \mathfrak{f}_j\right) \ = \ \mathfrak{b}_1 \wedge \bigcirc \mathfrak{f}_1$$

where:

$$\begin{aligned}
\mathfrak{b}_1 &\equiv (x = 1) \wedge aflag_x \\
\mathfrak{f}_1 &\equiv \mathsf{lbf}(x) \wedge (x = 2) \wedge \mathsf{empty} \\
&\equiv (aflag_x \vee (\neg aflag_x \wedge (x = \ominus x))) \wedge ((aflag_x \wedge (x = 2)) \vee (\neg aflag_x \wedge (x = 2)) \wedge \mathsf{empty} \\
&\equiv (aflag_x \wedge (x = 2) \wedge \mathsf{empty}) \vee (\neg aflag_x \wedge (x = 1) \wedge (x = 2) \wedge \mathsf{empty}) \\
&\equiv aflag_x \wedge (x = 2) \wedge \mathsf{empty}.
\end{aligned}$$

This shows that although there is no explicit assignments using $\Leftarrow$ at a state, a potential positive assignment can occur since $x = e$ can be treated as $(aflag_x \wedge (x = e)) \vee (\neg aflag_x \wedge (x = e))$. □

Note that framed programs can be non-deterministic, as there may be several models satisfying the program under the canonical models.

[5] For instance, $\bigcirc\ominus\ominus x$ is not permitted, but $\bigcirc\ominus x$ is.

PROG X : $\mathsf{frame}(x, y, s) \wedge (x, y, s) = (0, 0, 0) \wedge$
$\texttt{for}\ n\ \texttt{times do}\ ((x :=^{+} x + 1)\,;\ (y :=^{+} y + x)\,;\ (s :=^{+} s + y))$

PROG XI : $\mathsf{frame}(x, y, s) \wedge (x, y, s) = (0, 0, 0) \wedge$
$(\texttt{for}\ n\ \texttt{times do}\ (x :=^{+} x + 1)\,;\ \mathsf{len}(2)) \wedge$
$(\texttt{for}\ (n + 1)\ \texttt{times do}\ (y :=^{+} y + x)\,;\ \mathsf{skip}) \wedge$
$\texttt{for}\ (n + 2)\ \texttt{times do}\ (s :=^{+} s + y)$

PROG XII : $\mathsf{frame}(x, y, s) \wedge (x, y, s) = (0, 0, 0) \parallel$
$\texttt{for}\ n\ \texttt{times do}\ (x :=^{+} x + 1) \parallel$
$(\mathsf{skip}\,;\ \texttt{for}\ n\ \texttt{times do}\ (y :=^{+} y + x)) \parallel$
$(\mathsf{len}(2)\,;\ \texttt{for}\ n\ \texttt{times do}\ (s :=^{+} s + y))$

PROG XIII : $\mathsf{frame}(x, y, s) \wedge (x, y, s) = (0, 0, 0)\ \wedge$
$((x :=^{+} x + 1)\,;\ ((x, y) :=^{+} (x + 1, y + x))\,;$
$\texttt{for}\ n - 2\ \texttt{times do}\ ((x, y, s) :=^{+} (x + 1, y + x, s + y))\,;$
$((y, s,) :=^{+} (y + x, s + y))\,;\ (s :=^{+} s + y))$

PROG XIV : $\mathsf{frame}(x, y, s) \wedge (x, y, s) = (0, 0, 0)\ \wedge$
$\texttt{for}\ n\ \texttt{times do}\ (x :=^{+} x + 1)\ \wedge$
$((\mathsf{skip}\,;\ \texttt{for}\ n\ \texttt{times do}\ (y :=^{+} y + x))||(\mathsf{len}(2)\,;$
$\texttt{for}\ n\ \texttt{times do}\ (s :=^{+} s + y)))$

Fig. 5. Sequential, conjunctive, parallel, mixed sequential & conjunctive and mixed conjunctive & parallel computations.

Remark 4.7. There are two simple heuristics to be followed in the reduction of a framed program under the minimal model semantics:

1. Use the relation $\equiv$ as far as possible during a reduction;
2. Do not use the minimal model to obtain $\neg aflag_x$ until the last stage of a reduction, and make sure that x has really not been assigned a value by $\Leftarrow$ within all the conjuncts at the current state.

Moreover, the following facts are useful when carrying the reduction: Let q be a normal form of a framed program as in Definition 4.2.

1. If $aflag_x$ is not contained within the $\mathfrak{h}_i$'s and $\mathfrak{b}_j$'s then
$$q \equiv_{\mathfrak{m}} \neg aflag_x \wedge q.$$
2. If $aflag_x$ and $(x = e')$, where $e' \neq e$ $(e', e \in \mathbf{D})$, are not contained within the $\mathfrak{h}_i$'s and $\mathfrak{b}_j$'s then
$$\begin{aligned} (x = e) \wedge q &\equiv_{\mathfrak{m}} \neg aflag_x \wedge (x = e) \wedge q \\ (aflag_x \vee \neg aflag_x \wedge (x = e)) \wedge q &\equiv_{\mathfrak{m}} \neg aflag_x \wedge (x = e) \wedge q. \end{aligned}$$

5. Programming in framed tempura

In this section, we show how framing can be used in TLP; and some basic techniques in various types of programs including sequential, conjunctive, parallel, or mixed computations are illustrated with examples. Each example shown in Fig. 5 indicates a useful property.

We consider a simple computation: given a positive integer n, to compute the sum of all integers in the sequence $1, \ldots, n$; and the sum of the sums of integers in every prefix of the sequence $1, \ldots, n$, i.e., $1+(1+2)+\cdots+(1+\cdots+n)$. In the following, let n be a static variable, x, y, s dynamic variables. x represents an integer, y represents $1 + \cdots + n$, and s represents $1 + (1 + 2) + \cdots + (1 + \cdots + n)$. The programs are designed in different manners for the purpose of showing the different applications of framing techniques. In each case, Fig. 6 shows the computation of a program for $n = 4$.

```
sequential computation
      s0   s1   s2   s3   s4   s5   s6   s7   s8   s9  s10  s11  s12
      |----|----|----|----|----|----|----|----|----|----|----|----|
    x=0    1    1    1    2    2    2    3    3    3    4    4    4
    y=0    0    1    1    1    3    3    3    6    6    6   10   10
    s=0    0    0    1    1    1    4    4    4   10   10   10   20

conjunctive computation
      s0    s1    s2    s3    s4    s5    s6
      |-----|-----|-----|-----|-----|-----|
    x=0     1     2     3     4     4     4
    y=0     0     1     3     6    10    10
    s=0     0     0     1     4    10    20

parallel computation
      s0    s1    s2    s3    s4    s5    s6
      |-----|-----|-----|-----|-----|-----|
    x=0     1     2     3     4     4     4
    y=0     0     1     3     6    10    10
    s=0     0     0     1     4    10    20
```

Fig. 6. Different kinds of computations.

In each iteration, x is incremented by 1, y by x, and s by y. After n iterations, the computation sequences are $1, \ldots, n$ for x, $0, 1, 1+2, \ldots, 1+\cdots+n$ for y, and $0, 0, 1, 1+(1+2), \ldots, 1+(1+2)+\cdots+(1+2+\cdots+n)$ for s. Hence, the values of y and s at the final state are correct results.

At state $\mathfrak{s}_1, \mathfrak{s}_2$, the computations proceed sequentially, at state $\mathfrak{s}_3, \mathfrak{s}_4$, conjunctively by `for` iteration, and at state $\mathfrak{s}_5, \mathfrak{s}_6$, again sequentially.

Introducing the framing operator enables us not only to write concise programs but also to define await construct within the underlying logic. Hence, synchronized communication can be handled within Framed Tempura.

As discussed earlier, the communication between parallel components in Tempura is based on shared variables. To synchronize the real concurrent computation, we need a synchronous communication construct such as $\mathsf{await}(c)$ by which the semaphore constructs can be defined. In Tempura, the statement $\mathsf{halt}(c)$ may play a similar role as $\mathsf{await}(c)$. But, within a non-framed environment, they are actually different. $\mathsf{halt}(c)$ is capable of changing variables contained in c at the final state over an interval but $\mathsf{await}(c)$ is not although they both wait for c to become true and terminate the interval over which they act. The key difference between $\mathsf{await}(c)$ and $\mathsf{halt}(c)$ is that the former can only wait until another process acting in parallel changes c to true, while the latter can change c itself at the final state without the help of other processes acting in parallel.

Therefore, $\mathsf{halt}(c)$, in general, as a synchronization construct for concurrent computations, is not suitable. However, if the variables contained in c are all framed, and no positive assignments appear in c (this is usually satisfied because we consider c as a condition, i.e. a boolean expression), for this specific case, $\mathsf{halt}(c)$ is equivalent to $\mathsf{await}(c)$. This is immediate since the variables in c are framed and only positive assignments are able to change framed variables. For instance, $\mathsf{frame}(x) \wedge \mathsf{halt}(x = 1)$ is similar to $\mathsf{await}(x = 1)$.

Defining $\mathsf{await}(c)$ is difficult without some kind of framing construct since the values of variables are not inherited automatically from one state to another. But one requires some kind of indefinite stability, since it cannot be known at the point of use how long the waiting will last. At the same time one must also allow variables to change, so that an external process can modify the boolean parameter and it can eventually become true.

The await statement is defined as follows:

$$\mathsf{await}(c) \stackrel{\text{df}}{=} \mathsf{frame}(x_1) \wedge \cdots \wedge \mathsf{frame}(x_h) \wedge \mathsf{halt}(c)$$

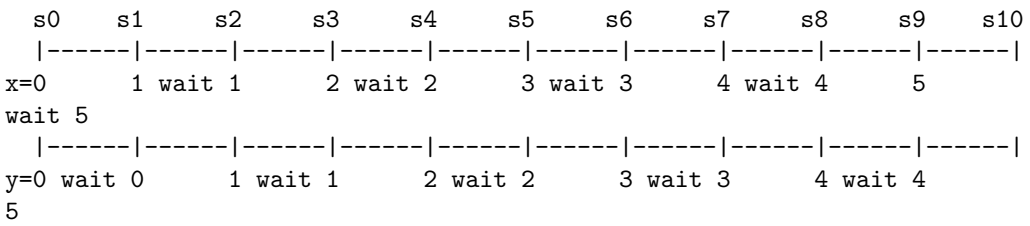

Fig. 7. A synchronized computation.

where $x_1, \ldots, x_h$ are dynamic variables contained in c. With the help of the frame operator, the semaphore statements (i.e., P–V operations) can be defined respectively by:

$$P(r) \stackrel{\text{df}}{=} \mathsf{frame}(r) \wedge (\mathsf{halt}(r > 0)\,;\ (r :=^{+} r - 1))$$
$$V(r) \stackrel{\text{df}}{=} \mathsf{frame}(r) \wedge (r :=^{+} r + 1)$$

where r is a dynamic variable initialized to 1.

Since $\mathsf{frame}(x)$ and $\mathsf{halt}(c)$ are both executable within the Framed Tempura, so is $\mathsf{await}(c)$. Therefore, synchronized communication for concurrent computations can be implemented. For instance, the following program synchronizes variables x and y in a parallel computation (see Fig. 7):

PROG XV :

$$\begin{array}{l}\mathsf{frame}(x) \wedge \mathsf{frame}(y) \wedge (x, y) = (0, 0) \wedge \\ \texttt{while}\ (x < 5)\ \texttt{do}\ (x :=^{+} x + 1\,;\ \mathsf{await}(y \geq x)) \\ \| \\ \texttt{while}\ (y < 5)\ \texttt{do}\ (\mathsf{await}(y < x)\,;\ y :=^{+} y + 1)\end{array}$$

6. Concluding remarks

In this paper, we presented a framing technique with some distinct advantages; in particular, it has simple semantics and can be used in different environments (including sequential, conjunctive, parallel and mixed contexts), and allows mixing of framed and non-framed variables. We believe that it allows one to write concise, efficient and elegant programs, which allow synchronization and communication between parallel components.

The temporal semantics of framed programs is captured using the minimal model. This model allows us to treat variables which are framed and not assigned new values by the positive immediate assignments in such a way that their values are inherited. The minimal model does it by means of perceiving the defaults of positive immediate assignments. It should be emphasized that by using the minimal model, the underlying logic is changed from monotonic to non-monotonic. This implies that some logic laws related to the monotonic law such as substitution law are no longer valid within framed programs.

In the future, we will further investigate the operational semantics and axiomatic semantics of framed programs. Furthermore, we will explore the consistency between the minimal model semantics and operational semantics. In addition, based on these semantics, verification of framed programs will also be investigated by means of the model checking technique.

Appendix. Proofs of selected results

Proof of Theorem 2.1. We need to show that, for all PTL formulas f, g, h, if $f \equiv g$ then:

(i) $f \wedge h \equiv g \wedge h$ (ii) $h \wedge f \equiv h \wedge g$
(iii) $\neg f \equiv \neg g$ (iv) $f^{+} \equiv g^{+}$
(v) $\bigcirc f \equiv \bigcirc g$ (vi) $\ominus f \equiv \ominus g$.

Each of (i)–(vi) is an immediate consequence of the definitions. We then take formulas $p_i \equiv p_i'$ $(1 \leq i \leq m)$ and $q \equiv q'$. We need to prove that

(vii) $\exists x : q \equiv \exists x : q'$
(viii) $(p_1, \ldots, p_m)\ \mathsf{prj}\ q \equiv (p_1', \ldots, p_m')\ \mathsf{prj}\ q'$.

Let σ be a model and k an integer such that $0 \leq k \preceq |\sigma|$. To show (vii) we observe that

$$\begin{aligned}
&(\sigma, 0, k, |\sigma|) \models \exists x : q \\
&\quad \Longleftrightarrow \quad (\sigma', 0, k, |\sigma'|) \models q \text{ for some } \sigma' \stackrel{x}{=} \sigma \\
&\quad \Longleftrightarrow \quad (\sigma', 0, k, |\sigma'|) \models q' \text{ for some } \sigma' \stackrel{x}{=} \sigma \\
&\quad \Longleftrightarrow \quad (\sigma, 0, k, |\sigma|) \models \exists x : q'.
\end{aligned}$$

To show (viii), let $\mathfrak{I} = (\sigma, 0, k, j)$ where $j = |\sigma|$. Then $\mathfrak{I} \models (p_1, \ldots, p_m)\ \mathsf{prj}\ q$ iff there are $k = r_0 \leq r_1 \leq \cdots \leq r_m \leq j$ such that $(\sigma, i, r_0, r_1) \models p_1$ and $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p_l$ for all $1 < l \leq m$, and $(\sigma', 0, 0, |\sigma'|) \models q$ for σ' given by:

1. If $r_m < j$ then $\sigma' = \sigma{\downarrow}(r_0, \ldots, r_m) \cdot \sigma_{(r_m+1..j)}$
2. If $r_m = j$ then $\sigma' = \sigma{\downarrow}(r_0, \ldots, r_h)$ for some $0 \leq h \leq m$.

Hence, by the induction hypothesis, $\mathfrak{I} \models (p_1', \ldots, p_m')\ \mathsf{prj}\ q'$. □

Proof of Theorem 2.2. Most of the laws can be found in [6,10]. We here prove some of them laws as others can be shown in the similar way. Note that $p \equiv q$ is equivalent to saying that, for every interpretation $\mathfrak{I}$, we have $\mathfrak{I} \models p$ iff $\mathfrak{I} \models q$. Below σ is an interval and k an integer such that $0 \leq k \preceq |\sigma|$.
Case 1: LAW1. Then

$$\begin{aligned}
(\sigma, 0, k, |\sigma|) \models \mathsf{empty}\,;\, p \ &\Longleftrightarrow\ (\sigma, 0, k, r) \models \mathsf{empty} \text{ and } (\sigma, r, r, |\sigma|) \models p \ \text{ for some } \ r \\
&\Longleftrightarrow\ k = r \text{ and } (\sigma, r, r, |\sigma|) \models p \\
&\Longleftrightarrow\ (\sigma, k, k, |\sigma|) \models p \\
&\Longleftrightarrow\ (\sigma, i, k, |\sigma|) \models p.
\end{aligned}$$

Note that the last equivalence holds since p is a lec-formula.
Case 2: LAW2. Then

$$\begin{aligned}
(\sigma, 0, k, |\sigma|) \models (p \wedge \mathsf{empty})\,;\, q \ &\Longleftrightarrow\ (\sigma, 0, k, r) \models p \wedge \mathsf{empty} \text{ and } (\sigma, r, r, |\sigma|) \models q \text{ for some } r \\
&\Longleftrightarrow\ (\sigma, 0, k, r) \models p \text{ and } r = k \text{ and } (\sigma, r, r, |\sigma|) \models q \\
&\Longleftrightarrow\ (\sigma, 0, k, k) \models p \text{ and } (\sigma, k, k, |\sigma|) \models q \\
&\Longleftrightarrow\ (\sigma, 0, k, |\sigma|) \models p \text{ and } (\sigma, 0, k, |\sigma|) \models q.
\end{aligned}$$

Note that the last equivalence holds since p is a rec-formula and q a lec-formula.
Case 3: LAW8. Then

$$\begin{aligned}
(\sigma, 0, k, |\sigma|) \models \bigcirc p \wedge \bigcirc q \ &\Longleftrightarrow\ (\sigma, 0, k+1, |\sigma|) \models p \text{ and } (\sigma, 0, k+1, |\sigma|) \models q \text{ and } k < |\sigma| \\
&\Longleftrightarrow\ (\sigma, 0, k+1, |\sigma|) \models p \wedge q \\
&\Longleftrightarrow\ (\sigma, 0, k, |\sigma|) \models \bigcirc(p \wedge q).
\end{aligned}$$

Case 4: LAW11. Then

$$\begin{aligned}
&(\sigma, 0, k, |\sigma|) \models w\,;\, p \vee q \\
&\qquad \Longleftrightarrow \quad (\sigma, 0, k, r) \models w \text{ and } ((\sigma, r, r, |\sigma|) \models p \text{ or } (\sigma, r, r, |\sigma|) \models q) \text{ for some } r \\
&\qquad \Longleftrightarrow \quad (\sigma, 0, k, r) \models w \text{ and } (\sigma, r, r, |\sigma|) \models p \text{ for some } r \text{ or} \\
&\qquad\qquad\quad\ (\sigma, 0, k, r) \models w \text{ and } (\sigma, r, r, |\sigma|) \models q \text{ for some } r \\
&\qquad \Longleftrightarrow \quad (\sigma, 0, k, |\sigma|) \models (w\,;\, p) \vee (w\,;\, q).
\end{aligned}$$

Case 5: LAW14. Then we need to show that $(\sigma, 0, k, |\sigma|) \models \bigcirc(\exists x : p) \leftrightarrow \exists x : \bigcirc p$. If $k = |\sigma|$, this is vacuously true. If $k < |\sigma|$, we have

$$\begin{aligned}
(\sigma, 0, k, |\sigma|) \models \bigcirc(\exists x : p) &\iff (\sigma, 0, k+1, |\sigma|) \models \exists x : p && \text{(i)}\\
&\iff (\sigma', 0, k+1, |\sigma'|) \models p \text{ for some } \sigma' \stackrel{x}{=} \sigma && \text{(ii)}\\
&\iff (\sigma', 0, k, |\sigma'|) \models \bigcirc p \text{ for some } \sigma' \stackrel{x}{=} \sigma && \text{(iii)}\\
&\iff (\sigma, 0, k, |\sigma|) \models \exists x : \bigcirc p. && \text{(iv)}
\end{aligned}$$

In the above: (i) follows from INEXT; (ii) from IEXISTS; (iii) from INEXT; and (iv) from IEXISTS.
Case 6: LAW15. Then

$$\begin{aligned}
(\sigma, 0, k, |\sigma|) \models \bigcirc p \,;\, q &\iff (\sigma, 0, k, r) \models \bigcirc p \text{ and } (\sigma, r, r, |\sigma|) \models q \text{ for some } r\\
&\iff (\sigma, 0, k+1, r) \models p \text{ and } k < r \text{ and } (\sigma, r, r, |\sigma|) \models q \text{ for some } r\\
&\iff (\sigma, 0, k+1, |\sigma|) \models p \,;\, q\\
&\iff (\sigma, 0, k, |\sigma|) \models \bigcirc(p \,;\, q).
\end{aligned}$$

Case 7: LAW26. Then:

$$\begin{aligned}
p^* &\equiv \mathsf{empty} \vee p^+\\
&\equiv \mathsf{empty} \vee (p \vee (p \,;\, p^+)) && \text{(i)}\\
&\equiv \mathsf{empty} \vee ((p \,;\, \mathsf{empty}) \vee (p \wedge \Box\mathsf{more}) \vee (p \,;\, p^+)) && \text{(ii)}\\
&\equiv \mathsf{empty} \vee (p \,;\, (\mathsf{empty} \vee p^+)) \vee (p \wedge \Box\mathsf{more}) && \text{(iii)}\\
&\equiv \mathsf{empty} \vee (p \,;\, p^*) \vee (p \wedge \Box\mathsf{more}).
\end{aligned}$$

In the above: (i) follows from LAW20; (ii) from LAW22; and (iii) from LAW11.
Case 8: LAW27. Then

$$\begin{aligned}
p^* &\equiv \mathsf{empty} \vee p^+ && \text{(i)}\\
&\equiv \mathsf{empty} \vee (p \vee (p \wedge \mathsf{more} \,;\, p^+)) && \text{(ii)}\\
&\equiv \mathsf{empty} \vee (p \,;\, \mathsf{empty}) \vee p \wedge \Box\mathsf{more} \vee (p \wedge \mathsf{more} \,;\, p^+) && \text{(iii)}\\
&\equiv \mathsf{empty} \vee ((p \wedge \mathsf{empty} \vee p \wedge \mathsf{more}) \,;\, \mathsf{empty}) \vee (p \wedge \mathsf{more} \,;\, p^+) \vee p \wedge \Box\mathsf{more}\\
&\equiv \mathsf{empty} \vee (p \wedge \mathsf{empty} \,;\, \mathsf{empty}) \vee (p \wedge \mathsf{more} \,;\, \mathsf{empty}) \vee (p \wedge \mathsf{more} \,;\, p^+) \vee p \wedge \Box\mathsf{more} && \text{(iv)}\\
&\equiv \mathsf{empty} \vee p \wedge \mathsf{empty} \vee (p \wedge \mathsf{more} \,;\, \mathsf{empty}) \vee (p \wedge \mathsf{more} \,;\, p^+) \vee p \wedge \Box\mathsf{more} && \text{(v)}\\
&\equiv \mathsf{empty} \vee (p \wedge \mathsf{more} \,;\, \mathsf{empty}) \vee (p \wedge \mathsf{more} \,;\, p^+) \vee p \wedge \Box\mathsf{more}\\
&\equiv \mathsf{empty} \vee (p \wedge \mathsf{more} \,;\, (\mathsf{empty} \vee p^+)) \vee p \wedge \Box\mathsf{more} && \text{(vi)}\\
&\equiv \mathsf{empty} \vee (p \wedge \mathsf{more} \,;\, p^*) \vee p \wedge \Box\mathsf{more} && \text{(vii)}
\end{aligned}$$

In the above: (i) follows from ACHOP-STAR; (ii) from LAW21; (iii) from LAW22; (iv) from LAW11; (v) from LAW19; (vi) from LAW11; and (vii) from ACHOP-STAR.
Case 9: LAW42. Then: $\mathfrak{I} = (\sigma, 0, k, |\sigma|)$, $\mathbf{p} = p_1, \ldots, p_m$ and $\mathbf{q} = q_1, \ldots, q_n$. We proceed as follows:

$\mathfrak{I} \models (\mathbf{p}, \mathsf{empty} \wedge w, p, \mathbf{q})$ prj q iff there exist integers $r_1, \ldots r_t, r_e, r_{t+1}, \ldots r_m$ and $r_0 = k$ such that $\mathfrak{I} \models [p_1, \ldots p_t, \mathsf{empty} \wedge p, p_{t+1}, \ldots p_m](r_1, \ldots r_t, r_e, r_{t+1}, \ldots r_m)$ and $r_m = |\sigma|$ and $\sigma \downarrow (r_0, \ldots r_h) \models q$ for some $0 \le h \le m$ or $h = e\mathbb{C}$ and $\sigma \downarrow \ldots r_h$

we have $(\sigma, r_t, r_t, r_t) \models p \Leftrightarrow (\sigma, r_t, r_t, r_{t+1}) \models p$. Hence

$$\begin{array}{ll} & (\sigma, 0, k, r_1) \models p_1 \text{ and } (\sigma, r_{l-1}, r_{l-1}, r_l) \models p \text{ for all } l,\ 1 < l \leq t \text{ and} \\ \Longleftrightarrow & (\sigma, r_t, r_t, r_{t+1}) \models p \wedge p_{t+1} \text{ and} \\ & (\sigma, r_{l-1}, r_{l-1}, r_l) \models p_l \text{ for all } t+1 < l \leq m \\ \Longleftrightarrow & (\sigma, 0, k, |\sigma|) \models [p_1, \ldots p_t, p \wedge p_{t+1}, \ldots p_m](r_1, \ldots r_t, r_{t+1}, \ldots r_m). \end{array}$$

Moreover

$$\begin{array}{ll} & \sigma \downarrow (r_0, \ldots r_t, r_e) = \sigma \downarrow (r_0, \ldots r_t)\ (r_t = r_e) \\ \text{and} & \sigma \downarrow (r_0, \ldots r_t, r_e, r_{t+1}, \ldots r_m) = \sigma \downarrow (r_0, \ldots r_t, r_{t+1}, \ldots r_m)(r_t = r_e). \end{array}$$

Therefore

$$(\sigma, 0, k, |\sigma|) \models (p_1, \ldots p_t, \mathsf{empty} \wedge p, p_{t+1}, \ldots p_m)\ \mathsf{prj}\ q \quad \equiv \quad (\mathbf{p}, w \wedge p, \mathbf{q})\ \mathsf{prj}\ q. \quad \square$$

Proof of Theorem 3.1. Case 1: LAW59. Then:

$$\begin{array}{lll} \texttt{while}\ b\ \texttt{do}\ p & & \\ \quad \equiv & (b \wedge p)^* \wedge \mathsf{fin}(\neg b) & \text{(i)} \\ \quad \equiv & (\mathsf{empty} \vee (b \wedge p \wedge \mathsf{more}\,;\ (b \wedge p)^*)) \wedge \mathsf{fin}(\neg b) \vee b \wedge p \wedge \Box\mathsf{more} \wedge \mathsf{fin}(\neg b) & \text{(ii)} \\ \quad \equiv & \mathsf{empty} \wedge \Box(\mathsf{empty} \rightarrow \neg b) \vee (b \wedge p \wedge \mathsf{more}\,;\ (b \wedge p)^*) \wedge \mathsf{fin}(\neg b) \vee b \wedge p \wedge \Box\mathsf{more} & \text{(iii)} \\ \quad \equiv & \mathsf{empty} \wedge (\mathsf{empty} \rightarrow \neg b) \vee (b \wedge p \wedge \mathsf{more}\,;\ (b \wedge p)^*) \wedge \mathsf{fin}(\neg b) \vee b \wedge p \wedge \Box\mathsf{more} & \text{(iv)} \\ \quad \equiv & (\mathsf{empty} \wedge \neg b) \vee (b \wedge p \wedge \mathsf{more}\,;\ \mathsf{fin}(\neg b) \wedge (b \wedge pp)^*) \vee b \wedge p \wedge \Box\mathsf{more} & \text{(v)} \\ \quad \equiv & (\mathsf{empty} \wedge \neg b) \vee (b \wedge p \wedge \mathsf{more}\,;\ \texttt{while}\ b\ \texttt{do}\ p) \vee b \wedge p \wedge \Box\mathsf{more}. & \text{(vi)} \end{array}$$

In the above: (i) follows from definition; (ii) from LAW27; (iii) from $(\Box\mathsf{more}) \wedge \mathsf{fin}(\neg b) \equiv \Box\mathsf{more}$; (iv) from LAW24; (v) from LAW52; and (vi) from definition.

Case 2: LAW57. Then:

$$\begin{array}{lll} \texttt{while}\ b\ \texttt{do}\ p & \equiv & (\mathsf{empty} \wedge \neg b) \vee (b \wedge p\,;\ \texttt{while}\ b\ \texttt{do}\ p) \\ & \equiv & (\mathsf{empty} \wedge \neg b) \vee b \wedge (p\,;\ \texttt{while}\ b\ \texttt{do}\ p) \\ & \equiv & \texttt{if}\ b\ \texttt{then}\ (p\,;\ \texttt{while}\ b\ \texttt{do}\ p)\ \texttt{else}\ \mathsf{empty}. \end{array}$$

Case 3: LAW58. Then

$$\begin{array}{lll} \texttt{while}\ b\ \texttt{do}\ p & \equiv & (\mathsf{empty} \wedge \neg b) \vee (b \wedge p \wedge \mathsf{more}\,;\ \texttt{while}\ b\ \texttt{do}\ p) \\ & \equiv & (\mathsf{empty} \wedge \neg b) \vee b \wedge (p \wedge \mathsf{more}\,;\ \texttt{while}\ b\ \texttt{do}\ p) \\ & \equiv & \texttt{if}\ b\ \texttt{then}\ (p \wedge \mathsf{more}\,;\ \texttt{while}\ b\ \texttt{do}\ p)\ \texttt{else}\ \mathsf{empty}. \end{array}$$

Case 4: LAW60. Then

$$\begin{array}{lll} \texttt{while}\ b\ \texttt{do}\ p & & \\ \quad \equiv & (b \wedge p)^* \wedge \mathsf{fin}(\neg b) & \text{(i)} \\ \quad \equiv & (\mathsf{empty} \vee (b \wedge p\,;\ (b \wedge p)^*)) \wedge \mathsf{fin}(\neg b) \vee b \wedge p \wedge \Box\mathsf{more} \wedge \mathsf{fin}(\neg b) & \text{(ii)} \\ \quad \equiv & \mathsf{empty} \wedge \Box(\mathsf{empty} \rightarrow \neg b) \vee (b \wedge p\,;\ (b \wedge p)^*) \wedge \mathsf{fin}(\neg b) \vee b \wedge p \wedge \Box\mathsf{more} & \text{(iii)} \\ \quad \equiv & \mathsf{empty} \wedge (\mathsf{empty} \rightarrow \neg b) \vee (b \wedge p\,;\ (b \wedge p)^*) \wedge \mathsf{fin}(\neg b) \vee b \wedge p \wedge \Box\mathsf{more} & \text{(iv)} \\ \quad \equiv & (\mathsf{empty} \wedge \neg b) \vee (b \wedge p\,;\ \mathsf{fin}(\neg b) \wedge (b \wedge p)^*) \vee b \wedge p \wedge \Box\mathsf{more} & \text{(v)} \\ \quad \equiv & (\mathsf{empty} \wedge \neg b) \vee (b \wedge p\,;\ \texttt{while}\ b\ \texttt{do}\ p) \vee b \wedge p \wedge \Box\mathsf{more}. & \text{(vi)} \end{array}$$

In the above: (i, vi) follow from definitions; (ii) from LAW26; (iii) from $(\Box\mathsf{more}) \wedge \mathsf{fin}(\neg b) \equiv \Box\mathsf{more}$; (iv) from LAW24; and (v) from LAW52.

Proof of Theorem 4.1. We here prove some of them laws as others can be shown in the similar way.

Case 1: LAW62. Then

$$\begin{aligned}
\mathsf{frame}(x) \wedge \mathsf{more} &\equiv \Box(\mathsf{more} \to \bigcirc\mathsf{lbf}(x)) \wedge \mathsf{more} && \text{(i)}\\
&\equiv (\mathsf{more} \to \bigcirc\mathsf{lbf}(x)) \wedge \odot\Box(\mathsf{more} \to \bigcirc\mathsf{lbf}(x)) \wedge \mathsf{more} && \text{(ii)}\\
&\equiv \bigcirc\mathsf{lbf}(x) \wedge \mathsf{more} \wedge \bigcirc\Box(\mathsf{more} \to \bigcirc\mathsf{lbf}(x)) && \text{(iii)}\\
&\equiv \bigcirc\mathsf{lbf}(x) \wedge \mathsf{more} \wedge \bigcirc\mathsf{frame}(x) && \text{(iv)}\\
&\equiv \bigcirc(\mathsf{lbf}(x) \wedge \mathsf{frame}(x)) \wedge \mathsf{more} && \text{(v)}\\
&\equiv \bigcirc(\mathsf{lbf}(x) \wedge \mathsf{frame}(x)). && \text{(vi)}
\end{aligned}$$

In the above: (i) follows from def.lbf; (ii) from LAW24; (iii) from LAW13; (iv) from the definition of lbf; (v) from LAW8; and (vi) from LAW25.

Case 2: LAW64. Then

$$\begin{aligned}
&\mathfrak{I} \models \mathsf{frame}(x) \wedge (p \vee q)\\
&\quad\Longleftrightarrow\quad \mathfrak{I} \models \mathsf{frame}(x) \text{ and } \mathfrak{I} \models p \vee q && \text{(i)}\\
&\quad\Longleftrightarrow\quad \mathfrak{I} \models \mathsf{frame}(x) \text{ and } (\mathfrak{I} \models p \text{ or } \mathfrak{I} \models q)\\
&\quad\Longleftrightarrow\quad (\mathfrak{I} \models \mathsf{frame}(x) \text{ and } \mathfrak{I} \models p) \text{ or } (\mathfrak{I} \models \mathsf{frame}(x) \text{ and } \mathfrak{I} \models q)\\
&\quad\Longleftrightarrow\quad \mathfrak{I} \models \mathsf{frame}(x) \wedge p \text{ or } \mathfrak{I} \models \mathsf{frame}(x) \wedge q && \text{(ii)}\\
&\quad\Longleftrightarrow\quad \mathfrak{I} \models \mathsf{frame}(x) \wedge p \vee \mathsf{frame}(x) \wedge q
\end{aligned}$$

where $\mathfrak{I}$ is an interpretation. In the above: (i) and (ii) follow from IAND.

Case 3: LAW65. Then

$$\begin{aligned}
&\mathfrak{I} \models \mathsf{frame}(x) \wedge (\mathsf{frame}(x) \wedge p\,;\, q)\\
&\quad\Longleftrightarrow\quad \mathfrak{I} \models \mathsf{frame}(x) \text{ and } \mathfrak{I} \models (\mathsf{frame}(x) \wedge p\,;\, q) && \text{(i)}\\
&\quad\Longleftrightarrow\quad \mathfrak{I} \models \mathsf{frame}(x) \text{ and}\\
&\qquad\quad (\sigma, 0, k, r) \models \mathsf{frame}(x) \wedge p \text{ and } (\sigma, r, r, |\sigma|) \models q \text{ for some } r && \text{(ii)}\\
&\quad\Longleftrightarrow\quad \mathfrak{I} \models \mathsf{frame}(x) \text{ and } (\sigma, 0, k, r) \models \mathsf{frame}(x) \text{ and}\\
&\qquad\quad (\sigma, 0, k, r) \models p \text{ and } (\sigma, r, r, |\sigma|) \models q \text{ for some } r && \text{(iii)}\\
&\quad\Longleftrightarrow\quad \mathfrak{I} \models \mathsf{frame}(x) \text{ and } (\sigma, 0, k, r) \models \mathsf{frame}(x) \text{ and } \mathfrak{I} \models p\,;\, q \text{ for some } r && \text{(iv)}\\
&\quad\Longleftrightarrow\quad \mathfrak{I} \models \mathsf{frame}(x) \text{ and } \mathfrak{I} \models p\,;\, q && \text{(v)}\\
&\quad\Longleftrightarrow\quad \mathfrak{I} \models \mathsf{frame}(x) \wedge (p\,;\, q) && \text{(vi)}
\end{aligned}$$

where $\mathfrak{I} = (\sigma, 0, k, |\sigma|)$ is an interpretation. In the above: (i) follows from IAND; (ii) from ICHOP; (iii) from IAND; (iv) from ICHOP; (vi) from IAND; and (v) from the fact that if σ is an interval and $0 \le k \le r \preceq |\sigma|$, then

$$(\sigma, 0, k, |\sigma|) \models \mathsf{frame}(x) \Longrightarrow (\sigma, 0, k, r) \models \mathsf{frame}(x).$$

Case 4: LAW66. Then

$$\begin{aligned}
&\mathfrak{I} \models \mathsf{frame}(x) \wedge (\mathsf{frame}(x) \wedge p \parallel q)\\
&\quad\Longleftrightarrow\quad \mathfrak{I} \models \mathsf{frame}(x) \wedge ((\mathsf{frame}(x) \wedge p\,;\, \mathsf{true}) \wedge q) \vee \mathsf{frame}(x) \wedge p \wedge (q\,;\, \mathsf{true})) && \text{(i)}\\
&\quad\Longleftrightarrow\quad \mathfrak{I} \models \mathsf{frame}(x) \wedge q \wedge (\mathsf{frame}(x) \wedge p\,;\, \mathsf{true}) \vee \mathsf{frame}(x) \wedge p \wedge (q\,;\, \mathsf{true}) && \text{(ii)}\\
&\quad\Longleftrightarrow\quad \mathfrak{I} \models \mathsf{frame}(x) \wedge q \wedge (p\,;\, \mathsf{true}) \vee \mathsf{frame}(x) \wedge p \wedge (q\,;\, \mathsf{true}) && \text{(iii)}\\
&\quad\Longleftrightarrow\quad \mathfrak{I} \models \mathsf{frame}(x) \wedge (q \wedge (p\,;\, \mathsf{true}) \vee p \wedge (q\,;\, \mathsf{true})) && \text{(iv)}\\
&\quad\Longleftrightarrow\quad \mathfrak{I} \models \mathsf{frame}(x) \wedge (p \parallel q). && \text{(v)}
\end{aligned}$$

In the above: (i) and (v) follow from definitions; (ii) from LAW64 and LAW61; (iii) from LAW65; and (iv) from LAW64. □

Proof of Theorem 4.5. To simplify the proof, we assume that the expressions involved in a framed program are well evaluated at every state during a reduction. The proof proceeds by induction on the structure of a program (cases not dealt with below can be handled in a similar way).

Case 1: $prog = (x = e)$. Then:

$$\begin{aligned} prog &\equiv ((x = e) \wedge aflag_x \vee (x = e) \wedge \neg aflag_x) \wedge (\mathsf{empty} \vee \mathsf{more}) \\ &\equiv ((x = e) \wedge aflag_x \vee (x = e) \wedge \neg aflag_x) \wedge \mathsf{empty} \vee \\ &\quad ((x = e) \wedge aflag_x \vee (x = e) \wedge \neg aflag_x) \wedge \bigcirc\mathsf{true} \\ &\equiv ((x = e) \wedge aflag_x \wedge \mathsf{empty}) \vee (x = e \wedge \neg aflag_x \wedge \mathsf{empty}) \vee (((x = e) \wedge aflag_x) \wedge \bigcirc\mathsf{true}) \\ &\quad \vee (((x = e) \wedge \neg aflag_x) \wedge \bigcirc\mathsf{true}). \end{aligned}$$

An immediate assignment is a non-deterministic program, so it needs other constructs to specify a definite interval. If an empty interval is specified, it is reduced to

$$((x = e) \wedge aflag_x \wedge \mathsf{empty}) \vee ((x = e) \wedge \neg aflag_x \wedge \mathsf{empty})$$

otherwise, it is reduced to

$$(((x = e) \wedge aflag_x) \wedge \bigcirc\mathsf{true}) \vee (((x = e) \wedge \neg aflag_x) \wedge \bigcirc\mathsf{true}).$$

Case 2: $prog = (x \Leftarrow e)$. Then:

$$\begin{aligned} prog &\equiv ((x = e) \wedge aflag_x) \wedge (\mathsf{empty} \vee \mathsf{more}) \\ &\equiv ((x = e) \wedge aflag_x \wedge \mathsf{empty}) \vee (((x = e) \wedge aflag_x) \wedge \bigcirc\mathsf{true}). \end{aligned}$$

Case 3: $prog = \mathsf{empty}$. Then the result clearly holds.

Case 4: $prog = \bigcirc p$. Then $prog$ is already in normal form.

Case 5: $prog = \Box q$. Then:

$$\begin{aligned} prog &\equiv q \wedge \odot\Box q && \text{(i)} \\ &\equiv \left(\bigvee(\mathfrak{h}_j \wedge \mathsf{empty}) \vee \bigvee(\mathfrak{b}_i \wedge \bigcirc\mathfrak{f}_i)\right) \wedge (\mathsf{empty} \vee \bigcirc\Box q) && \text{(ii)} \\ &\equiv \bigvee(\mathfrak{h}_j \wedge \mathsf{empty}) \vee \bigvee((\mathfrak{b}_i \wedge \bigcirc\mathfrak{f}_i) \wedge \bigcirc\Box q) && \text{(iii)} \\ &\equiv \bigvee(\mathfrak{h}_j \wedge \mathsf{empty}) \vee \bigvee(\mathfrak{b}_i \wedge \bigcirc(\mathfrak{f}_i \wedge \bigcirc\Box q)). && \text{(iv)} \end{aligned}$$

In the above: (i) follows from LAW24; (ii) from the hypothesis and AWNEXT; (iii) from AMORE; and (iv) from LAW8.

Case 6: $prog = p \wedge q$, where (here and below)

$$p \equiv \bigvee_{k=1}^{l_1}(\mathfrak{p}_k \wedge \mathsf{empty}) \vee \bigvee_{i=1}^{t_1}(\mathfrak{r}_i \wedge \bigcirc\mathfrak{t}_i) \quad \text{and} \quad q \equiv \bigvee_{h=1}^{l_2}(\mathfrak{h}_h \wedge \mathsf{empty}) \vee \bigvee_{j=1}^{t_2}(\mathfrak{b}_j \wedge \bigcirc\mathfrak{f}_j)$$

and we can rewrite $prog$, as follows:

$$\begin{aligned} prog &\equiv \left(\bigvee(\mathfrak{p}_k \wedge \mathsf{empty}) \vee \bigvee(\mathfrak{r}_i \wedge \bigcirc\mathfrak{t}_i)\right) \wedge \left(\bigvee(\mathfrak{h}_h \wedge \mathsf{empty}) \vee \bigvee(\mathfrak{b}_j \wedge \bigcirc\mathfrak{f}_j)\right) && \text{(i)} \\ &\equiv \bigvee(\mathfrak{p}_k \wedge \mathfrak{h}_k \wedge \mathsf{empty}) \vee \left(\bigvee(\mathfrak{r}_i \wedge \bigcirc\mathfrak{t}_i) \wedge \bigvee(\mathfrak{b}_j \wedge \bigcirc\mathfrak{t}_j)\right) && \text{(ii)} \\ &\equiv \bigvee(\mathfrak{p}_k \wedge \mathfrak{h}_k \wedge \mathsf{empty}) \vee \bigvee_{1 \le i \le t_1, 1 \le j \le t_2}((\mathfrak{r}_i \wedge \mathfrak{b}_j) \wedge \bigcirc(\mathfrak{t}_i \wedge \mathfrak{f}_j)). && \text{(iii)} \end{aligned}$$

In the above: (i) follows from the hypothesis; (ii) from AMORE and LAW25; and (iii) from LAW8.

Case 7: $prog = p\,;\,q$. Then:

$$
\begin{aligned}
prog &\equiv \Big(\bigvee(\mathfrak{p}_k \wedge \mathsf{empty}) \vee \bigvee(\mathfrak{r}_i \wedge \bigcirc \mathfrak{t}_i)\Big)\,;\,\Big(\bigvee(\mathfrak{h}_h \wedge \mathsf{empty}) \vee \bigvee(\mathfrak{b}_j \wedge \bigcirc \mathfrak{f}_j)\Big) && \text{(i)}\\
&\equiv \Big(\bigvee(\mathfrak{p}_k \wedge \mathsf{empty})\,;\,\bigvee(\mathfrak{h}_h \wedge \mathsf{empty})\Big) \vee \Big(\bigvee(\mathfrak{r}_i \wedge \bigcirc \mathfrak{t}_i)\,;\,\bigvee(\mathfrak{h}_h \wedge \mathsf{empty})\Big) \vee\\
&\quad \Big(\bigvee(\mathfrak{p}_k \wedge \mathsf{empty})\,;\,\bigvee(\mathfrak{b}_j \wedge \bigcirc \mathfrak{f}_j)\Big) \vee \Big(\bigvee(\mathfrak{r}_i \wedge \bigcirc \mathfrak{t}_i)\,;\,\bigvee(\mathfrak{b}_j \wedge \bigcirc \mathfrak{f}_j)\Big) && \text{(ii)}\\
&\equiv \Big(\bigvee(\mathfrak{p}_k \wedge \mathfrak{h}_k \wedge \mathsf{empty})\Big) \vee \Big(\bigvee(\mathfrak{r}_i \wedge \bigcirc(\mathfrak{t}_i\,;\,(\mathfrak{h}_1 \wedge \mathsf{empty})))\Big) \vee\\
&\quad \Big(\bigvee(\mathfrak{p}_1 \wedge \mathfrak{b}_j \wedge \bigcirc \mathfrak{f}_j)\Big) \vee \Big(\bigvee(\mathfrak{r}_i \wedge \bigcirc(\mathfrak{t}_i\,;\,\bigvee(\mathfrak{b}_j \wedge \bigcirc \mathfrak{f}_j)))\Big) && \text{(iii)}\\
&\equiv \bigvee(\mathfrak{p}_k \wedge \mathfrak{h}_k \wedge \mathsf{empty}) \vee \bigvee(\mathfrak{r}_i \wedge \bigcirc(\mathfrak{t}_i\,;\,(\mathfrak{h}_1 \wedge \mathsf{empty}))) \vee\\
&\quad \bigvee((\mathfrak{p}_1 \wedge \mathfrak{b}_j) \wedge \bigcirc \mathfrak{f}_j) \vee \bigvee_{1\le i\le t_1, 1\le j\le t_2} (\mathfrak{r}_i \wedge \bigcirc(\mathfrak{t}_i\,;\,(\mathfrak{b}_j \wedge \bigcirc \mathfrak{f}_j))). && \text{(iv)}
\end{aligned}
$$

In the above: (i) follows from the hypothesis; (ii) from LAW11 and LAW12; (iii) from LAW30, LAW15, LAW11 and LAW12; and (iv) from LAW9 and LAW11.

Case 8: $prog = p \parallel q$. First, we have that:

$$
\begin{aligned}
(q\,;\,\mathsf{true}) &\equiv \Big(\bigvee(\mathfrak{h}_h \wedge \mathsf{empty}) \vee \bigvee(\mathfrak{b}_j \wedge \bigcirc \mathfrak{f}_j)\Big)\,;\,\mathsf{true} && \text{(i)}\\
&\equiv \Big(\bigvee(\mathfrak{h}_h \wedge \mathsf{empty})\,;\,\mathsf{true}\Big) \vee \Big(\bigvee(\mathfrak{b}_j \wedge \bigcirc \mathfrak{f}_j)\,;\,\mathsf{true}\Big) && \text{(ii)}\\
&\equiv \bigvee \mathfrak{h}_h \wedge (\mathsf{empty}\,;\,\mathsf{true}) \vee \bigvee(\mathfrak{b}_j \wedge \bigcirc(\mathfrak{f}_j\,;\,\mathsf{true})) && \text{(iii)}\\
&\equiv \bigvee \mathfrak{h}_h \wedge \mathsf{true} \vee \bigvee(\mathfrak{b}_j \wedge \bigcirc(\mathfrak{f}_j\,;\,\mathsf{true})) && \text{(iv)}\\
&\equiv \bigvee(\mathfrak{h}_h \wedge \mathsf{empty}) \vee \mathfrak{h}_1 \wedge \bigcirc\mathsf{true} \vee \bigvee(\mathfrak{b}_j \wedge \bigcirc(\mathfrak{f}_j\,;\,\mathsf{true}))\\
&\equiv \bigvee(\mathfrak{h}_h \wedge \mathsf{empty}) \vee \bigvee_{j=1}^{t_2+1}(\mathfrak{b}_j \wedge \bigcirc(\mathfrak{f}_j\,;\,\mathsf{true}))
\end{aligned}
$$

where $\mathfrak{b}_{t_2+1} \equiv \mathfrak{h}_1$ and $\mathfrak{f}_{t_2+1} \equiv \mathsf{true}$. In the above: (i) follows from the hypothesis; (ii) from LAW12; (iii) from LAW15 and LAW16; and (iv) from $\mathsf{empty}\,;\,\mathsf{true} \equiv \mathsf{true}$. Thus

$$
\begin{aligned}
p \wedge (q\,;\,\mathsf{true}) &\equiv \Big(\bigvee(\mathfrak{p}_k \wedge \mathsf{empty}) \vee \bigvee(\mathfrak{r}_i \wedge \bigcirc \mathfrak{t}_i)\Big)\\
&\qquad \wedge \left(\bigvee(\mathfrak{h}_h \wedge \mathsf{empty}) \vee \bigvee_{j=1}^{t_2+1}(\mathfrak{b}_j \wedge \bigcirc(\mathfrak{f}_j\,;\,\mathsf{true}))\right) && \text{(i)}\\
&\equiv \bigvee(\mathfrak{p}_k \wedge \mathfrak{h}_k \wedge \mathsf{empty}) \vee \Big(\bigvee(\mathfrak{r}_i \wedge \bigcirc \mathfrak{t}_i)\Big) \wedge \bigvee_{j=1}^{t_2+1}(\mathfrak{b}_j \wedge \bigcirc(\mathfrak{f}_j\,;\,\mathsf{true})) && \text{(ii)}\\
&\equiv \bigvee(\mathfrak{p}_k \wedge \mathfrak{h}_k \wedge \mathsf{empty}) \vee \bigvee_{1\le i\le t_1, 1\le j\le t_2+1} ((\mathfrak{r}_i \wedge \mathfrak{b}_j) \wedge \bigcirc(\mathfrak{t}_i \wedge (\mathfrak{f}_j\,;\,\mathsf{true}))).
\end{aligned}
$$

In the above: (i) follows from the hypothesis; and (ii) from AMORE. Similarly, we can prove that

$$
q \wedge (p\,;\,\mathsf{true}) \quad \equiv \quad \bigvee(\mathfrak{h}_k \wedge \mathfrak{p}_k \wedge \mathsf{empty}) \vee \left(\bigvee_{1\le i\le t_1+1, 1\le j\le t_2} ((\mathfrak{b}_i \wedge \mathfrak{r}_j) \wedge \bigcirc(\mathfrak{f}_i \wedge (\mathfrak{t}_j\,;\,\mathsf{true})))\right).
$$

As a consequence, we have that

$$\begin{aligned}
prog &\equiv p \wedge (q \,;\, \mathsf{true}) \vee q \wedge (p \,;\, \mathsf{true}) \\
&\equiv \bigvee(\mathfrak{p}_k \wedge \mathfrak{h}_k \wedge \mathsf{empty}) \vee \left(\bigvee_{1\le i\le t_1, 1\le j\le t_2+1} ((\mathfrak{r}_i \wedge \mathfrak{b}_j) \wedge \bigcirc(\mathfrak{t}_i \wedge (\mathfrak{f}_j \,;\, \mathsf{true}))) \right) \\
&\quad \vee \bigvee(\mathfrak{h}_k \wedge \mathfrak{p}_k \wedge \mathsf{empty}) \vee \left(\bigvee_{1\le i\le t_1+1, 1\le j\le t_2} ((\mathfrak{b}_i \wedge \mathfrak{r}_j) \wedge \bigcirc(\mathfrak{f}_i \wedge (\mathfrak{t}_j \,;\, \mathsf{true}))) \right) \\
&\equiv \bigvee(\mathfrak{p}_k \wedge \mathfrak{h}_k \wedge \mathsf{empty}) \vee \left(\bigvee_{1\le i\le t_1, 1\le j\le t_2+1} ((\mathfrak{r}_i \wedge \mathfrak{b}_j) \wedge \bigcirc(\mathfrak{t}_i \wedge (\mathfrak{f}_j \,;\, \mathsf{true}))) \right) \\
&\quad \vee \left(\bigvee_{1\le i\le t_1+1, 1\le j\le t_2} ((\mathfrak{b}_i \wedge \mathfrak{r}_j) \wedge \bigcirc(\mathfrak{f}_i \wedge (\mathfrak{t}_j \,;\, \mathsf{true}))) \right).
\end{aligned}$$

Case 9: $prog = (p_1, \ldots, p_m)\ \mathsf{prj}\ q$, where q is a non-past formula. Suppose

$$q \equiv \bigvee(\mathfrak{h}_r \wedge \mathsf{empty}) \vee \bigvee(\mathfrak{b}_i \wedge \bigcirc \mathfrak{f}_i)$$

and, for $1 \le l \le m$,

$$p_l \equiv \bigvee(\mathfrak{p}_{lr} \wedge \mathsf{empty}) \vee \bigvee(\mathfrak{r}_{lj} \wedge \bigcirc \mathfrak{t}_{lj}).$$

Then

$$\begin{aligned}
prog &\equiv (p_1, \ldots, p_m)\ \mathsf{prj}\ \left(\bigvee(\mathfrak{h}_r \wedge \mathsf{empty}) \vee \bigvee(\mathfrak{b}_i \wedge \bigcirc \mathfrak{f}_i) \right) && \text{(i)} \\
&\equiv (p_1, \ldots, p_m)\ \mathsf{prj}\ (\mathfrak{h}_1 \wedge \mathsf{empty}) \vee \bigvee(p_1, \ldots, p_m)\ \mathsf{prj}\ ((\mathfrak{b}_i \wedge \bigcirc \mathfrak{f}_i)). && \text{(ii)}
\end{aligned}$$

In the above: (i) follows from the hypothesis; and (ii) from Law44. Furthermore, we have

$$\begin{aligned}
(p_1, \ldots, p_m)\ \mathsf{prj}\ (\mathfrak{h}_1 \wedge \mathsf{empty}) &\equiv \mathfrak{h}_1 \wedge (p_1, \ldots, p_m)\ \mathsf{prj}\ \mathsf{empty} && \text{(i)} \\
&\equiv \mathfrak{h}_1 \wedge (p_1 \,;\, \ldots \,;\, p_m). && \text{(ii)}
\end{aligned}$$

In the above: (i) follows from Law51; and (ii) from Law39. By Case 7, $(p_1 \,;\, \ldots \,;\, p_m)$ can be reduced to a normal form and so can $\mathfrak{h}_1 \wedge (p_1 \,;\, \ldots \,;\, p_m)$. Hence

$$\mathfrak{h}_1 \wedge (p_1 \,;\, \ldots \,;\, p_m) \equiv \bigvee(\mathfrak{p}_r \wedge \mathsf{empty}) \vee \bigvee(\mathfrak{r}_j \wedge \bigcirc \mathfrak{t}_j). \qquad (*)$$

Moreover, we have that

$$\begin{aligned}
& (p_1, \ldots, p_m)\ \mathsf{prj}\ (\mathfrak{b}_i \wedge \bigcirc \mathfrak{f}_i) \\
\equiv\ & \mathfrak{b}_i \wedge ((p_1, \ldots, p_m)\ \mathsf{prj}\ \bigcirc \mathfrak{f}_i) && \text{(i)} \\
\equiv\ & \mathfrak{b}_i \wedge ((p_1 \wedge p_2 \wedge \cdots \wedge p_m \wedge \mathsf{empty}) \,;\, \bigcirc \mathfrak{f}_i) \\
& \vee \mathfrak{b}_i \wedge \bigvee_{t=0}^{m-1} ((p_0 \wedge \cdots \wedge p_t \wedge \mathsf{empty}) \,;\, p_{t+1} \wedge \mathsf{more} \,;\, (p_{t+2}, \ldots, p_{m+1})\ \mathsf{prj}\ \mathfrak{f}_i) && \text{(ii)} \\
\equiv\ & \mathfrak{b}_i \wedge ((\mathfrak{p}_{11} \wedge \cdots \wedge \mathfrak{p}_{m1} \wedge \mathsf{empty}) \,;\, \bigcirc \mathfrak{f}_i) \\
& \vee \mathfrak{b}_i \wedge \bigvee_{t=0}^{m-1} \Bigg((\mathfrak{p}_{01} \wedge \mathfrak{p}_{11} \wedge \mathfrak{p}_{21} \wedge \cdots \wedge \mathfrak{p}_{t1} \wedge \mathsf{empty}) \,; \\
& \bigvee_{j=1}^{k_{t+1}} \mathfrak{r}_{t+1,j} \wedge \bigcirc \mathfrak{t}_{t+1,j} \wedge \mathsf{more} \,;\, (p_{t+2}, \ldots, p_{m+1})\ \mathsf{prj}\ \mathfrak{f}_i \Bigg) && \text{(iii)}
\end{aligned}$$

$$
\begin{aligned}
\equiv\ & \mathfrak{b}_i \wedge \mathfrak{p}_{11} \wedge \cdots \wedge \mathfrak{p}_{m1} \wedge (\mathsf{empty}\,;\ \bigcirc q_f) \\
& \vee \mathfrak{b}_i \wedge \bigvee_{t=0}^{m-1} \left((\mathfrak{p}_{01} \wedge \mathfrak{p}_{11} \wedge \cdots \wedge \mathfrak{p}_{t1} \wedge \mathsf{empty})\,;\ \bigvee_{j=1}^{k_{t+1}} \mathfrak{r}_{t+1\,j} \wedge \bigcirc \mathfrak{t}_{t+1\,j}\,;\ (p_{t+2}, \ldots, p_{m+1})\ \mathsf{prj}\ \mathfrak{f}_i \right) \quad \text{(iv)} \\
\equiv\ & \mathfrak{b}_i \wedge \mathfrak{p}_{11} \wedge \cdots \wedge \mathfrak{p}_{m1} \wedge \bigcirc \mathfrak{f}_i \\
& \vee \bigvee_{1 \le j \le k_1} (\mathfrak{b}_i \wedge \mathfrak{r}_{1j} \wedge \bigcirc pf_{1j}\,;\ (p_2, \ldots, p_m)\ \mathsf{prj}\ \mathfrak{f}_i) \\
& \vee \bigvee_{1 \le t \le m-1, 1 \le j \le k_{t+1}} (\mathfrak{b}_i \wedge \mathfrak{p}_{11} \wedge \cdots \wedge \mathfrak{p}_{t1} \wedge \mathfrak{r}_{t+1,j} \wedge \bigcirc \mathfrak{t}_{t+1,j}\,;\ (p_{t+2}, \ldots, p_{m+1})\ \mathsf{prj}\ \mathfrak{f}_i) \quad \text{(v)} \\
\equiv\ & \mathfrak{b}_i \wedge \mathfrak{p}_{11} \wedge \cdots \wedge \mathfrak{p}_{m1} \wedge \bigcirc \mathfrak{f}_i \\
& \vee \bigvee_{1 \le j \le k_1} \mathfrak{b}_i \wedge \mathfrak{r}_{1j} \wedge \bigcirc (\mathfrak{t}_{1j}\,;\ (p_2, \ldots, p_m)\ \mathsf{prj}\ \mathfrak{f}_i) \\
& \vee \bigvee_{1 \le t \le m-1, 1 \le j \le k_{t+1}} (\mathfrak{b}_i \wedge \mathfrak{p}_{11} \wedge \cdots \wedge \mathfrak{p}_{t1} \wedge \mathfrak{r}_{t+1,j} \wedge \bigcirc (\mathfrak{t}_{t+1,j}\,;\ (p_{t+2}, \ldots, p_{m+1})\ \mathsf{prj}\ \mathfrak{f}_i). \quad \text{(vi)}
\end{aligned}
$$

In the above: (i) follows from LAW51; (ii) from LAW56; (iii) from $\mathfrak{p}_{01} \equiv \mathsf{empty}$; (iv) from LAW16 and LAW25; (v) from LAW11 and LAW2; and (vi) from LAW15. Therefore,

$$
\begin{aligned}
& \bigvee (p_1, \ldots, p_m)\ \mathsf{prj}\ (\mathfrak{b}_i \wedge \bigcirc \mathfrak{f}_i) \\
\equiv\ & \bigvee \mathfrak{b}_i \wedge \mathfrak{p}_{11} \wedge \cdots \wedge \mathfrak{p}_{m1} \wedge \bigcirc \mathfrak{f}_i \\
& \vee \bigvee_{1 \le j \le k_1, 1 \le i \le h} \mathfrak{b}_i \wedge \mathfrak{r}_{1j} \wedge \bigcirc (\mathfrak{t}_{1j}\,;\ (p_2, \ldots, p_m)\ \mathsf{prj}\ \mathfrak{f}_i) \\
& \vee \bigvee_{1 \le t \le m-1, 1 \le j \le k_{t+1}, 1 \le i \le h} (\mathfrak{b}_i \wedge \mathfrak{p}_{11} \wedge \cdots \wedge \mathfrak{p}_{t1} \wedge \mathfrak{r}_{t+1,j} \wedge \bigcirc (\mathfrak{t}_{t+1,j}\,;\ (p_{t+2}, \ldots, p_{m+1})\ \mathsf{prj}\ \mathfrak{f}_i). \quad (**)
\end{aligned}
$$

From (*) and (**) it follows that *prog* can be reduced to normal form.
Case 10: $prog =$ `if` b `then` p `else` q. Then if b is true in a context in which *prog* is executed then *prog* is reduced to p and otherwise to q.
Case 11: $prog =$ `while` b `do` p. Then, by LAW59, we have

$$\texttt{while}\ b\ \texttt{do}\ p \ \equiv\ \neg b \wedge \mathsf{empty} \vee (b \wedge p \wedge \mathsf{more}\,;\ \texttt{while}\ b\ \texttt{do}\ p) \vee p \wedge b \wedge \Box\mathsf{more}.$$

Therefore, if b is false at the current state according to the context in which *prog* is executed, then the `while` statement is reduced to empty. Otherwise, it is reduced to

$$p \wedge \mathsf{more}\,;\ \texttt{while}\ b\ \texttt{do}\ p \vee p \wedge \Box\mathsf{more}$$

which is immediately re-reduced, as follows:

$$
\begin{aligned}
& p \wedge \mathsf{more}\,;\ \texttt{while}\ b\ \texttt{do}\ p\ \vee p \wedge \Box\mathsf{more} \\
& \quad \equiv p \wedge (\bigcirc\mathsf{true}\,;\ \texttt{while}\ b\ \texttt{do}\ p\ \vee \mathsf{more} \wedge \odot\Box\mathsf{more}) && \text{(i)} \\
& \quad \equiv p \wedge (\bigcirc(\mathsf{true}\,;\ \texttt{while}\ b\ \texttt{do}\ p\ ) \vee \bigcirc\Box\mathsf{more}) && \text{(ii)} \\
& \quad \equiv p \wedge \bigcirc((\mathsf{true}\,;\ \texttt{while}\ b\ \texttt{do}\ p\ ) \vee \Box\mathsf{more}) && \text{(iii)} \\
& \quad \equiv \left(\bigvee \mathfrak{p}_k \wedge \mathsf{empty} \vee \bigvee \mathfrak{r}_i \wedge \bigcirc \mathfrak{t}_i\right) \wedge \bigcirc((\mathsf{true}\,;\ \texttt{while}\ b\ \texttt{do}\ p\ ) \vee \Box\mathsf{more}) && \text{(iv)} \\
& \quad \equiv \left(\bigvee \mathfrak{r}_i \wedge \bigcirc \mathfrak{t}_i\right) \wedge \bigcirc((\mathsf{true}\,;\ \texttt{while}\ b\ \texttt{do}\ p\ ) \vee \Box\mathsf{more}) && \text{(v)} \\
& \quad \equiv \bigvee \mathfrak{r}_i \wedge \bigcirc(\mathfrak{t}_i \wedge ((\mathsf{true}\,;\ \texttt{while}\ b\ \texttt{do}\ p\ ) \vee \Box\mathsf{more})). && \text{(vi)}
\end{aligned}
$$

In the above: (i) follows from LAW24; (ii) from LAW13 and LAW15; (iii) from LAW9; (iv) and (v) from the hypothesis; and (vi) from LAW8 and LAW11.

Case 12: $prog = \mathsf{frame}(x)$. Then:

$$\begin{aligned}
prog &\stackrel{\mathrm{df}}{=} \Box(\mathsf{more} \to \bigcirc\mathsf{lbf}(x)) \\
&\equiv \Box\left(\neg\mathsf{more} \vee \bigcirc\mathsf{lbf}(x)\right) \\
&\equiv \Box\left(\mathsf{empty} \vee \bigcirc\mathsf{lbf}(x)\right) && \text{(i)} \\
&\equiv \left(\mathsf{empty} \vee \bigcirc\mathsf{lbf}(x)\right) \wedge \odot\Box\left(\mathsf{empty} \vee \bigcirc\mathsf{lbf}(x)\right) && \text{(ii)} \\
&\equiv \left(\mathsf{empty} \wedge \odot\Box(\mathsf{empty} \vee \bigcirc\mathsf{lbf}(x))\right) \vee \left(\bigcirc\mathsf{lbf}(x)\right. \\
&\quad \left. \wedge \odot\Box(\mathsf{empty} \vee \bigcirc\mathsf{lbf}(x))\right) \\
&\equiv \mathsf{empty} \wedge \left(\mathsf{empty} \vee \bigcirc\Box\left(\mathsf{empty} \vee \bigcirc\mathsf{lbf}(x)\right)\right) \vee \\
&\quad \bigcirc\mathsf{lbf}(x) \wedge \left(\mathsf{empty} \vee \bigcirc\Box\left(\mathsf{empty} \vee \bigcirc\mathsf{lbf}(x)\right)\right) && \text{(iii)} \\
&\equiv \mathsf{empty} \vee \bigcirc\mathsf{lbf}(x) \wedge \bigcirc\Box\left(\mathsf{empty} \vee \bigcirc\mathsf{lbf}(x)\right) \\
&\equiv \mathsf{empty} \vee \bigcirc\left(\mathsf{lbf}(x) \wedge \Box\left(\mathsf{empty} \vee \bigcirc\mathsf{lbf}(x)\right)\right).
\end{aligned}$$

In the above: (i) follows from $\mathsf{more} \stackrel{\mathrm{df}}{=} \neg\mathsf{empty}$; (ii) from LAW24; and (iii) from $\odot p \stackrel{\mathrm{df}}{=} \mathsf{empty} \vee \bigcirc p$.

By Proposition 4.6(2), we have $\mathsf{lbf}(x) \equiv aflag_x \vee \neg aflag_x \wedge x = \ominus x$. If $aflag_x = \mathsf{true}$, then the reduction is:

$$\mathsf{empty} \vee \bigcirc(\mathsf{lbf}(x) \wedge \Box(\mathsf{empty} \vee \bigcirc\mathsf{lbf}(x))) \equiv \mathsf{empty} \vee \bigcirc(aflag_x \wedge \Box(\mathsf{empty} \vee \bigcirc aflag_x))$$

otherwise $aflag_x = \mathsf{false}$, the reduction is

$$\begin{aligned}
&\mathsf{empty} \vee \bigcirc(\mathsf{lbf}(x) \wedge \Box(\mathsf{empty} \vee \bigcirc\mathsf{lbf}(x))) \\
&\quad \equiv \quad \mathsf{empty} \vee \bigcirc((\neg p_{x_k} \wedge x = \ominus x) \wedge \Box(\mathsf{empty} \vee \bigcirc(\neg aflag_x \wedge x = \ominus x))).
\end{aligned}$$

Case 13: $prog = (\exists x : p(x))$. Then we first use the renaming method to reduce this to $p(y)$, as in Proposition 2.3, and then apply the induction hypothesis. □

Proof of Theorem 4.3. Armed with the normal form, a program q can be decomposed to a so-called Normal Form Graph(NFG) [11] as follows (see Fig. 8(a)):

Initially, the root (denoted by a small double circle) of the Graph is labelled by program q, each basic product in the normal form of q becomes a son of q. With the terminal product, the edge labelled by present component $\mathfrak{h}$ and a terminal vertex labelled by a small black dot without appearing of empty; and with the future product, the edge labelled by $\mathfrak{b}_j$ and the next vertex (a small circle) labelled by next component $\mathfrak{f}_j$. Then, $\mathfrak{f}_j$ can further be reduced to a sub-graph of q and so on. If two vertices are identical, we merge them into one. It is clear that if q has only finite models, its NFG is also finite.

In order to distinguish operations between sequences and sets, we denote a finite canonical interpretation sequence $(I^0_{prop}, \ldots, I^k_{prop})$ by $\overline{I}_k$, and its corresponding coded set $\{(0, I^0_{prop}), \ldots, (k, I^k_{prop})\}$ by I_k. Thus, for an arbitrary canonical interpretation sequence $\overline{I} = (I^0_{prop}, \ldots, I^h_{prop})$ $(h \in N_0)$, its corresponding coded set is $I = \{(0, I^0_{prop}), \ldots, (h, I^h_{prop})\}$.

For convenience, we need to add extra information to the NFG of a program p. First, the label of each edge, i.e. present component in the normal form of p, e.g. $\mathfrak{h}_i$ or $\mathfrak{b}_j$ is changed to corresponding canonical interpretation on propositions I^h_{prop} for some $h \geq 0$ ($(h+1)$th edge from root on a path) ignoring program variables. For instance, if $\mathfrak{b}_j \equiv aflag_x \wedge x = 1 \wedge aflag_y \wedge aflag_z \wedge y = 2$, then $I^h_{prop} = \{aflag_x, aflag_y, aflag_z\}$. Second, a node is given an extra label I_k. The initial node is labelled by $I_{-1} = \phi$. With a node I_k ($(k+2)$th node from root), to find out the next edge with minimal canonical interpretation, we define a function $e - min$ as follows:

$$\begin{aligned}
&e - min(I_k) = min\left\{I^{k+1,(1)}_{prop}, \ldots, I^{k+1,(h)}_{prop}\right\} = I^{k+1,(i)}_{prop} \ (I^{k+1}_{prop} \text{ for short}), \text{ if } I^{k+1,(i)}_{prop} \\
&\subset I^{k+1,(j)}_{prop} \text{ or } I^{k+1,(i)}_{prop} \text{ is not comparable with } I^{k+1,(j)}_{prop}, \text{ for } \forall j, i \neq j, 0 \leq i, j \leq h
\end{aligned}$$

where $I^{k+1,(1)}_{prop}, \ldots, I^{k+1,(h)}_{prop}$ are all canonical interpretations associated with edges departing from node I_k. By Theorem 4.5, a framed program p can be reduced to its normal form. Since p is satisfiable, so p has at least one

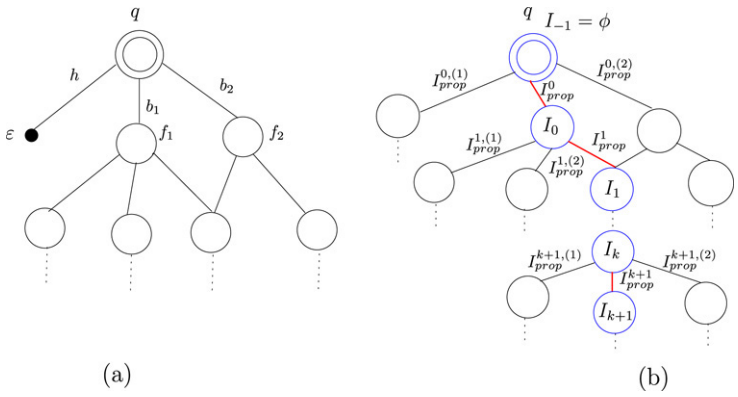

Fig. 8. NFG.

canonical model. Thus, we can construct its NFG as shown in Fig. 8(b). Let $I = \{(0, I^0_{prop}), (1, I^1_{prop}), (2, I^2_{prop}), \ldots\}$. We can define a function T over 2^I. For any $I' \subseteq I$, $T(I')$ is defined as:

$$T(I') = \{(n, I^n_{prop}) | \exists I_{n-1} \subseteq I', I^n_{prop} = e - min(I_{n-1}), n \geq 0\}$$

$e - min(I_{n-1})$ is a function returning the minimal interpretation among all canonical interpretations associated with edges departing from node I_{n-1}.

Initially $I_{-1} = \emptyset$, then we repeatedly apply function T to sets $I_{-1}, I_0, \ldots$. Thus, we have, $I_0 = T(I_{-1}) = \{(0, I^0_{prop})\}$, $I_1 = T(I_0) = \{(0, I^0_{prop}), (1, I^1_{prop})\}$, $I_2 = T(I_1) = T^2(I_0) = \{(0, I^0_{prop}), (1, I^1_{prop}), (2, I^2_{prop})\}$, $\ldots$, $I_n = T(I_{n-1}) = T^n(I_0) = \{(0, I^0_{prop}), (1, I^1_{prop}), \ldots, (n, I^n_{prop})\}$. Hence, $I_0 \subseteq I_1 \subseteq I_2 \subseteq \cdots \subseteq I_n \subseteq \cdots$. It is readily to see that T is monotonic.

Let $I = \bigcup_{n=0}^{\infty} I_n$, $\overline{I}_n$ stands for the prefix of minimal interpretation sequence $\overline{I}$. We now prove the following conclusions:

1. $\overline{I}$ is a canonical interpretation sequence of p.
We first prove $T(I) = I$.
(1) $T(I) \subseteq I$. For $\forall (n+1, I^{n+1}_{prop}) \in T(I)$, by definition, $(n+1, I^{n+1}_{prop}) \in I_{n+1} \subseteq I$, we have $(n+1, I^{n+1}_{prop}) \in I$. Hence $T(I) \subseteq I$.
(2) $I \subseteq T(I)$. For $\forall (n, I^n_{prop}) \in I$, since $(n, I^n_{prop}) \in I_n = T(I_{n-1})$ and $I_{n-1} \subseteq I$, by definition of T, $(n, I^n_{prop}) \in T(I)$, hence, $I \subseteq T(I)$.
By the above, $T(I) = I$. Thus, I is a fix-point of T.

2. Let $M = \{\sigma | \sigma \models_c p\}$ and $\sigma \in M, \sigma_p = \overline{I}$. Then σ is a minimal model of p.
Suppose $\exists \sigma' \in M, \sigma' \sqsubseteq \sigma$. We prove $\sigma = \sigma'$ by induction on $I^n_{prop} = I'^n_{prop}$.
(1) Since $\sigma' \sqsubseteq \sigma$, so $\sigma'_{prop} \sqsubseteq \sigma_{prop}$, i.e. $I'^i_{prop} \subseteq I^i_{prop}$ for all i, $0 \leq i \leq |\sigma|$. Thus $I'^0_{prop} \subseteq I^0_{prop}$, by definition of T, $I^0_{prop} \subseteq I'^0_{prop}$, therefore $I^0_{prop} = I'^0_{prop}$.
(2) Suppose for $n \leq k$ $(0 \leq k \leq |\sigma|)$, $I^h_{prop} = I'^h_{prop}$ $(0 \leq h \leq k)$. Let $n = k+1$. Since $I'^{k+1}_{prop} \subseteq I^{k+1}_{prop}$, on the other hand, by definition of T, $I^{k+1}_{prop} \subseteq I'^{k+1}_{prop}$, so $I^{k+1}_{prop} = I'^{k+1}_{prop}$. Therefore $\sigma = \sigma'$.
Consequently, σ is a minimal model of program p. □

References

[1] H. Barringer, M. Fisher, D. Gabbay, G. Gough, R. Owens, METATEM: A framework for programming in temporal logic, in: Proceedings of REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formulisms, Correctness, in: LNCS, vol. 430, Springer-Verlag, 1989.
[2] N. Bidoit, Negation in rule-based data base languages: A survey, Theoretical Computer Science 78 (1991) 3–83.
[3] H. Bowman, H. Cameron, P. King, S. Thompson, Specication and prototyping of structrued multimedia documents using interval temporal logic, in: Proceedings of International Conference on Temporal Logic, in: Applied Logic Series, Kluwer, 1997.
[4] H. Bowman, S. Thompson, A Tableau method for interval temporal logic with projection, in: Proceedings of TABLEAUX98, in: LNAI, vol. 1397, Springer-Verlag, 1998.

[5] H. Bowman, S. Thompson, A decision procedure and complete axiomatisation of finite interval temporal logic with projection, Journal of Logic and Computation 13 (2003) 195–239.
[6] Z. Duan, An extended interval temporal logic and a framing technique for temporal logic programming, Ph.D. Thesis, University of Newcastle upon Tyne, 1996.
[7] Z. Duan, Temporal Logic and Temporal Logic Programming, Science Press, Beijing, ISBN: 7-03-016651-5/TP.3158, 2006.
[8] Z. Duan, M. Holcombe, A. Bell, A logic for biological systems, BioSystems 55 (2000) 93–105.
[9] Z. Duan, M. Koutny, A framed temporal logic programming language, Journal of Computer Science and Technology 19 (2004) 341–351.
[10] Z. Duan, M. Koutny, C. Holt, Projection in temporal logic programming, in: F. Pfenning (Ed.), Proceedings of Logic Programming and Automatic Reasoning, in: LNAI, vol. 822, Springer-Verlag, 1994, pp. 333–344.
[11] Z. Duan, X. Yang, M. Koutny, Semantics of framed temporal logic programs, in: Proceedings of ICLP 2005, in: LNCS, vol. 3668, Springer-Verlag, 2005, pp. 356–370.
[12] E.A. Emerson, E.M. Clarke, Using branching temporal logic to synthesize synchronization skeletons, Science of Computer Programming 2 (1982) 241–266.
[13] M. Fisher, Towards a semantics for concurrent MTETATEM, in: Executable Modal and Temporal Logics, in: LNAI, vol. 897, Springer-Verlag, 1995.
[14] M. Fujita, S. Kono, H. Tanaka, T. Moto-oka, Tokio: Logic programming language based on temporal logic and its compilation to PROLOG, in: Proceedings of 3rd International Conference on Logic Programming, in: LNCS, vol. 225, Springer-Verlag, 1986, pp. 695–709.
[15] D.M. Gabbay, Theoretical foundations for non-monotonic reasoning in expert systems, Research Report 84/11, Dept. of Computing, Imperial College, 1984.
[16] R.W.S. Hale, Programming in temporal logic, Ph.D. Thesis, Cambridge University, 1988.
[17] E.C.R. Hehner, A practical theory of programming, Science of Computer Programming 14 (1990) 133–158.
[18] C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall International, 1985.
[19] F. Kröger, Temporal Logic of Programs, Springer-Verlag, 1987.
[20] L. Lamport, The temporal logic of actions, ACM Transaction on Programming Languages and Systems 16 (1994).
[21] Z. Manna, A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems, Springer-Verlag, 1992.
[22] R. Milner, Communication and Concurrency, Prentice Hall, 1989.
[23] B. Moszkowski, Executing Temporal Logic Programs, Cambridge University Press, 1986.
[24] B. Moszkowski, Some very compositional temporal properties, in: Programming Concepts, Methods and Calculi, Elsevier Science B.V., North-Holland, 1994, pp. 307–326.
[25] L. Ness, L.0: A parallel executable temporal logic language, in: Proceeding of ACM SIGSOFT, International Workshop on Formal Methods in Software Development, 1990.
[26] C.S. Tang, Toward a unified logic basis for programming languages, in: Proceedings of IFIP Congress, Elsevier Science Publishers B.V., North-Holland, 1983, pp. 425–429.
[27] C.S. Tang, A temporal logic language oriented toward software engineering,—introduction to XYZ system (I), Chinese Journal of Advanced Software Research 1 (1994) 1–27.