Operational semantics of Framed Tempura

Xiaoxiao Yang, Zhenhua Duan*

Institute of Computing Theory and Technology, Xidian University, Xi’an, Shaanxi, China

ARTICLE INFO

Article history:
Received 27 January 2008
Revised 23 July 2008
Accepted 5 August 2008
Available online 20 September 2008

Keywords:
Temporal logic programming
Non-monotonicity
Minimal model
Operational semantics
Consistency

ABSTRACT

This paper investigates the operational semantics of temporal logic programs. To this end, a temporal logic programming language called Framed Tempura is employed. The evaluation rules for both the arithmetic and Boolean expressions are defined. The semantic equivalence rules for the reduction of a program within a state is formalized. Furthermore, the transition rules within a state and transition rules over an interval between configurations are also specified. Moreover, some examples are given to illustrate how these rules work. Thus, the executable behavior of framed programs can be captured in an operational way. In addition, the consistency between the operational semantics and the minimal model semantics based on model theory is proved in detail.

© 2008 Elsevier Inc. All rights reserved.

1. Introduction

Temporal logic (TL) [1,2,26,27,30,34] was proposed for the purpose of specification and verification of concurrent systems. However, verification has suffered from a disadvantage that different languages have been used for writing programs, writing about properties of programs, and writing about whether and how a program satisfies a given property. One way to simplify this is to use the same language in each case. Therefore, a number of programming languages based on temporal logics have emerged in the past two decades, such as XYZ/E [31], Tempura [26], TOKIO [33], Framed Tempura [18] and METATEM [12,14], and others [3,29,13,15]. However, many aspects of programming in temporal logic programming languages are not well investigated (at least in Tempura). For instance, one such an aspect is the problem of framing, and the other is synchronous communication for parallel processes.

Projection temporal logic (PTL) [16,17,18,21] is an interval-based temporal logic, which is a useful formalism for reasoning about period of time with hardware and software systems. It can handle both sequential and parallel compositions, and offer useful and practical proof techniques for verifying concurrent systems. PTL augments interval temporal logic (ITL) [26] to include infinite models, past temporal operators (such as previous operator (\( \preceq \))) and a new projection operator (prj) for dealing with concurrent computation. The key construct used in PTL is the new projection construct, \((p_1, \ldots, p_m) prj q\), which can be thought of as a combination of the original parallel and projection constructs in ITL [26]. Further, an executable subset of PTL, called Framed Tempura, is developed. Framed Tempura can implement not only the basic statements in Tempura [26] but also the new projection operator (prj), synchronous communication (await), and framing (frame). The related contributions about PTL and Framed Tempura can be found in [16,17,18,19,20].

For convenience of the discussion, we assume that readers have basic knowledge of TL and briefly introduce only some necessary concepts relevant to PTL and ITL. The detailed formal definitions of these concepts are given in Section 2. An interval
is a finite or infinite sequence of states which map atomic propositions to Boolean values and variables to their domain. The length of a finite interval is the number of states minus 1 and specified by operator \( \text{len}(I) \) (\( I \in \mathbb{N}_0 \)) while the length of an infinite interval is \( \omega \). An interval with length 0 is denoted by \( \text{empty} \). The basic temporal operators are next (\( \boxtimes \)) and chop (\( ; \)). \( \bigcirc P \) holds over an interval if \( P \) holds from the next state; \( P; Q \) holds over an interval iff the interval can be split into 2 intervals and \( P \) holds over the first one while \( Q \) holds over the second one. The derived formula \( \text{halt}(P) \) holds over an interval iff \( P \) holds only at the last state.

1.1. Framing issue in temporal logic programming

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. Temporal logic programming (TLP) offers no ready-made solution in this respect as variables’ values are not assumed to be carried forward. In the 1990s, framing issue was discussed amongst researchers interested in TLP. However, although considerable attention has been given to framing [10,11,34,35,32], no consensus has yet emerged as to what is the best underlying semantics of framing operators and how to well formalize framing in a satisfiable and practicable manner. Firstly, let us begin with a simple example in Tempura [26]:

\[
\text{Program}(1) : (I = 1) \land (J := 2) : (J := I + J)
\]

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 \( I = 1 \) at the initial state of an interval over which the program is interpreted, and that \( J = 2 \) at the second state. One might expect (or even require) that \( J = 3 \) (the sum of the values of \( I \) and \( J \)) at the third state, but the program does not guarantee this. The reason is that \( I \) is unspecified at the second state, and so \( J \) 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 values of variables explicit. The above example would then be rewritten as:

\[
\text{Program}(2) : (I = 1) \land (\square \text{more} \to (\bigcirc I = 1)) \land (J := 2) : (J := I + J)
\]

where \( \square \) is the modal operator associated with the temporal “always”, \( \bigcirc \) is the “next” operator, and \( \text{more} \) means that the current interval is not yet over.

Now \( I \) 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. It is therefore important to have an efficient method (a framing technique, for instance) allowing one to carry forward values of variables, from the current state to the next.

There are at least two realistic ways to go about framing. One way is directly associated with the definition of the assignment operator as [11] does in an algebraic programming language. There the assignment is defined as follows:

\[
(x := e) \overset{\text{def}}{=} (x' = e) \land (y_1' = y_1) \land \cdots \land (y_m' = y_m),
\]

where the apostrophes represent the new values of variables, and \( y_1, \ldots, y_m \) are all the remaining variables of a program (In Tempura, equivalently, \( x := e \overset{\text{def}}{=} (x' = e) \land (\text{more} \to \bigcirc y = y)(y \in V - \{x\}) \)). 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 \((x := e) \land (y := f)\) 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 framing operators being used in sequential, conjunctive and parallel manner. Within the PTL framework, Duan [16,17] brings forward a new assignment operator (\( \leftarrow \)) and an assignment flag (\( \text{af} \)), and then defines a framing operator \( \text{frame}(x) \). Thus, a framing technique based on an explicit framing operator is well formalized. Further, based on the proposed framing technique, Framed Tempura, an executable version of framed temporal logic programming language is developed. Now we rewrite \( \text{Program}(1) \) in Framed Tempura as

\[
\text{Program}(3) : \text{frame}(I) \land (I = 1) \land (J := 2) : (J := I + J)
\]

where \( \text{frame}(I) \) means that variable \( I \) always keeps its old value over an interval if no assignment to \( I \) is encountered. Thus, \( I \) can inherit the value from the initial state to the next, i.e., \( I \) is also assigned 1 at the second state. Therefore, \( J \) equals to 3 at the third state. The formal definitions of the framing operator are introduced in Section 2.

Another problem that must be dealt with in temporal logic programming is that of communication between concurrent processes. Some models of concurrency involve shared variables, and some involve asynchronous channels, e.g., CCS [8,9] and CSP [7,6]. In TLP languages, such as XYZ/E [31,32] and Tempura [16,17,26], such a communication is based on shared variables. To effect communication between parallel processes in a shared variables model, a synchronization construct \( \text{await}(c) \), or some equivalent, is required [2]. The meaning of \( \text{await}(c) \) is simple: it changes no variables, but waits until the condition \( c \) becomes true, at which point it terminates.

How could the \( \text{await}(c) \) construct be implemented in a TLP like Tempura? To start with, the \( \text{halt}(c) \) statement might play a role similar to that of \( \text{await}(c) \). However, \( \text{halt}(c) \) 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 un-repeated assignments, when we attempt to synchronize parallel components. For instance, the program:

\[
\text{Program(4)}: (l = 0) \land \text{halt}(l = 1) \land \text{len}(2)
\]

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

\[
\text{Program(5)}: (l = 0) \land \Box (\text{more} \rightarrow (\bigcirc l = 1)) \land \text{halt}(l = 1) \land \text{len}(2)
\]

is obviously unsatisfiable. As another example, the program:

\[
\text{Program(6)}: (l = 0) \land \text{halt}(l = 1)
\]

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

\[
\text{Program(7)}: (l = 0) \land \Box (\text{more} \rightarrow (\bigcirc l = 1)) \land \text{halt}(l = 1)
\]

would be waiting for \( l = 1 \) to become true. No process acting in parallel can set \( l \) to 1, since such an assignment would conflict with \( l = 0 \). So the program is also unsatisfiable. The difference between programs (5) and (7) is that the former specifies the length of the interval over which the program is executed is 2 but the latter leaves the interval unspecified.

Solving this problem can also be attempted by a suitable framing technique. The framing technique with an explicit framing operator, the minimal model semantics of framed programs, the communication and synchronization with construct await(c) with Framed Tempura can be found in \[16,17,18\]. In this paper, we focus on the operational semantics of Framed Tempura.

1.2. Operational semantics of TLP

Semantics of a program in imperative languages can be captured in an operational or denotational or axiomatic manner. In temporal logic programming, these semantics of a program can also be investigated. Since a temporal logic programming language, e.g., Tempura, is a subset of the corresponding logic, and the logic has its model theory and axiomatic system, the semantics of a program can be captured naturally by the model theory and axiomatic theory respectively. Of course, when executed, a program can also be interpreted in a more operational way.

In the Tempura community, several interpreters have been developed for years. Moszkowski developed the first Lisp version [26] for the original Tempura in 1983; Hale built a C version [10] for Tempura in 1986; Duan extended Tempura with framing and a new projection operator (\text{prj}) and developed an interpreter for Framed Tempura in Prolog [16,17] in 1992; recently, a research group in Xidian University has built an interpreter for Framed Tempura in C++ [23]; with this version, frame, await operators, a new projection operator, and pointers are all implemented.

Although these interpreters work well for their own purposes, however, no formal operational semantics has been given so far. This prevents us from carrying out the verification and analysis of programs in a rigorous way since the formal semantics is an essential prerequisite for formal verification.

Therefore, in this paper, we are motivated to investigate the operational semantics of temporal logic programs based on Framed Tempura. The main contribution of the paper is three-fold: (1) improved the reduction rules given in an abstract conference paper [22], and completed the proofs of all lemmas and theorems; (2) proved the consistency between the minimal model semantics and the operational semantics; (3) investigated the operational semantics for non-deterministic programs and formalized reduction rules for the programs.

The configuration of framed temporal logic programs is defined in terms of quadruple \((\text{prog}, \sigma_{i-1}, s_i, i)\); where \text{prog} is a program, \(\sigma_{i-1}\) is an interval over which \text{prog} is executed, and \(s_i\) denotes \(i\)th state; further, the evaluation rules for expressions are formalized; these rules enable us to evaluate values of expressions not only at the current state but also in the previous and next states.

To reduce the interval temporal logic programs, the reduction process can be divided into two phases: one for state reduction and the other for interval reduction. For the state reduction, we give semantic equivalence rules and transition rules within a state. Semantic equivalence rules ensure that any well-formed framed program can be reduced to its normal form; whereas transition rules are specified for two purposes: one for dealing with concurrent assignments, and the other for capturing minimal semantics by means of choosing variables with the minimal set of propositions. For the interval reduction, we pay attention to how a program is being executed from one state to the next. So a group of transition rules over intervals are formalized. Basically, this approach of studying operational semantics is mainly based on structural operational semantics (SOS) [4]. With this method, we need not consider any abstract machine models and memory for accessing programs. Further, the reduction of a framed program is managed by reduction rules and the semantics of programs is captured in a dynamic manner. In fact, when the reduction process terminates, interval \(\sigma\) is the minimal model of the program \(\text{prog}\). The consistency between the minimal model and the operational semantics of a terminable program is proved. Further, the consistency between the two semantics with infinite intervals is also proved by means of the Knasker–Tarski fixed-point theorem [5].

In the next section, projection temporal logic (PTL) and Framed Tempura are briefly introduced. Section 3 presents minimal model semantics for capturing the intended meaning of Framed Tempura; In Section 4, the operational semantics of framed
programs are investigated; to do so, a configuration for a program is defined; and semantic equivalence rules, transition rules within a state and transition rules over intervals are also specified to capture the meaning of programs. Further, some examples are given to illustrate how the reduction rules work. In addition, the consistency between the operational semantics and the minimal model semantics is proved in Section 5. Finally, conclusions are drawn in Section 6.

2. Preliminaries

Projection temporal logic (PTL) [16,17,19] is an extension of interval temporal logic (ITL) [26]. It is a first order temporal logic [30,2] with projection.

2.1. PTL

2.1.1. Syntax

Let $Prop$ be a countable set of propositions and $V$ be a countable set of typed variables. $B = \{true, false\}$ represents the Boolean domain. $D$ denotes all data needed by us including integers, lists, sets etc. The terms $e$ and formulas $p$ are inductively defined as follows:

$$e ::= v | \odot e | \Box e | \neg e | f(e_1, \ldots, e_n),$$

$$p ::= \pi | e_1 = e_2 | P(e_1, \ldots, e_n) | \neg p | p_1 \land p_2 | \exists v : p | \odot p | \Box p | (p_1, \ldots, p_m) \prj p | p^+,$$

where $v$ is a variable, and $\pi$ a proposition. In $f(e_1, \ldots, e_n)$ and $P(e_1, \ldots, e_n)$, $f$ is a function and $P$ a predicate. It is assumed that the types of the terms are compatible with those of the arguments of $f$ and $P$. In particular, when $n = 0, f$ is a constant term.

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

2.1.2. Semantics

1. States

A state $s$ is a pair of assignments ($l_{var}, l_{prop}$) which, for each variable $v \in V$ gives $s[v] \overset{\text{def}}{=} l_{var}[v]$, and for each proposition $\pi \in Prop$ gives $s[\pi] \overset{\text{def}}{=} l_{prop}[\pi]$. Each $l_{var}[v]$ is a value in a data domain $D$ or nil (undefined) and the total domain is denoted by $D^* = D \cup \{\text{nil}\}$, whereas $l_{prop}[\pi]$ is a truth value in $B$.

2. Intervals

An interval $\sigma = (s_0, s_1, \ldots)$ is a non-empty sequence of states, possibly infinite. The length of $\sigma$ is defined as follows.

$$|\sigma| = \begin{cases} n & \text{if } \sigma = (s_0, \ldots, s_n) \\ \omega & \text{if } \sigma \text{ is infinite} \end{cases}$$

The empty sequence is denoted by $\epsilon, |\epsilon| = -1$. To have a uniform notation for both finite and infinite intervals, we will use extended integers as indices. That is, we consider set $N_0$ of non-negative integers and $\omega$.

$$N_\omega = N_0 \cup \{\omega\}$$

and extend the comparison operators: $=, <, \leq, \geq$ to $N_\omega$ by considering $\omega = \omega$, and for all $i \in N_0, i < \omega$. Moreover, we define $\leq$ as $\leq - \{(\omega, \omega)\}$. To simplify definitions, we will denote $\sigma$ as $(s_0, \ldots, s_{|\sigma|})$, where $s_{|\sigma|}$ is undefined if $\sigma$ is infinite. With such a notation, $\sigma(i, j) (0 \leq i \leq j \leq |\sigma|)$ denotes the sub-interval $(s_i, s_j)$.

3. Interpretations

An interpretation for a PTL term or formula is a tuple $I \overset{\text{def}}{=} (\pi, i, k, j)$, where $\pi$ is an interval, $i, k$ are integers, and $j$ an integer or $\omega$ such that $i \leq k \leq j \leq |\sigma|$. Intuitively, we use $(\pi, i, k, j)$ to mean that a term or formula is interpreted over a subinterval $\sigma(i, j)$ with the current state being $s_k$.

For every term $e$, the evaluation of $e$ relative to $I$ is defined as $I[e]$, by induction on the structure of the term, as shown in Fig. 1. We use $I_{var}$ and $I_{prop}$ to denote the state interpretation at state $s_k$. The pair $x_i : e_i$ means that $I_{var}[x_i] = e_i$, where $x_i \in V, e_i \in D$. Each $m$-place function symbol $f$ has an interpretation $I[f]$ which is a function mapping $m$ elements in $D^m$ to a single value in $D$. Interpretations of predicate symbols are similar but map to truth values. We assume that $I$ gives standard interpretations to operators such as $+,-,\ast$, and $<,>,\leq,\geq,=,\text{etc}$.

Projection construct $(p_1, \ldots, p_m) \prj q$, which can be thought of as a combination of the parallel and the projection operators presented in [26]. Intuitively, it means that $q$ is executed in parallel with $p_1: \cdots: p_m$ ("chop" is a sequential operator) over an interval obtained by taking the endpoints of the intervals over which $p_1, \ldots, p_m$ are executed. The projection 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 processes \(p_1, \ldots, p_m\) and process \(q\) may terminate at different time points.

To define the semantics of the projection construct we need an auxiliary operator. Let \(\sigma = (s_0, s_1, \ldots)\) be an interval and \(r_1, \ldots, r_h\) be integers \((h \geq 1)\) such that \(0 \leq r_1 \leq \cdots \leq r_h \leq |\sigma|\).

\[
\sigma \downarrow (r_1, \ldots, r_h) \overset{\text{def}}{=} (s_{r_1}, s_{r_2}, \ldots, s_{r_h}).
\]

The projection of \(\sigma\) onto \(r_1, \ldots, r_h\) is the interval (called projected interval) 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,

\[
(s_0, s_1, s_2, s_3) \downarrow (0, 2, 2, 3) = (s_0, s_2, s_3).
\]

As depicted in Fig. 2, the projected interval \((s_0, s_2, s_3)\) can be obtained by using \(\downarrow\) operator to take the endpoints of each process empty, \(len_2\), empty, \(empty\), \(empty\), \(len_1\).

For a variable \(v\), we will denote \(\sigma' \equiv \sigma\) whenever \(\sigma'\) is an interval which is the same as \(\sigma\) except that different values can be assigned to \(v\), and we call \(\sigma\) and \(\sigma'\) are \(v\)-equivalent. The operator concatenation \((\cdot)\) is defined as follows.

\[
\sigma \cdot \sigma' = \begin{cases} 
\sigma' & \text{if } |\sigma| = \omega \text{ or } \sigma' = \epsilon, \\
\sigma & \text{if } \sigma = \epsilon, \\
(s_0, s_1, s_{i+1}, \ldots) & \text{if } \sigma = (s_0, \ldots, s_i) \text{ and } \sigma' = (s_{i+1}, \ldots). 
\end{cases}
\]

The meaning of formulas is given by the satisfaction relation, \(\models\), which is inductively defined in Fig. 3.

A formula \(p\) is said to be satisfied, denoted by \(\sigma \models p\), if \((\sigma, 0, 0, |\sigma|) \models p\); a formula \(p\) is called satisfiable if \(\sigma \models p\) for some \(\sigma\); and \(p\) is called valid, denoted \(\models p\), if \(\sigma \models p\) for all \(\sigma\); sometimes, we denote \(\models p \rightarrow q\) (resp. \(\models p \rightarrow q\)) for \(p \models q\) (resp. \(p \rightarrow q\)) and \(\models \bigcirc(p \leftrightarrow q)\) (resp. \(\models \bigcirc(p \leftrightarrow q)\)) by \(p \equiv q\) (resp. \(p \equiv q\)). The former is called weak equivalence (resp. weak implication) and the latter strong equivalence (resp. strong implication).

**Example 1.** Evaluate the formula \(p \overset{\text{def}}{=} x = 3 \land \bigcirc y = x^* 3 + 1\) according to the interval \(\sigma = (l_{\text{var}}^0, l_{\text{prop}}^0, \ldots)\) where \(l_{\text{var}}^0 = \{x : 3\}\) and \(l_{\text{var}}^1 = \{y : 10\}\). We do not need to specify \(l_{\text{prop}}^1\) in this example.

\[
\begin{align*}
(\sigma, 0, 0, |\sigma|) & \models x = 3 \land \bigcirc y = x^* 3 + 1 \\
\iff (\sigma, 0, 0, |\sigma|) & \models x = 3 \land (\sigma, 0, 0, |\sigma|) \models \bigcirc y = x^* 3 + 1 \\
\iff (\sigma, 0, 0, |\sigma|) & \models [x] = (\sigma, 0, 0, |\sigma|)[3] \text{ and } (\sigma, 0, 0, |\sigma|)[y] = (\sigma, 0, 0, |\sigma|)[3] + (\sigma, 0, 0, |\sigma|)[1] \\
\iff s_0[x] & = 3 \text{ and } (\sigma, 0, 1, |\sigma|)[y] = s_0[x] 3 + 1 
\end{align*}
\]

**Fig. 1.** Interpretation of PTL terms.

**Fig. 2.** A projected interval.
\[ \iff I_0[\{x\} = 3 \text{ and } s_1[\{y\} = s_0[\{x\}]^3 + 1 ] \]
\[ \iff 3 = 3 \text{ and } I_1[\{y\} = I_0[\{x\}]^3 + 1 ] \]
\[ \iff 3 = 3 \text{ and } 10 = 3^3 + 1 \]
\[ \iff \text{true} \]

Therefore, \( \sigma \models p \).

In the process of evaluation, each step proceeds in virtue of the semantics of PTL terms and formulas. When the value of a variable \( x_i \) is irrelevant, the pair \( x_i : e_i \) may not be shown. Similarly, \( I_{prop} \) need not be specified if propositions are not involved in formulas.

### 2.1.3. Derived formulas and logic laws

Fig. 4 shows us some useful derived formulas derived from elementary PTL formulas. empty represents the final state and first expresses the left end state of an interval; more specifies that the current state is a non-final state; \( \Diamond p \) (namely sometimes \( p \)) means that \( p \) holds eventually in the future including the current state; \( 
\Box p \) (namely always \( p \)) represents that \( p \) holds always in the future from now on; \( \bigodot p \) (weak next) tells us that either the current state is the final one or \( p \) holds at the next state of the present interval; \( \text{Proj}(p_1, \ldots, p_m) \) represents a sequential computation of \( p_1, \ldots, p_m \) since the projected interval is a singleton; and \( p \land q \) (sometimes \( p \) chop \( q \)) represents a computation of \( p \) followed by \( q \), and the intervals for \( p \) and \( q \) share a common state. That is, \( p \) holds from now until some point in future and from that time point \( q \) holds. Note that \( p \land q \) is a strong chop which always requires that \( p \) be true on some finite subinterval.

\( \text{len}(n) \) specifies the distance \( n \) from the current state to the final state of an interval; skip means that the length of the interval is one unit of time. \( \text{fin}(p) \) is true as long as \( p \) is true at the final state while \( \text{keep}(p) \) is true if \( p \) is true at every state but the final one. The formula \( \text{halt}(p) \) holds if and only if formula \( p \) is true at the final state. Further, \( \text{if } b \text{ then } p \text{ else } q \), while \( b \text{ do } p \) and \( p \parallel q \) can also be defined by our underlying logic.

### Fig. 3. Interpretation of PTL formulas.

### Fig. 4. Derived PTL formulas.
Table 1  PTL laws: x is a static or dynamic variable and w is a state formula

<table>
<thead>
<tr>
<th>Law</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Law1</td>
<td>$\diamond p \equiv \diamond p \land \text{more}$</td>
</tr>
<tr>
<td>Law2</td>
<td>$\square p \supset \text{more}$</td>
</tr>
<tr>
<td>Law3</td>
<td>$\square(p \land q) \equiv \square p \land \square q$</td>
</tr>
<tr>
<td>Law4</td>
<td>$\diamond(p \lor q) \equiv \diamond p \lor \diamond q$</td>
</tr>
<tr>
<td>Law5</td>
<td>$\square(\exists x : p) \equiv \exists x : \diamond p$</td>
</tr>
<tr>
<td>Law6</td>
<td>$\square p \equiv p \land \square \diamond p$</td>
</tr>
<tr>
<td>Law7</td>
<td>$\square p \land \emptyset \equiv p \land \emptyset$</td>
</tr>
<tr>
<td>Law8</td>
<td>$\square p \land \text{more} \equiv p \land \square \diamond p$</td>
</tr>
<tr>
<td>Law9</td>
<td>$w ; (p \lor q) \equiv (w ; p) \lor (w ; q)$</td>
</tr>
<tr>
<td>Law10</td>
<td>$(p \lor q) ; w \equiv (p ; w) \lor (q ; w)$</td>
</tr>
<tr>
<td>Law11</td>
<td>$(w \land p) ; q \equiv w \land (p ; q)$</td>
</tr>
<tr>
<td>Law12</td>
<td>$\diamond p ; q \equiv \diamond (p ; q)$</td>
</tr>
<tr>
<td>Law13</td>
<td>$\text{empty} ; q \equiv q$</td>
</tr>
<tr>
<td>Law14</td>
<td>$(p_1, \ldots, p_m) \text{prj empty} \equiv p_1 \land \cdots \land p_m$</td>
</tr>
<tr>
<td>Law15</td>
<td>$\text{empty prj q} \equiv q$</td>
</tr>
<tr>
<td>Law16</td>
<td>$(p_1, \ldots, w \land \text{empty}, p_{i+1}, \ldots, p_m) \text{prj q} \equiv (p_1, \ldots, w \land p_{i+1}, \ldots, p_m) \text{prj q}$</td>
</tr>
<tr>
<td>Law17</td>
<td>$(w \land p_1, p_2, \ldots, p_m) \text{prj q} \equiv w \land (p_1, \ldots, p_m) \text{prj q}$</td>
</tr>
<tr>
<td>Law18</td>
<td>$(p_1, \ldots, p_m) \text{prj (w \land q)} \equiv w \land (p_1, \ldots, p_m) \text{prj q}$</td>
</tr>
<tr>
<td>Law19</td>
<td>$(\square p_1, \ldots, p_m) \text{prj} \square q \equiv \square (p_1 \land (p_2, \ldots, p_m) \text{prj q})$</td>
</tr>
<tr>
<td>Law20</td>
<td>$\exists x : p(x) \equiv \exists y : p(y)$</td>
</tr>
<tr>
<td>Law21</td>
<td>$\exists x : (p(x) \lor q(x)) \equiv \exists x : p(x) \lor \exists x : q(x)$</td>
</tr>
</tbody>
</table>

A PTL formula $p$ is left end closed (lec-formula) if $(\sigma, i, k, j) \models p \iff (\sigma, i, k, j) \models p$, for any interpretation $(\sigma, i, k, j)$. Similarly, $q$ is right end closed (rec-formula) if $(\sigma, i, k, j) \models q \iff (\sigma, i, k, j) \models q$ for any interpretation $(\sigma, i, k, j)$. 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 $s_k$, then $p$ does not refer to any state to the left of $s_k$, and similarly for a rec-formula. For instance, $\square(\text{more} \to \diamond(p_1 \leftrightarrow \diamond p_2))$ is a lec-formula, and first is a rec-formula. If $p$ is a state formula, then $p$ is a lec-formula as well as a rec-formula. A formula is called a terminal formula if $p \equiv p \land \text{empty}$. A formula $p$ is non-local if $\sigma \models p$ implies $\sigma \models p \land \sigma(\leq 1)$.

**Theorem 1.**  The logic laws in Table 1 hold.

The proof of these logic laws can be found in [16,17,18]. They are useful for reasoning about programs.

2.1. Replacement of variables

To reduce a program, we often use the substitution of terms for terms or variables. It is assumed that a static variable remains the same over an interval whereas a dynamic variable can have different values at different states. A formula (or term) is called static if it does not refer to any dynamic variable. However, the substitution is required to be compatible or admissible. In the following, we define these concepts in details.

**Definition 1.**  Let $\tau$ be a formula or term. If $t$ is a term and $x$ a variable used in $\tau$, then $\tau[t/x]$ denotes the result of simultaneous replacement of all free occurrences of $x$ by $t$ in $\tau$. The replacement is called compatible if either $x$ and $t$ are static or $x$ is dynamic.

In the following, we write $\tau(x)$ to imply that $\tau$ has one or more occurrences of variable $x$ and that there is no quantification over $x$.

**Definition 2.**  The replacement $\tau[t/x]$ is called admissible for $\tau(x)$ if it is compatible and none of the variables appearing in $t$ is quantified in $\tau$. We also say that $t$ is admissible for $x$ in $\tau(x)$ and write $\tau(t)$ to denote $\tau[t/x]$.

**Theorem 2.**  For a state term $e(x)$ and terms $t_1, t_2$ that are admissible for $x$ in $e(x)$, we have

$$t_1 = t_2 \supset e(t_1) = e(t_2)$$

**Proof.**  The proof of Theorem 2 can be found in [16,17] and the Appendix.
We can further get \( x(t) \) being a state term is necessary for Theorem 2. For instance, taking \( e(x) \) to be \( \bigcirc x \), and \( t_1, t_2 \) to be \( x, y \) respectively, \( x = y \) does not imply \( \bigcirc x = \bigcirc y \). We generalize \( e(x) \) to a function \( f \) with arity \( m \geq 0 \) and has the following consequence.

**Corollary 3.** For a state term \( f(x_1, \ldots, x_m) \), and state terms \( t_1, \ldots, t_m, e_1, \ldots, e_m \) that are admissible for \( x_1, \ldots, x_m \) in \( f(x_1, \ldots, x_m) \), we have

\[
(t_1 = e_1 \land \cdots \land t_m = e_m) \implies (f(t_1, \ldots, t_m) = f(e_1, \ldots, e_m))
\]

**Proof.** The proof of Corollary 3 can be found in [16,17] and the Appendix. \( \square \)

Theorem 2 and Corollary 3 are useful for reducing terms like equations. As an example, taking the function \( f \) as a binary operation such as +, −, *, the equation \( x - y = 2 \land x + y = 0 \) can be reduced as follows:

\[
\begin{align*}
  x - y &= 2 \land x + y = 0 \\
  \implies (x - y) + (x + y) &= 2 + 0 \\
  \implies 2 \cdot x &= 2 \\
  \implies x &= 1
\end{align*}
\]

We can further get \( y = -1 \). Similarly, it is also applicable for reducing an equation with an unary operation such as square \( (x^2) \) and square root \( (\sqrt{x}) \). For example

\[
x^2 = 4 \\
\implies x = 2 \text{ or } x = -2
\]

2.2. Framed Tempura

The programming language we use is Framed Tempura, a subset of PTL, which augments Tempura with infinite models, framing and a new projection \((\text{prj})\) operator. Further, the variables within a program can refer to their previous values.

2.2.1. Syntax

As an executed language of PTL, Framed Tempura consists of expressions and statements. Expressions can be regarded as PTL terms, and statements as PTL formulas.

1. Expressions

   The permissible arithmetic expression \( e \) in this paper is confined to constants, variables, restricted temporal expressions \( \bigcirc x \) (next expression) and \( \boxdot x \) (previous expression)

\[
e ::= n \mid x \mid \bigcirc x \mid \boxdot x \mid e_0 \text{ op } e_1
\]

where \( n \) is an integer, \( x \) a variable and \( \text{op} ::= + \mid - \mid * \mid / \mid \text{mod} \).

The Boolean expression \( b \) is defined by the following grammar.

\[
b ::= \text{true} \mid \text{false} \mid \neg b \mid b_0 \land b_1 \mid e_0 = e_1 \mid e_0 < e_1
\]

(2.2)

Boolean expressions are built on top of arithmetic expressions since the relational expressions in (2.2) involve arithmetic expressions.

2. Statements

There are 11 elementary statements in Framed Tempura. Five of them, \( x = e, \bigcirc p, p \land q, \exists v: p, \text{ and } (p_1, \ldots, p_m) \text{ prj } q \) are basic formulas in PTL; whereas the other six, \( \text{if } b \text{ then } p \text{ else } q \), \text{while } b \text{ do } p, p \parallel q, p : q, \emptyset \) empty and \( \boxdot p \) are derived formulas taken from PTL. For the purpose of inductively proving a property of programs, the assignment \( x = e \) and empty can be thought of as basic statements and the others can be treated as composite statements. Conventionally, \( x \) denotes a variable, \( e \) stands for an arbitrary arithmetic expression, \( b \) a Boolean expression, and \( p, p_1, \ldots, p_m \) and \( q \) programs.

- Assignment (Unification) : \( x = e \)
- Conjunction statement : \( p \land q \)
- Conditional statement : \( \text{if } b \text{ then } p \text{ else } q \)
- Existential quantification : \( \exists v : p \)
- Next statement : \( \bigcirc p \)
- Always statement : \( \Box p \)
- Sequential statement : \( p : q \)
- While statement : \( \text{while } b \text{ do } p \)
- Parallel statement : \( p \parallel q \)
∃x renaming variable x sequences of clocks (states) running according to two different time scales: one is a local state sequence, over which different time scales. Consider the following formulas:

As already mentioned, projection can be thought of as a special parallel computation which is executed on two

the temporal semantics, the concept of local variable is not effective.

p ... q first evaluates the Boolean expression; if b is true, then the process p is executed, otherwise q is executed. 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. One extreme case is the statement while b do empty, which is simply equivalent to ¬b ∧ empty.

The conjunction statement p ∧ q means that the processes p and q are executed concurrently and they share all the states and variables during the mutual execution. Further, the parallel construction p||q presents another concurrent computation manner. The difference between p||q and p ∧ q is that the former allows both processes p and q to be able to specify their own intervals while the latter does not. E.g., len(2)||len(3) holds but len(2) ∧ len(3) is obviously false. The existential quantification ∃x : p intends to hide the variable x within the process p. It may permit a process p to use a local variable x. However, within the temporal semantics, the concept of local variable is not effective.

In programming language terms, the interpretation of (p₁,...,pₘ) 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 p₁,...,pₘ are executed, and the other is a global state sequence over which q is executed in parallel with the sequence of processes p₁,...,pₘ as follows (see Fig. 5). We start q and p₁ at the first global state and p₁ is executed over a sequence of local states until its termination. Then (the remaining part of) q and p₂ are executed at the second global state, and p₂ is executed over a sequence of local states until its termination, and so on. Although q and p₁ start at the same time, p₁,...,pₘ and q may terminate at different time points. If q terminates before some pₘ, then, subsequently, pₘ,...,p₁ are executed sequentially. If p₁,...,pₘ are finished before q, then the execution of q is continued until its termination.

Example 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:

p₁ = def len(2) ∧ □(more → ((□i) = i + 2))

p₂ = def len(4) ∧ □(more → ((□i) = i + 3))

p₃ = def len(6) ∧ □(more → ((□i) = i + 4))

q = def len(4) ∧ (i = 2) ∧ (j = 0) ∧ □(more → ((□j) = j + i)).

Then executing (p₁,p₂,p₃) prj q yields the result shown in Fig. 6.

We use a renaming method to reduce a program with existential quantification. Given a formula ∃x : p(x) with a bound variable x, we can remove the existential quantification (∃x) from ∃x : p(x) to obtain a formula p(y) with a free variable y by renaming x as y. To do so, we require that:

• y do not occur (free or bound) in the whole program such as (q ∧ ∃x : p(x));
• y and x both be either dynamic or static;
• y substitutes for x only within the bound scope of x in \( \exists x : p(x) \).

In this case, we call \( p(y) \) a renamed formula of \( \exists x : p(x) \). Now we discuss some facts regarding renamed formulas.

**Lemma 4.** Let \( p(y) \) be a renamed formula of \( \exists x : p(x) \). Then, \( \exists x : p(x) \) is satisfiable if and only if \( p(y) \) is satisfiable. Furthermore, any model of \( p(y) \) is a model of \( \exists x : p(x) \).

**Proof.** By LEXISTS, given a model \( \sigma \), the following is true:

\[
\sigma \models \exists y : p(y) \text{ iff there exists } \sigma' \models \forall y, \sigma' \models p(y).
\]

Thus, if \( \sigma \models \exists y : p(y) \) then there is a \( \sigma' \), \( \sigma' \models p(y) \). Conversely, for a model \( \sigma \), if \( \sigma \models p(y) \), then \( \sigma \models \exists y : p(y) \) because \( \sigma \) is trivially \( y \)-equivalent to itself. Thus, \( \exists y : p(y) \) is satisfiable iff \( p(y) \) is satisfiable; and any model of \( p(y) \) is a model of \( \exists y : p(y) \).

Moreover, by Law 20 in Table 1, \( \exists y : p(y) \equiv \exists x : p(x) \). Hence, Lemma 4 is true. \( \square \)

The importance of Lemma 4 is that it guarantees the soundness of the renaming method for reducing programs involving existential quantifications. From Lemma 4, it is clear that \( \exists x : p(x) \) and \( p(y) \) are equivalent in satisfiability. To check whether or not \( \exists x : p(x) \) is satisfiable amounts to checking whether or not \( p(y) \) is satisfiable. Once we find a model for \( p(y) \), this model is also a model for \( \exists x : p(x) \). Therefore, to reduce \( \exists x : p(x) \), it is sufficient to reduce \( p(y) \).

One may object to the renaming method by saying that it offends the original spirit by using local variables in a program. However, we are investigating the temporal semantics of a program under the logic model theory. To interpret a formula with a given assignment, we make use of a flag called the assignment flag, denoted by predicate \( \text{af} \).

Definition 3 tells us that the models of \( \exists x : p(x) \) and \( p(y) \) are the same ignoring variable \( x \). Thus, we denote such the special equivalence relation between programs as \( \equiv \), which is the counterpart of the equivalence relation \( \equiv \) over the intervals. Notice that, by Law 20 in Table 1, \( \exists x : p(x) \equiv \exists y : p(y) \), and by Definition 3, \( \exists y : p(y) \equiv p(y) \); so we say that \( \exists x : p(x) \equiv p(y) \) if \( p(y) \) is a renamed program of \( \exists x : p(x) \), and the model of \( \exists x : p(x) \) can be obtained by hiding variable \( y \) in the model of \( p(y) \).

**Example 3.** \( \text{PROG I} \overset{\text{def}}{=} x = 1 \land y = 2 \land \exists x : (x = 2 \land y := 3x) \land x := x + y \)

\[
\begin{align*}
\bar{\exists} &\quad \bar{x} = 1 \land y = 2 \land (z = 2 \land y := 3z) \land x := x + 2 \\
\equiv &\quad x = 1 \land y = 2 \land z = 2 \land (x = 3 \land y = 6 \land \text{empty}) \quad (x := e \overset{\text{def}}{=} \bar{x} = e \land \text{all}(1))
\end{align*}
\]

In this program, we use variable \( z \), which does not appear in the whole program, to replace local variable \( x \), and denote the renamed program by \text{PROG II}. Thus, we have \( \text{PROG I} \bar{\exists} \overset{\text{def}}{=} \text{PROG II} \). Moreover, by Definition 3, the model of \text{PROG I} can be obtained by hiding variable \( z \) in the model of \text{PROG II}. Therefore, the model of \text{PROG I} is \( \sigma = \langle s_0, s_1 \rangle = \langle ([x : 1, y : 2]), ([x : 3, y : 6]) \rangle \).

### 2.2.2. Framing

The introduction of a framing technique to temporal logic programming is motivated by both practical and theoretical considerations: improving the efficiency of a program and synchronizing communication for parallel processes. The meaning of the framing operator, denoted by frame, can be stated as follows: frame(x) means that variable x always keeps its old value over an interval if no assignment to x is encountered.

The crux of the above technique is how to perceive the assignments of values to variables. To identify an occurrence of an assignment to a variable, say \( x \), we make use of a flag called the assignment flag, denoted by predicate \( \text{af}(x) \); it is true whenever an assignment of a value to \( x \) is encountered, and false otherwise. Note that it may not be used freely in a program but only concerned with an assignment and a framing operator. \( \text{af}(x) \) is easy to understand but difficult to formalize in a logic framework. The problem is that a program provides only positive information, that is, some explicit assignments from which we know those variables assigned explicitly within the program. However, what we need is negative information, i.e., those variables which do not encounter assignments at the current state.

For this purpose, we first define a new assignment which is required by framing; then we define framing operators; and finally, we present a minimal model-based approach for framing.

Let \( S_p = \{x_1, \ldots, x_n\} \subseteq V \) be a set of state (dynamic) variables within a program \( p \) and \( \Phi_p = \{p_{x_1}, \ldots, p_{x_n}\} \) be the set of propositions associated with state variables. We assume that a program \( p \) does not involve propositions other than \( \Phi_p \). A new assignment called positive immediate assignment is defined as
where \( p_x \) is an atomic proposition connected with variable \( x \) and cannot be used for other purpose. To identify an occurrence of an assignment to a variable, say \( x \), we make use of a flag called the assignment flag, denoted by a predicate \( \text{af}(x) \) which is defined as

\[
\text{af}(x) \iff p_x
\]

\( \text{af}(x) \) 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, \( \text{af}(x) \) should be true. Conversely, when \( \text{af}(x) \) is true, such an assignment should have been perceived in the execution of the program. As expected, when \( x \equiv e \) is encountered, \( p_x \) is set to true, hence \( \text{af}(x) \) is true; whereas if no assignment to \( x \) takes place, \( p_x \) is unspecified. In this case, we will use a minimal model to force it to be false.

Armed with the assignment flag, we can define state framing and interval framing operators. Intuitively, when a variable is framed at a state, its value remains unchanged if no assignment is encountered at that state. A variable is framed over an interval if it is framed at every state over the interval. We formalize this idea in Definition 4.

**Definition 4 (Looking back framing).**

\[
\text{lbf}(x) \equiv \neg \text{af}(x) \rightarrow \exists b: (\square x = b \land x = b)
\]

\[
\text{frame}(x) \equiv \square (\text{more} \rightarrow \bigcirc \text{lbf}(x))
\]

where \( b \) is a static variable.

A dynamic variable \( x \) is said to be framed in program \( p \) if \( \text{frame}(x) \) or \( \text{lbf}(x) \) is contained in \( p \). In this framing environment, framed variables and non-framed variables can be mixed. So, a framed program can inductively be defined as follows.

- \( x = e, x \equiv e \) and \( \text{empty} \) are framed programs.
- \( \text{lbf}(x) \) and \( \text{frame}(x) \) are framed programs.
- If \( p_1, \ldots, p_m, p, q \) are framed programs, so are the following:
  - \( p \land q \), if \( b \) then \( p \) else \( q \), while \( b \) do \( p \), \( \exists x : p, \bigcirc p, \square p, p : q, p \parallel q, (p_1, \ldots, p_m) \text{prj} q \), and \( \text{empty} \)

**Theorem 5.** The following laws hold:

1. \( x = e \equiv (p_x \land x = e) \lor (\neg p_x \land x = e) \).
2. \( \text{lbf}(x) \equiv p_x \lor (\neg p_x \land (x = \square x)) \).
3. \( \text{lbf}(x) \land x = e \equiv x \equiv e \land x \neq e \).
4. \( \text{lbf}(x) \land x \equiv e \equiv x \equiv e \).

The equivalent laws in Theorem 5 are useful to reduce assignment statements. In Table 2, some interesting algebraic properties regarding framing operators, such as idempotent, distributive, absorptive, and equivalent laws are given.

Let us recall that the first order logic has the following properties:

1. reflexivity: \([w_1, \ldots, w_n, w] \vdash w\).
2. monotonicity: if \([w_1, \ldots, w_n] \vdash w\) then \([w_1, \ldots, w_n, u] \vdash w\).
3. transitivity: if \([w_1, \ldots, w_n] \vdash u\) and \([w_1, \ldots, w_n, u] \vdash w\), then \([w_1, \ldots, w_n] \vdash w\).

In the expressions above, \( w_1, \ldots, w_n, u, w \) represent formulas of logical language. From above (2), the first order logic is monotonic. That is, adding a formula to a theory has the effect of strictly increasing the set of formulas that can be inferred. However, an important fact we claim is that adding framing operators to PTL makes the underlying semantics a radical shift from monotonicity to non-monotonicity. We conclude it in Fact 6.

**Fact 6.** A logic involving framing operators is non-monotonic. That is, given formulas, \( w_1, \ldots, w_n, u, w \), it may happen that \( w_1 \land \cdots \land w_n \rightarrow w \), but \( w_1 \land \cdots \land w_n \land u \rightarrow w \) does not.

**Example 4.** Consider a framed program: \( x = 1 \land \text{frame}(x) \land \text{len}(1) \). It is obvious that

\[
x = 1 \land \text{frame}(x) \land \text{len}(1) \lor \square x = 1 (*)
\]

holds. However, adding the assignment \( \bigcirc (x = 2) \) to the left of implication (*), we have,

\[
x = 1 \land \text{frame}(x) \land \text{len}(1) \land \bigcirc (x = 2) \lor \square x = 2
\]

As the example shows, \( \{x = 1, \text{frame}(x), \text{len}(1)\} \) implies \( \bigcirc x = 1 \) but, by adding \( \bigcirc (x = 2) \), \( \{x = 1, \text{frame}(x), \text{len}(1), \bigcirc (x = 2)\} \) implies \( \bigcirc x = 2 \) rather than \( \bigcirc x = 1 \). Therefore, framing operators destroy monotonicity in our underlying logic.
3. Minimal model semantics of framed programs

We use canonical models to capture the semantics of non-framed programs in Tempura. These models are of the kind described by Bidoit in [37]. In the following, we briefly introduce the canonical models. Suppose that a program $p$ contains a finite set $S_P$ of variables and a finite set $\Phi_P$ of propositions. A canonical interpretation on propositions is a subset $I_{\text{prop}} \subseteq \Phi_P$. Implicitly, propositions not in $I_{\text{prop}}$ are false. Let $\sigma = \langle I_{\text{var}}, I_{\text{prop}}, \ldots \rangle$ be a model. We denote the sequence of interpretation on propositions of $\sigma$ by $I_{\text{prop}} = \langle I_{\text{prop}}^0, \ldots \rangle$. If there exists a model $\sigma$ with $I_{\text{prop}}$ 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 $\sigma \models P$, and call $I_{\text{prop}}$ a canonical interpretation sequence (on propositions) of $P$. If $\sigma \models P$, for all $\sigma$ with canonical $I_{\text{prop}}$, then $P$ is valid under the canonical interpretation on propositions. We denote this by $\models P$. 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.

**Definition 5.** Let $p$ be a framed program, and $\Sigma_p = \{\sigma | \sigma \models_p p\}$. Let $\sigma_1, \sigma_2 \in \Sigma_p$. We define
- $\sigma_{\text{prop}} \subseteq \sigma'_{\text{prop}}$ iff $|\sigma| = |\sigma'|$ and $I_i^{\text{prop}} \subseteq I_i'^{\text{prop}}$, for all $0 \leq i \leq |\sigma|$.
- $\sigma \subseteq \sigma'$ iff $\sigma_{\text{prop}} \subseteq \sigma'_{\text{prop}}$.
- $\sigma \sqsubset \sigma'$ iff $\sigma \subseteq \sigma'$ and $\sigma' \not\subseteq \sigma$.
- $\sigma_1 \cong \sigma_2$ iff $\sigma_1 \sqsubseteq \sigma_2$ and $\sigma_2 \sqsubseteq \sigma_1$.

For instance, $\langle (\emptyset, \{x:1\}) \rangle \sqsubset \langle (\{p_{x}\}, \emptyset) \rangle$, $\langle (\{p_{x}\}, \{x:1\}) \rangle \cong \langle (\{p_{x}\}, \{x:2\}) \rangle$.

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.

**Definition 6.** Let $p$ be a framed program, and $I = (\sigma, i, k, j)$ be a canonical interpretation. Then the minimal satisfaction relation $\models_m p$ is defined as

$(\sigma, i, k, j) \models_m p$ iff $(\sigma, i, k, j) \models_p p$ and there is no $\sigma'$ such that $\sigma' \sqsubset \sigma$ and $(\sigma', i, k, j) \models_m p$.

A minimal model of a program $p$ is a canonical model $\sigma$ such that $(\sigma, 0, 0, |\sigma|) \models_m p$. We denote this by $\sigma \models_m p$. Moreover, the equivalence relations $\equiv_m$ and $\cong_m$ as well as the strong implication relation $\models_m$ can be defined similarly as the relations $\equiv$ and $\cong$.

The intended meaning of a program $p$ is captured by its minimal model. For instance, if $p$ is $x \leftarrow 1 \land \text{frame}(x) \land \text{len}(1)$ then under the minimal model, $x = 1$ is defined at both state $s_0$ and $s_1$, this is the intended meaning of $p$. However, within only the canonical model, $p_x$ is unspecified at state $s_1$, so it could be true at $s_1$. This causes $x$ to be unspecified at state $s_1$. Therefore, $x_1$ could be any value from its domain.

**Definition 7.** A framed program $q$ is in normal form if

$q \overset{\text{def}}{=} \left( \bigvee_{i=1}^{k} q_{ei} \land \text{empty} \right) \lor \left( \bigvee_{j=1}^{h} q_{ej} \land \text{o} q_{fj} \right)$

where $k + h \geq 1$ and the following hold:
- $q_{fj}$ is an internal program, that is, one in which variables may refer to the previous states but not beyond the first state of the current interval over which the program is executed.
- each $q_{ej}$ and $q_{ei}$ is either true or a state formula of the form $p_1 \land \ldots \land p_m$ ($m \geq 1$) such that each $p_1(1 \leq 1 \leq m)$ is either $(x = e)$ with $e \in D$, $x \in V$, or $p_x$, or $\neg p_x$.

We simply write $q_{ei} \land \text{empty}$ instead of $\bigvee_{i=1}^{k} q_{ei} \land \text{empty}$. Usually, if $q$ terminates at the current state it is reduced to $q_e \land \text{empty}$; otherwise it is reduced to $q_{ej} \land \text{o} q_{fj}$. Also, we call conjunctions, $q_{ei} \land \text{empty}$, $q_{ej} \land \text{o} q_{fj}$, basic products; the former is called terminal product whereas the latter is called future products. Further, we call $q_{ei}$ and $q_{ej}$ present components, $\text{o} q_{fj}$ future components of basic products. A key conclusion is that any framed program can be reduced to its normal

<table>
<thead>
<tr>
<th>Law</th>
<th>Idempotent</th>
<th>$\text{frame}(x) \land \text{frame}(x)$</th>
<th>$\text{frame}(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Law 22</td>
<td>Distributive</td>
<td>$\text{frame}(x) \land p \lor \text{frame}(x) \land q$</td>
<td>$\text{frame}(x) \land (p \lor q)$</td>
</tr>
<tr>
<td>Law 24</td>
<td>Absorptive</td>
<td>$\text{frame}(x) \land (\text{frame}(x) \land p \lor q)$</td>
<td>$\text{frame}(x) \land (p \lor q)$</td>
</tr>
<tr>
<td>Law 25</td>
<td>Equivalent</td>
<td>$\text{frame}(x) \land \text{more}$</td>
<td>$\text{empty}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Law</th>
<th>Idempotent</th>
<th>$\text{frame}(x) \land \text{frame}(x)$</th>
<th>$\text{frame}(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Law 22</td>
<td>Distributive</td>
<td>$\text{frame}(x) \land p \lor \text{frame}(x) \land q$</td>
<td>$\text{frame}(x) \land (p \lor q)$</td>
</tr>
<tr>
<td>Law 24</td>
<td>Absorptive</td>
<td>$\text{frame}(x) \land (\text{frame}(x) \land p \lor q)$</td>
<td>$\text{frame}(x) \land (p \lor q)$</td>
</tr>
<tr>
<td>Law 25</td>
<td>Equivalent</td>
<td>$\text{frame}(x) \land \text{more}$</td>
<td>$\text{empty}$</td>
</tr>
</tbody>
</table>
form. Therefore, one way to execute programs in Framed Tempura is to transform them logically equivalent to their normal forms.

**Theorem 7.** For any framed program \( \text{prog} \) there is a framed program \( \text{prog}' \) in the normal form such that \( \text{prog} \equiv \text{prog}' \)

**Proof.** The proof of Theorem 7 can be found in [16,17,18]. \( \square \)

**Example 5.** Consider the program

\[ \text{PROC II} \overset{\text{def}}{=} \text{frame}(x) \land (x = 1) \land \text{len}(1) \]

We reduce it in the following way:

\[ \text{frame}(x) \land (x = 1) \land \text{len}(1) \]
\[ \equiv \text{frame}(x) \land (x = 1) \land \emptyset \]
\[ \equiv \text{frame}(x) \land (x = 1) \land \emptyset \land \text{more} \]
\[ \equiv \emptyset \text{empty}(\text{frame}(x)) \land (x = 1) \land \emptyset \]
\[ \equiv x = 1 \land \emptyset \text{empty}(\text{frame}(x) \land \emptyset) \]

Then the normal form of this program is

\[ P \equiv q_{c1} \land \emptyset q_{f1} \]

where

\[ q_{c1} \equiv (x = 1) \]
\[ \equiv (x = 1 \land p_x) \lor (x = 1 \land \neg p_x) \quad \text{(Theorem 5)} \]

\[ q_{f1} \equiv \text{empty}(\text{frame}(x) \land \emptyset) \]
\[ \equiv \text{empty}(\text{frame}(x) \land \emptyset) \quad \text{(Law 25)} \]
\[ \equiv (x = 0 \land \neg p_x) \lor p_x \land \emptyset \quad \text{(Theorem 5)} \]
\[ \equiv (x = 1 \land \neg p_x) \land \emptyset \lor (p_x \land \emptyset) \]

This program is executed over an interval with two states, \( \sigma = (s_0, s_1) \). Conventionally, at each state, we use the pair \((x : e), [p_x]\) to express state formula \( x = e \land p_x \). And when there is no variable or proposition at a state, it is denoted by \( \emptyset \). Hence, four canonical models can satisfy the program \( P \):

\[ \sigma_1 = ((x:1), [p_x]), (\emptyset, [p_x])) \]
\[ \sigma_2 = ((x:1), [p_x]), ([x:1], \emptyset)) \]
\[ \sigma_3 = ((x:1), \emptyset), ([x:1], \emptyset)) \]
\[ \sigma_4 = ((x:1), \emptyset), ([x:1], \emptyset)) \]

However, the intended meaning of this program is captured by its minimal model \( \sigma_4 \). For this model, \( x \) is defined as 1 in both states of the interval.

**Example 6.** In the following, we give another framed program associated with local variables.

\[ \text{PROC IV} \overset{\text{def}}{=} y = 2 \land \text{sum} = 0 \land \text{while} (y > 0) \text{ do } \exists x : \text{frame}(x) \land x = y^2 \land \text{sum} := \text{sum} + x \land y := y - 1 \]

The program can be reduced as follows:

\[ y = 2 \land \text{sum} = 0 \land \text{while} (y > 0) \text{ do } \exists x : \text{frame}(x) \land x = y^2 \land \text{sum} := \text{sum} + x \land y := y - 1 \]
\[ \equiv y = 2 \land \text{sum} = 0 \land \text{while} (y > 0) \text{ do } \exists x : \text{frame}(z) \land z = y^2 \land \text{sum} := \text{sum} + z \land y := y - 1 \]
\[ \text{(where \( z \) does not appear in the whole program.)} \]
\[ \equiv y = 2 \land \text{sum} = 0 \land ((\text{frame}(z) \land z = y^2 \land \text{sum} := \text{sum} + z \land y := y - 1) ; q) \]
\[ \text{(where \( q \) denotes \text{while} (y > 0) \text{ do } \text{frame}(z) \land z = y^2 \land \text{sum} := \text{sum} + z \land y := y - 1))} \]
\[ \equiv y = 2 \land \text{sum} = 0 \land ((\text{frame}(z) \land z = 4 \land \text{sum} := \text{sum} + z \land y := y - 1) ; q) \quad \text{(Law 25)} \]
\[ \equiv y = 2 \land \text{sum} = 0 \land z = 4 \land ((\text{frame}(z) \land z = 4 \land \text{sum} := \text{sum} + y = 1 \land \text{empty}) ; q) \quad \text{(Law 10, 12)} \]
Thus, we have $\text{Prog IV} \equiv q_{c1} \land qf_1$, where

$$q_{c1} \equiv \neg y \land sum = 0 \land z = 4$$

$$= (y = 2 \land \neg p_y) \land (sum = 0 \land \neg p_{sum}) \land (z = 4 \land \neg p_z)$$

and

$$qf_1 \equiv (\text{frame}(z) \land \text{lf}(z) \land sum = 4 \land y = 1 \land empty) \land q$$

$$= (\text{lf}(z) \land sum = 4 \land y = 1 \land empty) \land q$$

$$= \text{lf}(z) \land sum = 4 \land y = 1 \land q$$

The reduction process into two phases: one for state reduction and the other for interval reduction. The state reduction is introduce some useful notations and then specify evaluation rules of expressions. For reducing framed programs, we divide for reasoning about program properties. This section focuses on the operational semantics for framed programs. We firstly programs are considered. To avoid ambiguity, whenever, a non-deterministic and/or non-terminable programs. However, some results are discussed in a broader scope in which non-deterministic programs and/or non-terminable within a state

Thus, we have $qf_1 \equiv q_{c2} \land qf_2$, where

$$q_{c2} \equiv (z \equiv 1) \land sum = 4 \land y = 1$$

$$= (z = 1 \land p_z) \land (sum = 4 \land \neg p_{sum}) \land (y = 1 \land \neg p_y)$$

and the reduction of $qf_2$ is similar to that of $qf_1$ shown below:

$$qf_2 \equiv (\text{frame}(z) \land \text{lf}(z) \land sum = 5 \land y = 0 \land empty) \land q$$

$$= \text{lf}(z) \land y = 0 \land sum = 5 \land empty$$

(Law 25)

We have $qf_2 \equiv q_e \land empty$, where

$$q_e \equiv \text{lf}(z) \land y = 0 \land sum = 5$$

$$= (z = 1 \land p_z) \land (y = 0 \land \neg p_y) \land (sum = 5 \land \neg p_{sum})$$

Therefore, by Definition 3, the model of $\text{Prog IV}$ is $\sigma = ((y : 2, sum : 0), \emptyset), ((y : 1, sum : 4), \emptyset), ((y : 0, sum : 5), \emptyset)$, without considering variable $z$.

A program $p$ is said to be deterministic if $p$ has one and only one minimal model. A program $p$ is called terminable if it can be reduced to a terminal product $p_e \land empty$. In the following, we are mainly concerned with deterministic framed programs. However, some results are discussed in a broader scope in which non-deterministic programs and/or non-terminable programs are considered. To avoid ambiguity, whenever, a non-deterministic and/or non-terminable program is involved, we clarify it in an explicit manner.

4. Operational semantics for Framed Tempura

One reason for requiring a formal definition of the semantics of a programming language is that it can serve as the basis for reasoning about program properties. This section focuses on the operational semantics for framed programs. We firstly introduce some useful notations and then specify evaluation rules of expressions. For reducing framed programs, we divide the reduction process into two phases: one for state reduction and the other for interval reduction. The state reduction is mainly on how to transform a program into its normal form. To do so, we give semantic equivalence rules and transition rules within a state. The interval reduction is concerned with a program executed over an interval. To this end, we formalize interval transition rules to transfer a program from one state to another.

4.1. Notations

Consider a framed program such as $p_1 \land p_2 \land \cdots \land p_m$, where $p_i (1 \leq i \leq m)$ is a framed program or true. Naturally, it is taken for granted to execute this kind of programs from the left to the right. However, in fact, operators such as $\land$ and $\lor$ are not sensitive to the order of the components in a program. To express directly this property of programs $\text{prog}_1, \ldots, \text{prog}_m$ connected by operators $\land$ and $\lor$, we use the following notations.

$$\land \{\text{prog}_1, \ldots, \text{prog}_m\} \overset{\text{def}}{=} \text{prog}_1 \land \cdots \land \text{prog}_m (m \geq 1)$$

$$\lor \{\text{prog}_1, \ldots, \text{prog}_m\} \overset{\text{def}}{=} \text{prog}_1 \lor \cdots \lor \text{prog}_m (m \geq 1)$$

For ease of discussion, in the following, we define a notation $p_{s(x)}$ called a state component as follows:
configuration is \((p, \sigma, s_2, 2)\), where \(\sigma = s_0, s_1 >
\)

**Fig. 7.** The relationship between \(\sigma\) and \(s_i\).

### Table 3

**Evaluation rules of arithmetic expressions**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>((n, \sigma_{i-1}, s_i, i) \downarrow n)</td>
</tr>
<tr>
<td>A2</td>
<td>((x, \sigma_{i-1}, s_i, i) \downarrow \Pi_i{s_i(x)})</td>
</tr>
<tr>
<td>A3</td>
<td>((\bigcirc^n \bigcirc^m x, \sigma_{i-1}, s_i, i) \downarrow \Pi_i{s_i(-(n-m))(x)}), if (1 \leq n - m \leq i)</td>
</tr>
<tr>
<td>A4</td>
<td>((\bigcirc^m \bigcirc^n x, \sigma_{i-1}, s_i, i) \downarrow \Pi_i{s_i(-(n-m))(x)}), if (1 \leq n - m \leq i)</td>
</tr>
<tr>
<td>A5</td>
<td>((e_0, \sigma_{i-1}, s_i, i) \downarrow n_0, (e_1, \sigma_{i-1}, s_i, i) \downarrow n_1), where (n = n_0 \text{ op } n_1)</td>
</tr>
</tbody>
</table>

A state program is the conjunction of state components such as \(ps(x_1) \land ps(x_2) \land \cdots \land ps(x_n)\). Actually, the state programs correspond to the present components \(p_{i_0}\) and \(p_{i_1}\) in the normal form.

We have two types of configurations, one for expressions, and the other for programs. A configuration regarding a program \(p\) is a quadruple \((p, \sigma_{i-1}, s_i, i)\), where \(p\) is a framed program, \(\sigma_{i-1} = (s_0, \ldots, s_{i-1})\) \((i > 0)\) a model which records information of all states, \(s_i\) the current state at which \(p\) is being executed and \(i\) a counter for counting the number of states in \(\sigma_{i-1}\). Further, for \(i = 0\), let \(\sigma_{-1} = \epsilon\) be an empty sequence. Thus, the initial configuration is \(c_0 = (p, \epsilon, s_0, 0)\). When a program is terminating, it is reduced to true and the state is written as \(\varnothing\). So the final configuration is \(c_f = (\text{true}, \sigma, \varnothing, |\sigma| + 1)\).

Actually, after program \(p\) is transformed from current state \(s_0\) to next state \(s_{i+1}\), \(s_{i+1}\) becomes the new current state and \(s_i\) is appended to \(\sigma_{i-1}\) as a reduced state. So \(\sigma_{i-1}\) is extended to \(\sigma_i\). Therefore, the length of \(\sigma\) is continuously increased with reduced states until the last state is appended to \(\sigma\). In this way, the whole model of program \(p\) can be obtained eventually. As shown in Fig. 7, \(s_0\) and \(s_1\) are the reduced states, \(s_2\) is the current state, \(\sigma = (s_0, s_1)\) does not contain current state \(s_2\) until it becomes a reduced state.

Similarly, for an arithmetic (or Boolean) expression \(\text{exp}\), the configuration is \((\text{exp}, \sigma_{i-1}, s_i, i)\).

For accessing to the values of variables in a state, we have the following notations as defined in [28]. Let \(s_i\) be a state, \(x\) and \(y\) variables. We assume that \(p_x\) denotes assignment flag \(p_x\) or \(-p_x\) and \(m\) is a value in \(D\). \(s_i[(m, p_x)/x]\) means that \(x\) is mapped to a pair \((m, p_x)\) and other variables are not changed at \(s_i\). Thus, we have,

\[
s_i[(m, p_x)/x](y) = \begin{cases} (m, p_x) & y = x \\ s_i(y) & y \neq x \end{cases}
\]

We use the projection function \(\Pi_i\) defined as usual to obtain the components of the pair. For instance, to get the first component from the pair, we have \(\Pi_1\{s_i(x)\} = \Pi_1\{m, p_x\} = m\).

Further, for convenience, we use a simple notation \(s_i[w]\) to mean that all variables and propositions appearing in state program \(w\) at current state \(s_i\) are instantiated with their values. As an example, \(s_i[(m_1, p_x_1)/x_1][(m_2, p_x_2)/x_2] \cdots [(m_n, p_x_n)/x_n]\) is abbreviated to \(s_i[w]\) if \(w = (x_1 = m_1 \land p_x_1) \land (x_2 = m_2 \land p_x_2) \cdots \land (x_n = m_n \land p_x_n)\).

### 4.2 Evaluation of expressions

The evaluation relation, \(\downarrow\), is concerned with how the overall results of expressions are obtained. The evaluation rules of arithmetic expressions are given in Table 3. Rule A1 tackles with integers and A2 with variables. They are straightforward. Rules A3 and A4 deal with the expressions with next (\(\bigcirc\)) and previous (\(\bigcirc\)) operators. In Framed Tempura, variables can refer to their previous values obtained from interval \(\sigma\). The condition, \(1 \leq n - m \leq i\), ensures that the evaluation of variables must be within the range of the interval. Actually, rule A2 can be treated as a special case of rule A3 with \(m = n\). Notice that the configuration \((\bigcirc^m x, \sigma_{i-1}, s_i, i)(m \geq 1)\) is not permitted since we cannot evaluate expressions involving only future operators in the current state. Rule A4 handles the evaluation of expressions with arithmetic operators.

**Example 7.** An expression configuration \((\bigcirc^n x, \sigma, s_2, 2)\), where \(\sigma = (s_0, p_x)/x\), \(s_1[(5, p_x)/x]\) and \(s_2\) is the current state, can be evaluated with three cases as follows:

1. \(n = 1\), \((\bigcirc x, \sigma, s_2, 2) \downarrow \Pi_1 \{s_1(x)\} = 5\) (A3)
2. \(n = 2\), \((\bigcirc^2 x, \sigma, s_2, 2) \downarrow \Pi_1 \{s_0(x)\} = 3\) (A3)
3. \(n = 3\), \((\bigcirc^3 x, \sigma, s_2, 2) \downarrow \Pi_1 \{s_0(x)\} = 3\) (A3)

1 Here, we denote \(\sigma_{i-1}\) by \(\sigma\).
By rule A3, the first two cases are well-formed configurations and $x$ is evaluated to 5 and 3, respectively. But the third case is a stuck configuration because the evaluation goes beyond the first state of the interval.

Rules B1–B6 in Table 4 deal with Boolean expressions. These rules can easily be understood as those in conventional programming languages.

4.3. State reduction

The meaning of elementary statements of Framed Tempura has been explained in Section 2.2.1. However, these descriptions are informal and hence it is impossible to interpret programs in a rigorous way. Therefore, we are motivated to formalize a group of operational rules and explore a well-suited approach to catch the minimal model for framed programs. We first introduce semantic equivalence rules to normalize a program, and then specify the transition rules within a state to catch the minimal models. All of these rules form a state reduction system. Accordingly, we further explore some properties regarding the state reduction system.

4.3.1. Semantic equivalence rules

As Example 5 shows, the normal form plays an essential role in the reduction of temporal logic programs. From Theorem 7, we know that every framed program has its normal form. In the process of reducing a program into its normal form, we need some logic laws to allow convenient reasoning and transferring of programs. For this purpose, we present the semantic equivalence rules in Table 5. The correctness of these rules has been proved in [16,17,18].

Rule Ass is concerned with the assignment statements including positive immediate assignment $x = e$, equality assignment $x = e$ and state framing $\text{lbf}(x)$. In the case of $x = e \land \text{lbf}(x)$ and $x = e \land \text{lbf}(x)$, which mean that $x$ is framed at one state, and there is a new value $e$ assigning to $x$, we record the new value and set flag $p_1$ to true. So, we reduce both of them to $x = e$ since $x = e$ defined as $x = e \land p_1$.

When $\text{OP}$ is in a program, implying that program $p$ will continue to be executed in the next state, more is appended to show that the interval over which the program is executed is not over. So, it is conjoined with the program using rule Next. Rule ALw states that $\Box p$ has different reduction rules depending on more or empty encountered in programs. With the former, $\Box p$ is reduced to $p \land \Box \Box p$, while with the latter, the program is reduced to $p \land \text{empty}$.

Statement $(r : q)$ can be handled by rule Chop. This rule is given in light of the structure of program $r$ in three forms. In the case of $(r = w \land p)$, and $w$ being a state program or true, $(r : q)$ is reduced to $(w \land (p : q))$; in the case of $(r = \Box p)$, $(r : q)$ is transformed to $(\Box (p : q))$; and in the case of $(r = \text{empty})$, $(r : q)$ is reduced to $q$.

Rules If and While deal with the conditional and iterative statements. Statement if $b$ then $p$ else $q$ is simply transformed to its equivalent program according to the definition; whereas while $b$ do $p$ is also equivalently transformed to if-statement. We use rule PAR to reduce parallel statement $p[q]$ by its definition. Note that $(q : \text{true} \land p)$ and $(q : (p : \text{true}))$ are mutually exclusive. Rule Fr is concerned with interval framing operator frame. Two cases need to be considered. If empty is in the programs, frame$(x) \land \text{empty}$ is reduced to empty, whereas if more is in the programs, frame$(x) \land \text{more}$ is rewritten to $\Box \text{lbf}(x) \land \text{frame}(x)$.

Rule Pry is specified according to different structures of $p_1, \ldots, p_m$ and $q$ so that the statement can be handled. $(p_1, \ldots, p_m) \text{pry empty}$ is transformed to $(p_1 \land \ldots \land p_{m+1})$, whether $p_m$ is reduced to $q$. Of the processes $p_1, \ldots, p_m$, if $p_i = w \land \text{empty}$ then $(p_1, \ldots, p_{i-1}, w \land \text{empty}, p_{i+1}, \ldots, p_m) \text{pry q}$ is rewritten to $(p_1, \ldots, p_{i-1}, (w \land p_{i+1}), \ldots, p_m) \text{pry q}$. Further, $(w \land p_1), p_2, \ldots, p_m) \text{pry q}$ is reduced to $w \land (p_1, \ldots, p_m) \text{pry q}$; and similarly, $(p_1, \ldots, p_m) \text{pry (w \land q)}$ is rewritten to $w \land (p_1, \ldots, p_m) \text{pry q}$. In addition, the structure of $(\Box p_1, \ldots, p_m) \text{pry q}$ is transformed to $(\Box (p_1, (p_2, \ldots, p_m) \text{pry q})$.

Statement $\exists x : p(x)$ can be managed by rule Loc. It tells us that the existential quantification in program $\exists x : p(x)$ can be removed by the renaming method. Rule Cong is a congruence rule, where $\text{prog}[q'/q]$ denotes the program given by replacing some occurrences of $q$ in $\text{prog}$ by $q'$ if $q \equiv q'$.
Semantic equivalence rules of framed programs

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sub-term</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASS</td>
<td>((\text{ASS} =) \land {\text{lb}(x), x = e} \equiv x = e)</td>
</tr>
<tr>
<td></td>
<td>((\text{ASS} \Leftarrow) \land {\text{lb}(x), x = e} \equiv x = e)</td>
</tr>
<tr>
<td>NXT</td>
<td>(\circ p \equiv \land (\circ p, \text{more}))</td>
</tr>
<tr>
<td>ALW</td>
<td>((1) \land {\neg p, \text{empty}} \equiv {p, \text{empty}}; (2) \land {\neg p, \text{more}} \equiv {p, \lor \circ p})</td>
</tr>
<tr>
<td>CHOP</td>
<td>((1) \land {w, p}; q \equiv \land {w, p; q}; (2) \lor p; q \equiv \lor {p; q}; (3) \emptyset q \equiv \emptyset q)</td>
</tr>
<tr>
<td>IF</td>
<td>if (b) then (p) else (q) (\equiv \lor {{{b, p}, \land {\neg b, q}}})</td>
</tr>
<tr>
<td>WIL</td>
<td>while (b) do (p) (\equiv) if (b) then ((p \land \text{more})); while (b) do (p) (\equiv) else (q)</td>
</tr>
<tr>
<td>PAR</td>
<td>(p</td>
</tr>
<tr>
<td>FR</td>
<td>((1) \land {\text{frame}(x), \text{empty}} \equiv \text{empty})</td>
</tr>
<tr>
<td></td>
<td>((2) \land {\text{frame}(x), \text{more}} \equiv \land {\text{lb}(x), \text{frame}(x)})</td>
</tr>
<tr>
<td>PR</td>
<td>((1) {p_1, \ldots, p_m} \text{pr} \text{empty} \equiv p_1; \ldots; p_m)</td>
</tr>
<tr>
<td></td>
<td>((2) \emptyset \text{pr} q \equiv q)</td>
</tr>
<tr>
<td></td>
<td>((3) {p_1, \ldots, p_{n+1}, \land {w, \text{empty}}, p_{n+1}, \ldots, p_m} \text{pr} q \equiv {p_1, \ldots, p_{n+1}, \land {w, \text{empty}}, p_{n+1}, \ldots, p_m} \text{pr} q)</td>
</tr>
<tr>
<td></td>
<td>((4) \land {w, p_1, \ldots, p_m} \text{pr} q \equiv \land {w, {p_1, \ldots, p_m} \text{pr} q})</td>
</tr>
<tr>
<td></td>
<td>((5) {p_1, \ldots, p_m} \text{pr} \land {w, q} \equiv \land {w, {p_1, \ldots, p_m} \text{pr} q})</td>
</tr>
<tr>
<td></td>
<td>((6) (\circ p_1, \ldots, p_m) \text{pr} \land q \equiv \lor {p_1, \ldots, p_m} \text{pr} q)</td>
</tr>
<tr>
<td>LOG</td>
<td>(\exists x: p(x) \equiv p(y)) if (p(y)) is a renamed program of (\exists x: p(x))</td>
</tr>
<tr>
<td>CONG</td>
<td>(q \equiv q')</td>
</tr>
</tbody>
</table>

Semantic equivalence rules regarding true and false are listed in Table 6. They are straightforward. As an example, more and empty are mutually exclusive, i.e., \(p \equiv \text{more}\) and \(\neg p \equiv \text{empty}\). For instance, by rules F3 and T3, we have \(\land \{\text{more}, \text{empty}\} \equiv \text{false}\) and \(\lor \{\text{more}, \text{empty}\} \equiv \text{true}\).

4.3.2. Transition rules within a state

Since adding framing operators to a program gives rise to non-monotonicity (see Example 5), we cannot adopt the structural approach to capture the semantics of statement \(p \land q\) by means of the composition of the semantics of \(p\) and \(q\). In addition, unlike imperative programming languages in which a variable must be initialized before it is used, the assignment statements \(\land \{x_1 \Leftarrow e_1, \ldots, x_n \Leftarrow e_n\}\) or \(\land \{x_1 = e_1, \ldots, x_n = e_n\}\) are executed concurrently, and the evaluation of \(e_1, \ldots, e_n\) such as in \((x = y + z) \land (z = y + x + 2) \land (y = z + x)\) may depend on one another. So it is crucial to evaluate one of them to a constant.2

Notation \(\rightarrow\) is a binary relation over the set of configurations w.r.t. a state. \(c \rightarrow c'\) implies that \(c\) is transformed to \(c'\) by several steps within a state.

Definition 8. Let \(\text{conf}\) be the set of all configurations.

\[
\begin{align*}
(1) & \quad 0 \rightarrow: \quad (c, c) \in \text{conf} \\
(2) & \quad i + 1 \rightarrow: \quad (a, b) i \rightarrow a, b \in \text{conf} \text{ and } \exists c \in \text{conf}, a \rightarrow c \text{ and } c \rightarrow b, i \geq 0 \\
(3) & \quad +: \quad \bigcup_{i \geq 0} i \\
(4) & \quad \rightarrow: \quad \bigcup_{i \geq 0} + 
\end{align*}
\]

In Table 7, transition rules within a state are given to deal with the concurrent assignments within a state. With these rules, we not only solve assignments like equations but also capture the minimal model. We assume a program is of three forms in the light of different assignment operators: \(\land \{x_1 \Leftarrow e_1, \ldots, x_n \Leftarrow e_n\}\) or \(\land \{x_1 = e_1, \ldots, x_n = e_n\}\) or \(\land \{p, \text{lb}(x)\}\). The purpose of treating a program with three forms is that we can easily get the minimal set of propositions by considering each kind of forms.

Rule SUB-TERM is given according to Theorem 2 and Corollary 3 in Section 2.1.4. It focuses on solving the equational assignments, where \(f\) is an \(m\)-arity function (more often a binary or unary function) on terms. After a variable is evaluated by rule SUB-TERM, we further use Rule Min1 or Min2 or Min3 to get the minimal set of propositions associated with the variable, and at the same time evaluate other variables using the constant.

Rule Min1 is concerned with concurrent positive immediate assignments. If \(e_j\) can be evaluated to constant \(n_j\), then conjunct \(x_j \Leftarrow e_j\) is eliminated from the program in the configuration, where and \(x_j\) is set to \(n_j\) and \(p x_j\) is set to true in state \(s_i\). What

---

2 For ease of specifying rules, we abbreviate \(\land \{x_1 \Leftarrow e_1, \ldots, x_n \Leftarrow e_n\}\) and \(\land \{x_1 = e_1, \ldots, x_n = e_n\}\) to \(\land_{k=1}^n \{x_k \Leftarrow e_k\}\) and \(\land_{k=1}^n \{x_k = e_k\}\), respectively.
actually happens is that \( x_j \leftarrow e_j \) is moved from the program to state \( s_i \) in the configuration, where \( x_k = e_k[n_j/x_j] \) means that variable \( x_j \) is replaced by value \( n_j \) in expression \( e_k \) \((1 \leq k \leq n \text{ and } k \neq j)\). Rules MIN2 and MIN3 are minimal rules with respect to selecting the minimal set of propositions for minimal models. It is obvious that the minimal rules are derived from the minimal model semantics. Rule MIN2 is based on the fact, \( x_j = e_j \equiv x_j = e_j \land p_{n_j} \lor x_j = e_j \land \neg p_{n_j} \). Obviously, \( x_j = e_j \land \neg p_{n_j} \) should be selected according to the minimal model semantics; whereas rule MIN 3 is dependent on the fact, \( \text{lbfc}(x) \equiv (x = \emptyset \land \neg p_{n_j}) \lor p_{x_j} \). Therefore, \( x = \emptyset \land \neg p_{n_j} \) is selected so that the minimal set of propositions can be obtained. Notice that rule MIN 3 does not hold at state \( s_0 \). Rule CONG II says that we can use program prog’ to replace prog in configurations as long as prog \( \equiv \) prog’.

One reason for the restriction without state components \( x_j = e_j \) or \( x_j \leftarrow e_j \) or \( \text{lbfc}(x_j) \) in \( p \) is that \( x = e \) and \( x \leftarrow e \) can be absorbed by the following two rules from Table 5:

\[
\begin{align*}
\text{(Ass =)} & \quad \text{lbfc}(x) \land x = e \equiv x \leftarrow e \\
\text{(Ass \leftarrow)} & \quad \text{lbfc}(x) \land x \leftarrow e \equiv x \leftarrow e
\end{align*}
\]

For example, by rule (Ass =), program \( \land(\text{lbfc}(x_1), x_1 = 1, x_2 = x_1 + 2, \emptyset) \) is transformed to \( \land(x_1 = 1, x_2 = x_1 + 2, \emptyset) \), where \( x_1 = 1 \) is absorbed. Another reason for the restriction is that \( x = e \) may conflict with \( x \leftarrow e \), leading to unsatisfiable programs. For example,

\[
\land(x_1 = 1, x_1 \leftarrow 2, x_2 = 0, \emptyset)
\]

is not a satisfiable program since \( x_1 = 1 \land x_1 \leftarrow 2 \equiv \) false. Notice that although we define equality \( x = e \), (or \( x \leftarrow e \)) is a program, however, by applying replacement laws of variables and rule SUB-TERM, an equation such as \( x + y = e \) can be a program.

**Example 8.** Evaluate variables \( x \) and \( y \) in program \( x + y = 3 \land x - y = 1 \land \emptyset \) by operational rules:

\[
\begin{align*}
&c_0 = (\land(x + y = 3, x - y = 1, \emptyset, s_0, 0) \\
&\rightarrow (\land(x + y = 3, x - y = 1, (x + y) + (x - y) = 3 + 1) \land \emptyset, s_0, 0) \quad \text{(SUB-TERM)} \\
&\rightarrow (\land(x + y = 3, x - y = 1, 2x = 4) \land \emptyset, s_0, 0) \quad \text{(SUB-TERM)} \\
&\rightarrow (\land(x + y = 3, x - y = 1, x = 2) \land \emptyset, s_0, 0) \quad \text{(MIN 2)} \\
&\rightarrow (\land(2 + y = 3, 2 - y = 1) \land \emptyset, s_0((2, \neg p_{x})/x), 0) \quad \text{(MIN 2)} \\
&\rightarrow (\land(y = 3, 3 - 2, 2 - y = 1) \land \emptyset, s_0((2, \neg p_{x})/x), 0) \quad \text{(MIN 2)} \\
&\rightarrow (\land(2 - 1 = 1) \land \emptyset, s_0((2, \neg p_{x})/x)((1, \neg p_{y})/y), 0) \quad \text{(MIN 2)} \\
&\rightarrow (\land(\emptyset, s_0((2, \neg p_{x})/x)((1, \neg p_{y})/y), 0)
\end{align*}
\]
4.3.3. Properties for state reduction

The state reduction enjoys some interesting properties. This subsection presents some of them.

**Lemma 8.** By using semantic equivalence rules in Tables 5 and 6, any framed program prog can be reduced to a framed program prog' in the normal form such that prog $\equiv$ prog'.

**Proof.** The proof of Lemma 8 can be found in the Appendix. □

Lemma 8 confirms that a framed program can be reduced to its normal form by means of semantic equivalence rules. Note that despite the conclusions of Lemma 8 and Theorem 7 are the same, the former is derived from the operational semantics while the latter is based on the model theory.

**Definition 9.** A reduced configuration $c_{rd}$ regarding any satisfiable program $p$ can be defined as

$$c_{rd} \overset{\text{def}}{=} (\emptyset, \sigma_{-1}, s_i[p_e], i) \text{ or } (\bigcirc p_f, \sigma_{-1}, s_i[p_c], i)$$

provided that $p \equiv p_e \land \emptyset$ or $p \equiv p_c \land \bigcirc p_f$, as defined in the normal form, is obtained.

**Definition 10.** A configuration $c_p = (p, \sigma_{-1}, s_i, i)$ is reducible if $c_p$ satisfies the following conditions: in $c_p \xrightarrow{n} c_{p_n} \ (n \geq 0$ and $c_{p_0} = c_p$), all of state variables within state components in $c_{p_0}$ can be evaluated to constants by operational rules.

Actually, Definition 10 gives a condition under which the operational rules can always be used in each step. For instance, $c_p = (\land (x + y = 3, x - y = 1), \epsilon, s_0, 0)$ is reducible while $c_p = (\land (y = x + z, \emptyset), \epsilon, s_0, 0)$ is not because $y$ cannot be evaluated to a constant.

**Lemma 9.** If $p$ is satisfiable and $c_p = (p, \sigma_{-1}, s_i, i)$ is reducible, we have $c_p \xrightarrow{\ast} c_{rd}$.

**Proof.** In configuration $c_p = (p, \sigma_{-1}, s_i, i)$, if expression $e$ in a state component, i.e., $x = e$ or $x \leftarrow e$ or $\text{left}(x)$, appearing in $p$, can be evaluated to constant $n$, then by transition rules within a state, pair $(n, p_x)$ can be transferred into state $s_i$. So, actually, state programs in a configuration can appear in two parts: one is in the present component $p_c$ (or $p_e$), and the other is transferred to state $s_i$. For simplicity, a state component can be written as the form $x = e \land p_x$, and program $p$ can be equivalently written as $p_e \land \bigcirc p_f$ (or $p_e \land \emptyset$, see Lemma 8). The proof proceeds by induction on $n$, the number of state components in $p_c$ (or $p_e$).

1. **Base.**

For $n = 1$, we have $p \equiv (x = e \land p_x) \land \bigcirc p_f$. Since $c_p$ is reducible, so it is certain that $e$ can be evaluated to a constant $m$.

Thus, by transition rules within a state, we have

$$(p, \sigma_{-1}, s_i, i) \rightarrow (\bigcirc p_f, \sigma_{-1}, s_i[(m, p_x)/x], i)$$

2. **Induction.**

Let $p \equiv (x_1 = e_1 \land p_{x_1}) \land \ldots \land (x_k = e_k \land p_{x_k}) \land \bigcirc p_f$ and $e_1, \ldots, e_k$ can be evaluated to $n_1, \ldots, n_k$. Suppose the following holds.

$$(p, \sigma_{-1}, s_i, i) \rightarrow (\bigcirc p_f, \sigma_{-1}, s_i[(\land_{h=1}^k(n_h, p_{x_h})/x_h)], i)$$

For $n = k + 1$, we have $p \equiv (x_1 = e_1 \land p_{x_1}) \land \ldots \land (x_{k+1} = e_{k+1} \land p_{x_{k+1}}) \land \bigcirc p_f$. Since $c_p$ is reducible, so for some $j(1 \leq j \leq k + 1)$, $e_j$ can be evaluated to $n_j$. By transition rules within a state, we have,

$$(p, \sigma_{-1}, s_i, i) \rightarrow (\bigcirc p_f, \sigma_{-1}, s_i[(\land_{h=1}^{k+1}(n_h, p_{x_h})/x_h)], i)$$

(by hypothesis)

$$= (\bigcirc p_f, \sigma_{-1}, s_i[p_c], i)$$

The proof of $(p, \sigma_{-1}, s_i, i) \rightarrow (\emptyset, \sigma_{-1}, s_i[p_e], i)$ is similar. Therefore, we have $c_p \xrightarrow{\ast} c_{rd}$. □

Lemma 9 tells us that the reduced configuration $c_{rd}$ can be reached by means of series of reduction steps within a state.

**Definition 11.** Configurations $c'$ and $c''$ are joinable, denoted by $c'' \Downarrow_{jo} c'$, if there is a configuration $c \in \text{conf}$ such that $c' \rightarrow c \rightarrow c''$.

**Theorem 10.** Let $p$ be a satisfiable deterministic program. For any reducible configuration $c_p$, relation $\rightarrow$ satisfies the Church–Rosser property.
\[ a \leftarrow c_p \rightarrow b \Rightarrow a \downarrow_{\text{jo}} b \]

where \( a, b \in \text{conf} \) and \( c_p = (p, \sigma_{i-1}, s_i, i) \).

**Proof.** Since program \( c_p \) is reducible, by Lemma 9, we have \( c_p \rightarrow^* c_{rd} \). Suppose that there exist configurations \( a \) and \( b \) such that \( c_p \rightarrow^* a \rightarrow c_{rd1} \) and \( c_p \rightarrow^* b \rightarrow c_{rd2} \). Furthermore, since \( p \) is a deterministic program, reduced configuration of \( p \) is unique. So, let \( c_{rd1} = c_{rd2} = c_{rd} \), we have, \( a \rightarrow c_{rd} \leftarrow b \). Therefore, \( a \downarrow_{\text{jo}} b \). \( \square \)

### 4.4. Interval reduction

One of the characteristic features of Framed Tempura is that programs are executed over a sequence of states (finite or infinite), and the reduction at each state is done by several steps. Actually, the semantic equivalence rules for statements and the transition rules within a state are used to transform a program into its normal form and catch the minimal model within a state by setting values of variables and propositions in the current state. Lemma 9 tells us that once all of variables and transition rules within a state are set, the remained subprogram is of the form \( \bigcirc p \) or empty. This means that the reduction of the program need move to the next state or stop. Therefore, interval transition rules are required to help the reduction process. Rule Tr1 in Table 8 is useful for dealing with \( \bigcirc p \) while rule Tr2 is helpful for reducing empty. Accordingly, the execution of \( \bigcirc p, \sigma_{i-1}, s_i, i \) means that \( p \) requests to be executed at the next state \( s_{i+1} \), and current state \( s_i \) needs to be appended to model \( \sigma_{i-1} \). So \( i \), the number of states in \( \sigma_{i-1} \), needs to be increased by one. The execution of \( \text{empty}, \sigma_{i-1}, s_i, i \) is simple. State \( s_i \) is appended to \( \sigma_{i-1} \) and the final configuration \( \text{true}, \sigma_i, \emptyset, i + 1 \) is reached. Thus, \( \sigma_i = \sigma_{i-1} \cdot (s_i) \) is the model of the program.

**Example 9.** By using the operational rules, we can re-reduce the program in Example 5.

\[
\begin{align*}
& \langle \bigwedge \text{frame}(x), x = 1, \text{empty}, \epsilon, s_0, 0 \rangle \rightarrow \langle \bigwedge \text{frame}(x), x = 1, \text{empty}, \epsilon, s_0, 0 \rangle \quad \text{\( \text{len}(1) \overset{\text{def}}{=} \text{empty} \)} \\
& \quad \rightarrow \langle \bigwedge \text{frame}(x), x = 1, \text{empty}, \epsilon, s_0, 0 \rangle \quad \text{\( \text{NEXT} \)} \\
& \quad \rightarrow \langle \bigwedge \text{frame}(x), x = 1, \text{empty}, \epsilon, s_0(1, \neg p_x/x), 0 \rangle \quad \text{\( \text{Fr}(2) \)} \\
& \quad \rightarrow \langle \bigwedge \text{frame}(x), x = 1, \text{empty}, \epsilon, s_0(1, \neg p_x/x), 0 \rangle \quad \text{\( \text{MIN2} \)} \\
& \quad \rightarrow \langle \text{true}, s_0(1, \neg p_x/x), s_1(1, \neg p_x/x), \emptyset, 2 \rangle \quad \text{\( \text{Tr2} \)}
\end{align*}
\]

Thus, the minimal model for the program is \( \sigma = \langle s_0(1, \neg p_x/x), s_1(1, \neg p_x/x) \rangle \). As we can see, applying these operational rules, we can directly obtain the minimal interpretation on propositions at each state, and eventually catch the minimal model for the program.

Binary relation \( \rightarrow \) between two configurations with different states is specified by rules in Table 8. We can union \( \rightarrow \) and \( \rightarrow^* \) by the definition of \( \rightarrow^* \) given below.

**Definition 12**

1. \( \rightarrow^* \) : \( \langle c, c' \rangle \in \text{conf} \) and \( c \rightarrow c' \)
2. \( \overset{\text{i} \geq 1}{\rightarrow} \) : \( \langle a, b \rangle \in \text{conf} \) and \( \exists c \in \text{conf} \), \( a \overset{i}{\rightarrow} c \) and \( c \rightarrow b \), \( i \geq 0 \)
3. \( \overset{\text{c} \geq 0}{\rightarrow} \) : \( \bigcup_{c \geq 0} \overset{\text{c}}{\rightarrow} \)
4. \( \overset{0}{\rightarrow} \) : \( \bigcup_{i \geq 0} \overset{0}{\rightarrow} \)

**Table 8**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Transition rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tr1</td>
<td>( \langle \bigcirc p, \sigma_{i-1}, s_i, i \rangle \rightarrow \langle p, \sigma_i, s_{i+1}, i + 1 \rangle )</td>
</tr>
<tr>
<td>Tr2</td>
<td>( \langle \text{empty}, \sigma_{i-1}, s_i, i \rangle \rightarrow \langle \text{true}, \sigma_i, \emptyset, i + 1 \rangle )</td>
</tr>
</tbody>
</table>
Let $p$ be a satisfiable terminable and deterministic program means of the operational rules. or at least one finite model. In the following, as we will see, the finite minimal model of a program can also be captured by the proof can be found in [16,17,18].

**Proof.** The proof can be found in [16,17,18].

Theorem 11. Let $p$ be a satisfiable framed program (which may be non-terminable, and/or non-deterministic). If, (1) $p$ has at least one finite model or (2) $p$ has finitely many models, then $p$ has at least one minimal model on propositions.

**Proof.** The proof can be found in [16,17,18].

Theorem 12. Let $p$ be a satisfiable terminable and deterministic program, and $c_0 = (p, \epsilon, s_0, 0)$. Then $c_0 \rightarrow c_f$ for some $c_f$.

5. Consistency between minimal model and operational semantics

5.1. Consistency for finite models

In this section, we explore the consistency between the minimal model semantics and the operational semantics with finite models. With the minimal model semantics, we have the following conclusion:

**Theorem 11.** Let $p$ be a satisfiable terminable and deterministic program means of the operational rules. Theorem 11 asserts the existence of a minimal model for a given program as long as the program has finitely many models.

**Proof.** The proof can be found in [16,17,18].

In this section, we explore the consistency between the minimal model semantics and the operational semantics with finite models. With the minimal model semantics, we have the following conclusion:

**Theorem 11.** Let $p$ be a satisfiable framed program (which may be non-terminable, and/or non-deterministic). If, (1) $p$ has at least one finite model or (2) $p$ has finitely many models, then $p$ has at least one minimal model on propositions.

**Proof.** The proof can be found in [16,17,18].

5. Consistency between minimal model and operational semantics

5.1. Consistency for finite models

In this section, we explore the consistency between the minimal model semantics and the operational semantics with finite models. With the minimal model semantics, we have the following conclusion:

**Theorem 11.** Let $p$ be a satisfiable framed program (which may be non-terminable, and/or non-deterministic). If, (1) $p$ has at least one finite model or (2) $p$ has finitely many models, then $p$ has at least one minimal model on propositions.

**Proof.** The proof can be found in [16,17,18].

In this section, we explore the consistency between the minimal model semantics and the operational semantics with finite models. With the minimal model semantics, we have the following conclusion:

**Theorem 11.** Let $p$ be a satisfiable framed program (which may be non-terminable, and/or non-deterministic). If, (1) $p$ has at least one finite model or (2) $p$ has finitely many models, then $p$ has at least one minimal model on propositions.
Theorem 13. Let \( p \) be a satisfiable terminable framed program. If \((p, \epsilon, s_0, 0) \rightarrow^* (\text{true}, \sigma, \emptyset, i+1)\), then \( \sigma \) is a minimal model of \( p \).

**Proof.** We consider the following two cases:

1. \( |\sigma| = 0 \)
   In this case, \( p \equiv w_0 \land \emptyset \), where \( w_0 \) is a state program or true. We can choose the minimal set of propositions from \( w_0 \) by means of rules MIN1 and MIN2 and MIN3, and denote the selected state program as \( w_{0_{\text{min}}} \). Accordingly, \( \langle s_0|w_{0_{\text{min}}} \rangle \subseteq \langle s_0|w_0 \rangle \) for any \( s_0|w_0 \rangle \models p \). Formally, we have,

\[
(p, \epsilon, s_0, 0) \rightarrow^* (w_0 \land \emptyset, \epsilon, s_0, 0) \quad \text{(Lemma 8)}
\]

\[
\rightarrow^* (\emptyset, \epsilon, s_0|w_{0_{\text{min}}}, 0) \quad \text{(MIN(1–3) and Lemma 9)}
\]

\[
\rightarrow (\text{true}, s_0|w_{0_{\text{min}}}, \emptyset, 1) \quad \text{(Tr2)}
\]

Therefore, \( \langle s_0|w_{0_{\text{min}}} \rangle \) is a minimal model of \( p \).

2. \( |\sigma| = n \geq 1 \)
   We first prove that any prefix of \( \sigma \) is a sequence of minimal interpretation on propositions. The proof proceeds by induction on the length of a prefix of \( \sigma \).

   (a) Base
   The prefix of \( \sigma \), \( (s_0) \), is a minimal interpretation.
   
   Suppose \( p \equiv w_0 \land \bigcirc p_1 \), where \( w_0 \) is a state program or true.

   \[
(p, \epsilon, s_0, 0) \rightarrow^* (w_0 \land \bigcirc p_1, \epsilon, s_0, 0) \quad \text{(Lemma 8)}
\]

\[
\rightarrow^* (\bigcirc p_1, \epsilon, s_0|w_{0_{\text{min}}}, 0) \quad \text{(MIN(1–3) and Lemma 9)}
\]

\[
\rightarrow (p_1, s_0|w_{0_{\text{min}}}, s_1, 1) \quad \text{(Tr1)}
\]

By rules MIN1 or MIN2 or MIN3, \( \langle s_0|w_{0_{\text{min}}} \rangle \) is a minimal sequence.

   (b) Induction
   Suppose that the prefix of \( \sigma \), \( (s_0, \ldots, s_{i-1}) \), is a minimal sequence. We prove that \( (s_0, \ldots, s_i) \) is a minimal model.
   
   Suppose \( p \) has been reduced to \( p_i \) over interval \( (s_0, \ldots, s_{i-1}) \), and \( p_i \equiv w_i \land \emptyset \).

   \[
(p, \epsilon, s_0, 0) \rightarrow^* (p_i, (s_0, \ldots, s_{i-1}), s_i, 1)
\]

\[
\rightarrow^* (w_i \land \emptyset, (s_0, \ldots, s_{i-1}), s_i, 1) \quad \text{(hypothesis)}
\]

\[
\rightarrow^* (\emptyset, (s_0, \ldots, s_{i-1}), s_i|w_{i_{\text{min}}}, 1) \quad \text{(MIN(1–3) and Lemma 9)}
\]

\[
\rightarrow (\text{true}, (s_0, \ldots, s_{i-1}, s_i|w_{i_{\text{min}}}, \emptyset, i+1) \quad \text{(Tr2)}
\]

By hypothesis, \( (s_0, \ldots, s_{i-1}) \) is a minimal prefix. Further, it is obvious that \( w_{i_{\text{min}}} \) is a minimal interpretation on propositions. So \( \sigma_i = (s_0, \ldots, s_i|w_{i_{\text{min}}}) \) is a minimal model of \( p \). \( \square \)

Theorems 12 and 13 are concerned with the semantics of framed programs with finite models. The semantics can be captured by means of operational rules given above. On the other hand, for any satisfiable framed programs with finite models, the semantics can also be captured by minimal model semantics (see Theorem 11). Therefore, under the condition of programs with finite models, the two types of the semantics are consistent. However, with infinite models, the consistency between the semantics given by operational rules and the minimal models needs to be further investigated.

5.2. Consistency for infinite models

In this section, the consistency between minimal model semantics and operational semantics over infinite intervals is proved using Knaster–Tarski’s fixed-point theorem [5].
Theorem 14. Let \( p \) be a satisfiable framed program (which can be non-terminable, and/or non-deterministic), then \( p \) has at least one minimal model on propositions.

Proof. The proof can be found in [18]. \( \square \)

Theorem 14 tells us that, for any framed program with finite or infinite models, the minimal models can be obtained by minimal model semantics. However, in the following, Theorem 15 is also concerned with minimal models, but is captured by operational semantics.

Theorem 15. Let \( p \) be a satisfiable, deterministic and non-terminable framed program, then by operational semantics, \( p \) has at least a minimal model on propositions.

Proof. The minimal model of program \( p_0 \equiv p \) can be obtained by the following reduction steps.

\[
\begin{align*}
c_0 &= (p_0, \epsilon, s_0, 0) \\
\quad \vdash (\bigcirc p_1, \epsilon, s_0[w_{0\text{min}}], 0) \\
\quad \vdash (p_1, s_0[w_{0\text{min}}], s_1, 1) = c_1 & \quad \text{(MIN(1–3) and Lemma 9)} \\
\quad \vdots& \quad \text{(TR2)} \\
\quad \vdash (\bigcirc p_n, s_0[w_{0\text{min}}], \ldots, s_{n-1}[w_{n-1\text{min}}], s_{n-1}[w_{n-1\text{min}}], n-1) = c_n & \quad \text{(MIN(1–3) and Lemma 9)} \\
\quad \vdots& \quad \text{(TR2)} \\
\end{align*}
\]

Therefore, we have
\[
c_0 \vdash (p_i, s_0[w_{0\text{min}}], \ldots, s_{i-1}[w_{i-1\text{min}}], s_i, i) = c_i
\]

Note that rules MIN(1–3) are used to select the minimal interpretation \( w_{\text{min}} \) on propositions at each state \( s_i \) and rule TR2 to transform a program from current state to another. Other reduction rules are implicitly used and we do not write them out. Naturally, we use \( \sigma_{i-1} \) to denote \( (s_0[w_{0\text{min}}], \ldots, s_{i-1}[w_{i-1\text{min}}]), (i \geq 1) \) and \( \sigma_{-1} = \epsilon \).

There is an infinite sequence of intervals, \( \sigma_0, \sigma_1, \ldots \) since the configuration sequence \( c_0 \rightarrow c_1 \rightarrow c_2 \rightarrow \cdots \) is infinite. Let \( N_\omega = N_0 \cup \{\omega\} \) and for all \( i \in N_\omega \), we have \( i < \omega \). We define prefix as follows.

\[
\text{prefix}(\sigma_j) = (\sigma_k | k \leq j \text{ and } k, j \in N_\omega \cup \{-1\})
\]

Let \( \sigma_\omega = (s_0, s_1, \ldots) \) and \( L_p = \text{prefix}(\sigma_\omega) = (\sigma_{-1}, \sigma_0, \ldots, \sigma_n, \ldots) \). We define binary relation \( \preceq \) over \( L_p : \sigma_1 \preceq \sigma_2 \text{iff } j \). It is not hard to see that \( (L_p, \preceq) \) is a partially ordered set which satisfies reflexive, transitive and antisymmetric. Further, for any \( \omega \) chain \( \sigma_\omega, \sigma_1, \ldots \) in \( (L_p, \preceq) \), there exists at least one upper bound. So \( (L_p, \preceq) \) is a complete partial order set (cposet) with bottom \( \sigma_{-1} \) (i.e., \( \epsilon \)). In the following, we give some notations.

1. Combination operator \( (\cup) \): \( \sigma_1 \cup \sigma_2 = \sigma_3 \text{ if } \sigma_1 \leq \sigma_2 \) otherwise undefined.
2. Concatenation operator \( (\cdot) \): \( \sigma \cdot \sigma' \) as defined in Section 2.
3. Projection operator \( (\Pi) \): for \( \sigma_i = (p_i, \sigma_{i-1}, s_i, i), \Pi_2(\sigma_i) = \sigma_{i-1}, \Pi_3(\sigma_i) = s_i \).
4. Last element of a sequence

- \( \text{lt}(\sigma_i) = (s_i) \text{ if } \sigma_i = (s_0, \ldots, s_i) \)
- \( \text{lt}(\epsilon) = \epsilon \)
- \( \text{lt}(\sigma) = \omega \), then \( \text{lt}(\sigma) \) is undefined.

Now we define function \( f \) as follows:

\[
f(\sigma_{i-1}) = \sigma_{i-1} \cdot (s_i[w_{i\text{min}}]), \quad i \in N_\omega
\]

where \( (s_i[w_{i\text{min}}]) = \text{lt}(\Pi_2(\sigma_i)) \) and configuration \( \sigma_i = (p_i, \sigma_{i-1} \cdot (s_i[w_{i\text{min}}]), s_{i+1}, i+1) \) comes from the reduction steps \( (p_0, \epsilon, s_0, 0) \vdash (p_i, \sigma_{i-1} \cdot (s_i[w_{i\text{min}}]), s_{i+1}, i+1) \). We have the following result.

\[
\begin{align*}
f(\bigcup_{n \in N_0} \sigma_{n-1}) &= \bigcup_{n \in N_0} \sigma_{n-1} & (\sigma \cdot \sigma' = \sigma \text{ if } |\sigma| = \omega) \\
\quad = \sigma_{-1} \cup \bigcup_{n \in N_0} \sigma_n \\
\quad = \epsilon \cup \bigcup_{n \in N_0} \sigma_n \\
\quad = \bigcup_{n \in N_0} \sigma_n = \bigcup_{n \in N_0} f(\sigma_{n-1})
\end{align*}
\]
Therefore, \( f \) is a continuous function over cposet \((\mathcal{P}, \leq)\). By Knaster–Tarski's fixed-point theorem, there exists a least fixed point \( \sigma_{\text{fix}} = \bigsqcup_{n \in \mathbb{N}_0} f^n(\epsilon) \). Let \( \sigma = \sigma_{\text{fix}} \).

Further, we prove that \( \sigma = \bigsqcup_{n \in \mathbb{N}_0} f^n(\epsilon) \) is a minimal model. To do so, we need to prove that any prefix of \( \sigma \) is a minimal prefix, i.e., a minimal sequence of interpretations on propositions. The proof proceeds by induction on \( n \), the length of a prefix.

(1) Base
For \( n = 0 \), we have \( f^0(\epsilon) = \epsilon \).

(2) Induction.
Suppose \( n = k \), \( f^k(\epsilon) \) is a minimal prefix of \( \sigma \). That is, \((p_0, \epsilon, s_0, 0) \xrightarrow{k} (p_k, \sigma_{k-1}, s_k, k) \) and \( \sigma_{k-1} \) is a minimal prefix of \( \sigma \). In the following, we prove that \( f^{k+1}(\epsilon) \) is a minimal prefix of \( \sigma \): \[
(p_0, \epsilon, s_0, 0) \xrightarrow{k} (p_k, \sigma_{k-1}, s_k, k)
\]
\[
\Rightarrow (\bigcirc p_{k+1}, \sigma_{k-1}, s_k[w_{k_{\text{min}}}], k)
\]
\[
\xrightarrow{} (p_{k+1}, \sigma_{k-1} \cdot (s_k[w_{k_{\text{min}}}]), s_{k+1}, k + 1)
\]
(\text{Min(1–3) and Lemma 9})

By rules Min(1–3), it is obviously that \( (s_k[w_{k_{\text{min}}}] \) is a minimal interpretation at state \( s_k \). Further, by hypothesis, \( \sigma_{k-1} \) is a minimal prefix of \( \sigma \). So \( \sigma_k = \sigma_{k-1} \cdot (s_k[w_{k_{\text{min}}}] \) is a minimal prefix of \( \sigma \).

Hence, for \( \forall n \in \mathbb{N}_0, f^n(\epsilon) \) is a minimal prefix of \( \sigma \). By Scott's fixed-point induction, \( \sigma = \bigsqcup_{n \in \mathbb{N}_0} f^n(\epsilon) \) is a minimal model. \( \square \)

By Theorems 14 and 15, both the model theoretic semantics and operational semantics can capture the minimal models of a program over infinite intervals, so the two kinds of the semantics are consistent.

### 5.3. Non-deterministic Framed Tempura

In this section, we further investigate the operational semantics of nondeterministic framed programs. For example, selection statement and conditional selection statement, etc. In the following, we first briefly introduce the syntax of both statements, and then specify their operational semantics.

- **The selection statement** can be defined directly by disjunction: \( p_1 \) or \( p_2 \) \( \text{def} \) \( p_1 \lor p_2 \)

Thus, a multiple selection statement is: \( \text{OR}_{k=1}^n p_k \) \( \text{def} \) \( p_1 \lor p_2 \lor \cdots \lor p_n \)

- **Conditional Selection**: \( c_1 \rightarrow p_1 \square c_2 \rightarrow p_2 \square \cdots \square c_n \rightarrow p_n \) is the guarded command construct proposed by Dijkstra [36], which can be represented using a multiple selection statement as: \( \text{OR}_{k=1}^n (\text{if} \ c_k \text{ then} \ p_k) \).

The constructs of the selection statement and conditional selection statement are the composition of the elementary statements in Framed Tempura. Therefore, to formalize their operational semantics, we can decompose their constructs into an elementary framed program such as \( p_k (1 \leq k \leq n) \) and the operational rules for reduction of \( p_k \) have been given in the previous sections. Therefore, we can specify rules (Non-Or) and (Non-Guarded) as follows:

(Non-Or) \[
(p_k, \sigma_{j-1}, s_i, i) \xrightarrow{} (q, \sigma_{j-1}, s_j, j) \quad (1 \leq k \leq n, 0 \leq i \leq j)
\]

(Non-Guarded) \[
(c_k, \sigma_{j-1}, s_i, i) \xrightarrow{} \text{true} \quad (1 \leq k \leq n)
\]

With the above non-deterministic reduction rules, the operational semantics of framed programs can be specified. In the following, we investigate some properties such as reachability and minimal models for nondeterministic programs.

To reduce a nondeterministic program such as \( p = p_1 \lor \cdots \lor p_{n_0} \lor p_1^0 \lor \cdots \lor p_{n_0}^0 \), here, \( p_1 \equiv p_1^0 (1 \leq i \leq n_0), 1 \leq n_0 \in \mathbb{N}_0 \).

We can select a subprogram \( p_m (1 \leq m \leq n_{0}) \) to execute, and obtain another non-deterministic program such as \( p_1^k \lor \cdots \lor p_{n_0}^k \) and \( 1 \leq n_1 \in \mathbb{N}_0 \). This process can continue until some successor of \( p \) terminates if \( p \) is terminable. In general, in the \( k \)-th step of the reduction process, we have, \( p_1^k \lor \cdots \lor p_{n_0}^k \), here \( 1 \leq n_k \) and \( k, n_k \in \mathbb{N}_0 \).

**Theorem 16.** Let \( p \) be a nondeterministic and satisfiable program. If \( p \) is terminable, we have \( (p, \epsilon, s_0, 0) \xrightarrow{} c_f \) for some \( c_f \).

**Proof.** Suppose the reduction process of \( p \) terminates in \( k \) \((k \geq 0)\) steps. We consider the following two cases.

(1) \( k = 0 \), i.e., program \( p \) terminates at the current state \( s_0 \). We assume that \( p \equiv p_1^0 \lor \cdots \lor p_{n_0}^0 \) \((n_0 \geq 1)\). There must exist \( m \) such that \( p_m^0 = \text{w} \land \text{empty} \) \((1 \leq m \leq n_0)\), where \( \text{w} \) is a state formula or true. Thus, we have
and further, there exists \( p_{m_1}, 0 \leq m_1 \leq n_1 \) that can be reduced to \( p_{m_1}^1 = w_1 \lor \bigcirc (p_0^1 \lor \cdots \lor p_{n_2}^1) \)

and similarly, we have the following sequence of reduction.

\[
\begin{align*}
(p_{m_0}^0, & \equiv w_0 \lor \bigcirc (p_0^1 \lor \cdots \lor p_{n_1}^1), 0) \\
\equiv ( & \bigcirc (p_0^1 \lor \cdots \lor p_{n_1}^1), \sigma_0[w_0], 0) \\
\Rightarrow ( & p_0^1 \lor \cdots \lor p_{n_1}^1, (s_0[w_0]), 1) \\
\end{align*}
\]

Therefore, by operational rule (Non-or), we have \( (p, \sigma, 0) \Rightarrow (p_0^1 \lor \cdots \lor p_{n_1}, (s_0[w_0]), 1) \).

(2) Induction. For \( r = 1 \), since \( p \equiv p_0^1 \lor \cdots \lor p_{n_0}^0 \), there exists a program \( p_{m_0}^0 \) such that

\[
(p_{m_0}^0, \epsilon, s_0, 0) \Rightarrow (\bigcirc (p_0^1 \lor \cdots \lor p_{n_1}^1), \epsilon, s_0[w_0], 0) \quad \text{(Lemma 9)}
\]

Therefore, by operational rule (Non-or), we have \( (p, \epsilon, 0) \Rightarrow (p_0^1 \lor \cdots \lor p_{n_1}, s_0[w_0], 1, 1) \).

Further, by operational rule (Non-or), \( (p_0^1 \lor \cdots \lor p_{n_1}^1, \sigma_{l-2}, s_{l-1}, l-1) \Rightarrow (p_0^1 \lor \cdots \lor p_{n_1}^1, \sigma_{l-1}, s_{l}, l) \). So we have \( (p, \epsilon, s_0, 0) \Rightarrow (p_0^1 \lor \cdots \lor p_{n_1}^1, \sigma_{l-1}, s_{l}, l) \).

Moreover, since the reduction process of \( p \) terminates in \( l \) steps, there must exist \( p_{m_l}^1, 0 \leq m_l \leq n_l \) such that

\[
(p_{m_l}^1, \sigma_{l-1}, s_{l}, l) \Rightarrow (\text{empty}, \sigma_{l-1}, s_{l}[w_1], l) \quad \text{(Lemma 9)}
\]

Therefore, by rule (Non-or), we have \( (p, \epsilon, 0) \Rightarrow \sigma \). \( \square \)

For a nondeterministic program, there may be more than one minimal models that can satisfy the program. In the following, Theorem 17 tells us that if \( p \) terminates, \( p \) has a finite minimal model, whereas Theorem 18 confirms the existence of infinite models for satisfiable, nonterminable and nondeterministic programs.

**Theorem 17.** Let \( p \) be a nondeterministic and satisfiable program. If \( p \) is terminable and \( (p, \epsilon, s_0, 0) \Rightarrow \text{true, } \sigma, \emptyset, |\sigma| + 1 \), then \( \sigma \) is a minimal model of \( p \).

**Proof.** Case 1. \( |\sigma| = 0 \). We assume that \( p \equiv p_1^0 \lor \cdots \lor p_{n_0}^0 \). There must exist a program \( p_m^0, 1 \leq m \leq n_0 \) such that \( p_m^0 \equiv w \land \text{empty} \), where \( w \) is a state formula:
Let $p$ be a nondeterministic and non-terminable program. If $p$ is satisfiable, then $p$ has an infinite minimal model.

**Proof.** The reduction of nondeterministic programs is different from deterministic ones. To execute such a program, we select one subprogram $p_k$ among $p_{k_1}^1, p_{k_2}^1, \ldots, p_{k_n}^1$ at each state $s_k$. For example, we assume that $p = p_1^0 \lor \cdots \lor p_n^0$ and we select a program $p_0 \in \{p_1^0, \ldots, p_n^0\}$ to execute. In this way, the selected programs can be obtained by the following reduction:

\[
\left( p_0^0, \epsilon, s_0, 0 \right) \rightarrow (w \land \text{empty}, \epsilon, s_0, 0) \quad \text{(Lemma 8)}
\]

\[
\rightarrow (\text{empty}, \epsilon, s_0[w_{\text{min}}], 0) \quad \text{(Lemma 9, Min(1–3))}
\]

\[
\rightarrow (\text{true}, (s_0[w_{\text{min}}], \varnothing, 1)) \quad \text{(Tr2)}
\]

We can choose the minimal set of propositions from $w$ by means of rules Min1 or Min2 or Min3, and denote the selected state program as $w_{\text{min}}$. Therefore, it is obvious that $\sigma = (s_0[w_{\text{min}}])$ is a minimal model of $p_0^m$. By rule (Non-Or), we have $(p_1^0 \lor \cdots \lor p_n^0, \epsilon, s_0, 0) \rightarrow (\text{true}, (s_0[w_{\text{min}}], \varnothing, 1))$. So, $\sigma = (s_0[w_{\text{min}}])$ is a minimal model of program $p$ as well.

**Case 2.** $|\sigma| = k (k \geq 1)$. We first prove that any prefix of $\sigma$ is a sequence of minimal interpretations on propositions. The proof proceeds by induction on the length of a prefix of $\sigma$.

**Base.** The prefix of $\sigma$, $(s_0)$, is a minimal interpretation. For $p = p_1^0 \lor \cdots \lor p_n^0$, there exists $p_0^m (1 \leq m \leq n_0)$. We assume that $p_0^m \equiv w \land \bigcirc p_1$, where $w$ is a state formula. We have

\[
\left( p_0^m, \epsilon, s_0, 0 \right) \rightarrow (w \land \bigcirc p_1, \epsilon, s_0, 0) \quad \text{(Lemma 8)}
\]

\[
\rightarrow (\bigcirc p_1, \epsilon, s_0[w_{\text{min}}], 0) \quad \text{(Lemma 9, Min(1–3))}
\]

\[
\rightarrow (p_1, (s_0[w_{\text{min}}], s_1, 1)) \quad \text{(Tr2)}
\]

It is obvious that $(s_0[w_{\text{min}}])$ is a minimal prefix of $\sigma$.

**Induction.** Suppose that the prefix of $\sigma$, $(s_0, \ldots, s_{k-1})$, is a minimal sequence. We prove that $(s_0, \ldots, s_k)$ is a minimal sequence.

From the above, we have $(p, \epsilon, s_0, 1) \rightarrow (p_1, s_0, 1)$ and further $(p_1, s_0, 1) \rightarrow (p_k, s_{k-1}, s_k, k)$. Suppose that $p_k = p_1^k \lor \cdots \lor p_n^k$. Since $|\sigma| = k$, i.e., $\sigma = (s_0, \ldots, s_k)$, there exists a program $p_1^k (1 \leq l \leq n_k)$ such that $p_1^k = u \land \text{empty}$, where $u$ is a state formula. So, we have

\[
\left( p_1^k, s_{k-1}, s_k, k \right) \rightarrow (u \land \text{empty}, s_{k-1}, s_k, k) \quad \text{(Lemma 8)}
\]

\[
\rightarrow (\text{empty}, s_{k-1}, s_k[u_{\text{min}}], k) \quad \text{(Lemma 9, Min(1–3))}
\]

\[
\rightarrow (\text{true}, s_{k-1} \cdot (s_k[u_{\text{min}}], \varnothing, k + 1)) \quad \text{(Tr2)}
\]

By rule (Non-Or), we have $(p_1^k \lor \cdots \lor p_n^k, s_{k-1}, s_k, k) \rightarrow (\text{true}, s_{k-1} \cdot (s_k[u_{\text{min}}], \varnothing, k + 1))$. Thus, $(p, \epsilon, s_0, 0) \rightarrow (\text{true}, s_{k-1} \cdot (s_k[u_{\text{min}}], \varnothing, k + 1))$. By hypothesis, $s_{k-1}$ is a minimal sequence, and by rules Min(1–3), $(s_k[u_{\text{min}}])$ is also minimal interpretation on propositions. So, $\sigma = s_{k-1} \cdot (s_k[u_{\text{min}}])$ is a minimal sequence. Since $p$ terminates at state $s_k$, $\sigma$ is a minimal model. □

**Theorem 18.** Let $p$ be a nondeterministic and non-terminable program. If $p$ is satisfiable, then $p$ has an infinite minimal model.
point theorem, there exists a least fixed point $\sigma_{fix} = \bigsqcup_{n \in \mathbb{N}_0} f^0(\epsilon)$. Let $\sigma = \sigma_{fix}$, $L_p$ is the set of all prefixes of $\sigma$. We now prove that $\sigma$ is a minimal model of $p$.

The proof proceeds by induction on $n$, the length of a prefix. Base. For $n = 0$, we have $f^0(\epsilon) = \epsilon$.

Induction. Suppose $n = k$, $f^{k}(\epsilon)$ is a minimal prefix of $\sigma$. That is $(p, \epsilon, s_0, 0) \xrightarrow{\epsilon} (p_k, \sigma_{k-1}, s_k, k)$ and $\sigma_{k-1}$ is a minimal prefix of $\sigma$. We assume that $p_k \equiv w_k \land \bigcirc (p_1^{k+1} \lor \cdots \lor p^{k+1}_{n_{k+1}})(k \geq 1, n_{k+1} \geq 1)$. In the following, we prove that $f^{k+1}(\epsilon)$ is a minimal prefix of $\sigma$.

\[
(p, \epsilon, s_0, 0) \xrightarrow{\epsilon} (p_k, \sigma_{k-1}, s_k, k) \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \equiv (\bigcirc (p_1^{k+1} \lor \cdots \lor p^{k+1}_{n_{k+1}}), \sigma_{k-1}, s_k|w_{k_{\min}}|, k) \quad \text{(Lemma 9, MIN(1-3))} \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \rightarrow (p_1^{k+1} \lor \cdots \lor p^{k+1}_{n_{k+1}}, \sigma_k, s_{k+1}, k + 1) \quad \text{(TrI)}
\]

By rules MIN(1-3), it is obvious that $(s_k|w_{k_{\min}}|)$ is a minimal interpretation at state $s_k$. Further, by hypothesis, $\sigma_{k-1}$ is a minimal prefix of $\sigma$. So $\sigma_k = \sigma_{k-1} \cdot (s_k|w_{k_{\min}}|)$ is a minimal prefix of $\sigma$. Therefore, for $\forall n \in \mathbb{N}_0$, $f^n(\epsilon)$ is a minimal prefix of $\sigma$. By Scott’s fixed-point induction, $\sigma = \bigsqcup_{n \in \mathbb{N}_0} f^n(\epsilon)$ is a minimal model of $p$. □

6. Conclusion

This paper investigated the operational semantics of the temporal logic programming language Framed Tempura. A new configuration with intervals is described. Based on the configuration, we establish a reduction system involving semantic equivalence rules, transition rules within a state and over intervals. These rules enable us to reduce framed programs and explore the properties of framed programs in a rigorous way. Moreover, in the reduction system, we proved some useful theorems regarding the reduction of programs. In particular, the theorems regarding the consistency between minimal model semantics [20] and operational semantics over finite or infinite intervals are proved. Based on the operational rules given in the paper, a new interpreter [23] has been developed in C++ and all rules have been implemented. Our experience convinces us the operational rules work well in practice.

In the future, we will investigate the communication and synchronization techniques between parallel and concurrent programs under the framework of operational semantics. Moreover, we will explore the axiomatic semantics of framed programs based on the proof system for PTL, and study the consistency among minimal model, axiomatic and operational semantics. In addition, as applications, on one hand, Framed Tempura can be employed as a modeling language to describe the behavior of composite Web services such as business processes of BPEL, and the framed programs can further be transformed to PROMELA; on the other hand, Propositional Projection Temporal Logic can be used to specify the properties of the required services, and the negation of the properties can also be transformed to Never Claim. Thus, the verification of composite Web services can be done using Model Checker SPIN [24,25].

Appendix A

The proof of Theorem 2

Proof. The proof proceeds by induction on the structure of $e(x)$. Obviously, we do not need to consider constants terms. Let $I = (\sigma, 0, k, |\sigma|)$ be an interpretation.

- $e(x)$ is a variable

  \[
  I \models t_1 = t_2 \\
  \iff I[t_1] = I[t_2] \\
  \iff I[e(t_1)] = I[e(t_2)] \\
  \iff I[e(t_1)] = e(t_2)
  \]

- $e(x)$ is a function $f(x)$

  \[
  I \models t_1 = t_2 \\
  \iff I[t_1] = I[t_2] \\
  \iff f(I[t_1]) = f(I[t_2]) \\
  \iff I[f(t_1)] = I[f(t_2)] \\
  \iff I[f(t_1)] = f(t_2) \quad \Box
  \]

The proof of Corollary 3

Proof. Let $I = (\sigma, 0, k, |\sigma|)$ be an interpretation. We have
The proof of Lemma 8

To simplify the proof, we only consider deterministic programs. For nondeterministic programs, a similar proof can be given. The proof proceeds by induction on the structure of statements. The state component \( \text{ps}(x) \) and statement empty can be thought of as basic statements and the others can be treated as composite statements. Note that the Cong I rule in Table 5 is implicitly used throughout the whole proof.

(1) If \( \text{prog} \) is the state component \( \text{ps}(x) \) then we have

\[
\begin{align*}
\text{ps}(x) & \triangleq \text{ps}(x) \land \text{true} \\
& \triangleq \text{ps}(x) \land (\text{more} \lor \text{empty}) \quad \text{(Table 6)} \\
& \equiv \text{ps}(x) \land \text{more} \lor \text{ps}(x) \land \text{empty} \\
& \equiv \text{ps}(x) \land \emptyset \lor \text{ps}(x) \land \text{empty} & (\text{more} \overset{\text{def}}{=} \text{true})
\end{align*}
\]

(2) If \( \text{prog} \) is the terminable statement, empty, the conclusion is straightforward.

(3) If \( \text{prog} \) is a statement in the form \( \Box \text{p} \), it is already in its normal form.

(4) If \( \text{prog} \) is the always statement \( \Box \text{p} \), then by the hypothesis we have, \( \text{p} \equiv \text{w} \land \text{empty} \lor \text{w}' \land \Box \text{p}' \), where \( \text{w}, \text{w}' \) are state programs or true:

\[
\begin{align*}
\Box \text{p} & \equiv \Box \text{p} \land (\text{more} \lor \text{empty}) \quad \text{(Table 6)} \\
& \equiv (\Box \text{p} \land \text{more}) \lor (\Box \text{p} \land \text{empty}) \\
& \equiv (\Box \text{p} \land \text{empty}) \lor (\Box \text{p} \land \text{more}) \\
& \lor ((\text{w} \land \text{empty} \lor \text{w}' \land \Box \text{p}') \land \Box \text{p}) \\
& \lor ((\text{w} \land \text{empty} \lor \text{w}' \land \Box \text{p}') \land \text{empty}) & (\text{hypothesis}) \\
& \equiv (\text{w}' \land \Box (\text{p} \lor \Box \text{p})) \lor ((\text{w} \land \text{empty})) & \text{(Table 6)}
\end{align*}
\]

(5) If \( \text{prog} \) is the conjunction statement \( \text{p} \land \text{q} \), then, by the hypothesis, we have,

\[
\begin{align*}
\text{p} \land \text{q} & \equiv (\text{w} \land \text{empty} \lor \text{w}' \land \Box \text{p}') \land (\text{u} \land \text{empty} \lor \text{u}' \land \Box \text{q}') \\
& \equiv (\text{w} \land \text{-empty} \lor \text{w}' \land \Box \text{p}') \land (\text{w} \land \text{u} \land \text{empty} \lor \text{w}' \land \Box (\text{p}' \land \text{q}')) & \text{(Table 6)}
\end{align*}
\]

(6) If \( \text{prog} \) is a sequential statement \( \text{p} ; \text{q} \), then, by hypothesis, we have,

\[
\begin{align*}
\text{p} \equiv \text{w} \land \text{empty} \lor \text{w}' \land \Box \text{p}', where \text{w} and \text{w}' are state programs or true. As empty and \Box \text{p}' are exclusive, so we consider them separately:
\]

\[
\begin{align*}
\text{w} \land \text{empty} ; \text{q} & \equiv \text{w} \land (\text{empty} ; \text{q}) & \text{(CHOR(1))} \\
\text{w} \land \text{empty} & \equiv \text{w} \land \text{empty} & \text{(CHOR(3))}
\end{align*}
\]

and

\[
\begin{align*}
\text{w}' \land \Box \text{p}' ; \text{q} & \equiv \text{w}' \land (\Box \text{p}' ; \text{q}) & \text{(CHOR(1))} \\
\text{w}' \land \Box \text{p}' & \equiv \text{w}' \land (\Box \text{p}' ; \text{q}) & \text{(CHOR(2))}
\end{align*}
\]

(7) If \( \text{prog} \) is the conditional statement, by hypothesis, we have \( \text{p} \equiv \text{w} \land \text{empty} \lor \text{w}' \land \Box \text{p}' \) and \( \text{q} \equiv \text{u} \land \text{empty} \lor \text{u}' \land \Box \text{q}' \), where \( \text{w}, \text{w}', \text{u}, \text{u}' \) are state programs or true. Thus, by If rule, we have

\[
\text{if } \text{b} \text{ then } \text{p else } \text{q} \equiv (\text{b} \land \text{p}) \lor (\neg \text{b} \land \text{q}) \quad \text{(If)}
\]

As seen, if \( \text{b} \) is true the statement is reduced to \( \text{p} \equiv \text{w} \land \text{empty} \lor \text{w}' \land \Box \text{p}' \) otherwise it is reduced to \( \text{q} \equiv \text{u} \land \text{empty} \lor \text{u}' \land \Box \text{q}' \).

(8) If \( \text{prog} \) is the while statement, we assume that \( \text{p} \equiv (\text{w} \land \Box \text{q}) \lor (\text{u} \land \text{empty}) \). By Wh rule, we have

\[
\text{I} \models (t_1 = e_1) \land \cdots \land (t_m = e_m)
\]

\[
\iff \text{I} \models t_1 = e_1 \text{ and } \cdots \text{ and } \text{I} \models t_m = e_m
\]

\[
\iff \text{I}[t_1] = \text{I}[e_1] \text{ and } \cdots \text{ and } \text{I}[t_m] = \text{I}[e_m]
\]

\[
\iff f(\text{I}[t_1], \ldots, \text{I}[t_m]) = f(\text{I}[e_1], \ldots, \text{I}[e_m])
\]

\[
\iff \text{I}[f(t_1, \ldots, t_m)] = \text{I}[f(e_1, \ldots, e_m)]
\]

\[
\iff \text{I} \models f(t_1, \ldots, t_m) = f(e_1, \ldots, e_m)
\]
while \( b \) do \( p \) \( \equiv \) if \( b \) then \((p \land \text{more}; \text{while } b \text{ do } p)\) else
\[ (\neg b \land \text{empty}) \quad (\text{IF}) \]
\[ = (b \land (p \land \text{more}; \text{while } b \text{ do } p)) \lor (\neg b \land \text{empty}) \quad (\text{more } \equiv \text{true}) \]
\[ = (b \land ((w \land q) \lor (u \land \text{empty}) \land \text{true}; \text{while } b \text{ do } p)) \lor (\neg b \land \text{empty}) \quad (\text{hypothesis}) \]
\[ = (b \land (w \land q \land \text{true}; \text{while } b \text{ do } p)) \lor (\neg b \land \text{empty}) \quad (\text{Chop}(1)) \]
\[ = (b \land (w \land q \land \text{true}; \text{while } b \text{ do } p)) \lor (\neg b \land \text{empty}) \quad (\text{Chop}(2)) \]
\[ = b \land w \land q \land \text{true}; \text{while } b \text{ do } p) \lor (\neg b \land \text{empty}) \]

(9) If \( \text{prog} \) is the parallel statement, \( p|q \), by PAR rule, we have
\[ p|q \equiv ((q : \text{true}) \land p) \lor (q \land (p : \text{true})) \]

It is obviously that \((q : \text{true}) \land p \) and \( q \land (p : \text{true}) \) are exclusive. Therefore, we consider them separately. Since the normal forms of sequential statement and conjunctive statement have been proved, so the parallel statement has its normal form.

(10) If \( \text{prog} \) is the projection statement \((p_1, \ldots, p_m)\) \( \text{prj} \) \( q \), by the hypothesis, we have,
\[ p_i \equiv w'_i \land \bigcirc p'_i \lor w'_i \land \text{empty}(1 \leq i \leq m), q \equiv u' \land \bigcirc q' \lor u \land \text{empty} \]

Where \( w_i, w'_i, u, u' \) are state programs or true. According to the context in which \( \text{prog} \) is executed, we consider the projection statement in the following four cases.

(a) \( p_1 \equiv w'_1 \land \bigcirc p'_1, q \equiv u' \land \bigcirc q' \)

\[(p_1, \ldots, p_m) \text{prj } q \equiv ((w'_1 \land \bigcirc p'_1), p_2, \ldots, p_m) \text{prj } (u' \land \bigcirc q') \]
\[= w'_1 \land ((\bigcirc p'_1, p_2, \ldots, p_m) \text{prj } (u' \land \bigcirc q')) \quad (\text{Prj}(4)) \]
\[= w'_1 \land u' \land (\bigcirc p'_1, p_2, \ldots, p_m) \text{prj } q' \quad (\text{Prj}(5)) \]
\[= w'_1 \land u' \land (\bigcirc p'_1, p_2, \ldots, p_m) \text{prj } q' \quad (\text{Prj}(6)) \]

(b) \( p_1 \equiv w'_1 \land \bigcirc p'_1, q \equiv u \land \text{empty} \)

\[(p_1, \ldots, p_m) \text{prj } q \equiv ((w'_1 \land \bigcirc p'_1), p_2, \ldots, p_m) \text{prj } (u \land \text{empty}) \]
\[= w'_1 \land ((\bigcirc p'_1, p_2, \ldots, p_m) \text{prj } (u \land \text{empty})) \quad (\text{Prj}(4)) \]
\[= w'_1 \land u \land (\bigcirc p'_1, p_2, \ldots, p_m) \text{prj } \text{empty} \quad (\text{Prj}(5)) \]
\[= w'_1 \land u \land (\bigcirc p'_1, p_2, \ldots, p_m) \text{prj } q \quad (\text{Prj}(1)) \]
\[= w'_1 \land u \land (\bigcirc p'_1, p_2, \ldots, p_m) \text{prj } q \quad (\text{Chop}(2)) \]

(c) \( p_1 \equiv w_1 \land \text{empty}, q \equiv u \land \text{empty} \)

\[(p_1, \ldots, p_m) \text{prj } q \equiv ((w_1 \land \text{empty}), p_2, \ldots, p_m) \text{prj } (u \land \text{empty}) \]
\[= w_1 \land ((\text{empty}, p_2, \ldots, p_m) \text{prj } (u \land \text{empty})) \quad (\text{Prj}(4)) \]
\[= w_1 \land u \land (\text{empty}, p_2, \ldots, p_m) \text{prj } \text{empty} \quad (\text{Prj}(5)) \]
\[= w_1 \land u \land (\text{empty}, p_2, \ldots, p_m) \text{prj } q \quad (\text{Prj}(1)) \]
\[= w_1 \land u \land (p_2, \ldots, p_m) \quad (\text{Chop}(3)) \]

We have proved that the sequential statement can be reduced to a normal form. Therefore, \( p_2, \ldots, p_m \equiv v \land \text{empty} \lor l \land \bigcirc \text{prog}' \), where \( v \) and \( l \) are state programs or true. Thus, we have the following conclusion:

\[(p_1, \ldots, p_m) \text{prj } q \equiv w_1 \land u \land (v \land \text{empty} \lor l \land \bigcirc \text{prog}') \]
\[= (w_1 \land u \land v \land \text{empty} \lor (w_1 \land u \land l \land \bigcirc \text{prog}')) \]

(d) \( p_1 \equiv w_1 \land \text{empty}, q \equiv u' \land \bigcirc q' \)

\[(p_1, \ldots, p_m) \text{prj } q \equiv ((w_1 \land \text{empty}), p_2, \ldots, p_m) \text{prj } (u' \land \bigcirc q') \]
\[= w_1 \land ((\text{empty}, p_2, \ldots, p_m) \text{prj } (u' \land \bigcirc q')) \quad (\text{Prj}(4)) \]
\[= w_1 \land u' \land ((\text{empty}, p_2, \ldots, p_m) \text{prj } q) \quad (\text{Prj}(5)) \]
\[= w_1 \land u' \land ((\text{empty}, p_2, \ldots, p_m) \text{prj } q) \quad (\text{Prj}(3)) \]

By hypothesis, the normal forms of \( p_2, \ldots, p_m \) are \( w'_i \land \bigcirc p'_i \lor w'_i \land \text{empty}(2 \leq i \leq m) \). Since the form \((w'_1 \land \bigcirc p'_1) \text{prj } q \) has been proved to have the normal form as shown in (a), we now only need to consider construct \((w_i \land \text{empty}) \text{prj } q \)'

\[(p_1, \ldots, p_m) \text{prj } q \equiv w_1 \land u' \land ((w_2 \land \text{empty}, \ldots, w_m \land \text{empty}) \text{prj } q) \]
\[= w_1 \land u' \land (w_2 \land \ldots, w_m \land (\text{empty}, \ldots, \text{empty}) \text{prj } q) \quad (\text{Prj}(4)) \]
\[= w_1 \land u' \land (w_2 \land \ldots, w_m \land (\text{empty}) \text{prj } q) \quad (\text{Prj}(3)) \]
\[= (w_1 \land w_2 \land \ldots, w_m \land u') \land q' \quad (\text{Prj}(2)) \]
So (d) can be reduced to a normal form. Therefore, Lemma 8 holds.

(12) If \( \text{prog} \) is the existential quantification statement \( \exists x : p(x) \), and \( p(y) \) is a renamed formula of \( \exists x : p(x) \). Suppose that \( p(y) \) can be reduced to its normal form such as \( p(y) \equiv (w(y) \land \text{empty}) \lor (u(y) \land \Box q(y)) \), so we have

\[
\exists x : p(x) \equiv p(y) \equiv (w(y) \land \text{empty}) \lor (u(y) \land \Box q(y))
\]

References