ITL$^+$: A Temporal Programming Model with Atomic Blocks (Extended Version)

Xiaoxiao Yang[1], Yu Zhang[1], Ming Fu[2], and Xinyu Feng[2]

[1] Institute of Software, Chinese Academy of Sciences, Beijing, China
[2] University of Science and Technology of China, Hefei, China
{xxyang, yzhang}@ios.ac.cn {fuming,xyfeng}@ustc.edu.cn

Abstract. Atomic blocks, a high-level language construct that allows programmers to explicitly specify the atomicity of operations without worrying about the implementations, are a promising approach that simplifies concurrent programming. On the other hand, temporal logic is a successful model in logic programming and concurrency verification, but none of existing temporal programming models supports concurrent programming with atomic blocks yet.
In this paper, we propose a temporal programming model (ITL$^+$) which extends the interval temporal logic (ITL) to support programming with atomic blocks. The novel construct that formulates atomic execution of code blocks, which we call *atomic interval formulas*, is always interpreted over two consecutive states, with the internal states of the block being abstracted away. We show that the *framing* mechanism of ITL also works in the new model, which consequently supports our development of an executive language based on ITL$^+$. The language supports concurrency by introducing a loose interleaving semantics which tracks only the mutual exclusion between atomic blocks. We demonstrate the usage of ITL$^+$ by modeling practical concurrent programs.

1 Introduction

Atomic blocks in the forms of **atomic**$\{C\}$ or $\langle C \rangle$ are a high-level language construct that allows programmers to explicitly specify the atomicity of the operation C, without worrying how the atomicity is achieved by the underlying language implementation. They can be used to model architecture-supported atomic instructions, such as compare-and-swap (CAS), or to model high-level transactions implemented by software transactional memory (STM). They are viewed as a promising approach to simplify concurrent programming in a multi-core era, and have been used in both theoretical study of fine-grained concurrency verification [21] and in mordern programming languages [9, 16, 25] to support transactional programming.

On the other side, temporal logic has proved very useful in specifying and verifying concurrent programs [17] and has seen particular success in temproal logic programming model, where both algorithms and their properties can be specified in the same language [1]. Indeed, a number of temporal logic programming languages have been developed for the purpose of simulation and verification of software and hardware systems, such as Temporal Logic of Actions (TLA) [11], Catus [18], Tempura [15],

MSVL [6, 23], etc. All these make a good foundation of applying temporal logic to implement and verify concurrent algorithms.

However, to our best knowledge, none of these languages support concurrent programming with atomic blocks. One can certainly implement arbitrary atomic code blocks using traditional concurrent mechanism like locks, but this misses the point of providing abstract and implementation-independent constructs for atomicity declarations, and the resulting programs could be very complex and increase dramatically the burden of programming and verification.

Therefore, we are motivated to define the notation of *atomicity* in the framework of temporal logic and propose a new temporal logic programming language ITL$^+$. Our work is based on the interval temporal logic (ITL) and we extend related programming languages such as Tempura and MSVL with the mechanism of executing code blocks *atomically*, together with a novel parallel operator that tracks only interleaving of atomic blocks. ITL$^+$ could facilitate specifying, verifying and developing reactive systems in a more efficient and uniform way. The framing operators and the minimal model semantics inherited from ITL may enable us to narrow the gap between temporal logic and programming languages in a realistic way.

Our contributions are summarized as follows:

- We extend ITL with the notion of *atomicity*, which supports the formulation of atomic executions of code blocks. The semantics is defined in the interval model and an atomic block is always interpreted over two consecutive states, with internal state transitions abstracted away. Such a formalization respects the essence of atomic execution: the environment must not interfere with the execution of the code block, and the internal execution states of atomic blocks must not be visible from outside. In fact, formulas inside atomic blocks are interpreted over separate intervals in our model, and the connection between the two levels of intervals is precisely defined.
- How values are carried through intervals is a central concern of temporal logic programming. We adopt Duan's framing technique using assignment flags and minimal models [5], and show that the framing technique works smoothly with the two levels of intervals, which can carry necessary values into and out of atomic blocks — the abstraction of internal states does not affect the framing in our model. The technique indeed supports our development of an executable language based on ITL$^+$.
- We define a novel notion of parallelism by considering only the interleaving of atomic blocks — parallel composition of two programs (formulas) without any atomic blocks will be translated directly into their conjunctions. For instance, $x = 1 \parallel y = 1$ will be translated into $x = 1 \wedge y = 1$, which reflects the fact that accesses from different threads to different memory locations can indeed occur simultaneously. Such a translation enforces that access to shared memory locations must be done in atomic blocks, otherwise the formulas can result in false, which indicates a flaw in the program.

  The loose notion of parallelism also helps to largely reduce the state spaces of corresponding automata, as we no longer consider interleaving of non-atomic codes, which is one of the main sources of state explosion in concurrent programs.

– ITL$^+$ not only allows us to express various low-level memory primitives like CAS, but also make it possible to model transactions of arbitrary granularity. We illustrate the practical use of ITL$^+$ by examples.

The rest of the paper is organized as follows: in Section 2, we briefly review the temporal logic ITL and Duan's framing technique, then we extend the logic and define ITL$^+$ in Section 3, as well as its semantics in the interval model. In particular, we show that the framing technique works well with atomic blocks. Section 4 presents an executable subset of ITL$^+$ for modeling concurrent programs with atomic blocks. In Section 5, we illustrate some programming examples and discuss simulation and verification tools based on ITL$^+$. Section 6 concludes the paper.

2 Interval temporal logic

We start with a brief introduction to interval temporal logic (ITL), which was proposed for reasoning about intervals of time for hardware and software systems [15, 5].

2.1 ITL basics

Syntax of ITL terms and formulas. ITL terms and formulas are defined by the following grammar:

$$\begin{array}{lll} \text{ITL terms:} & e, e_1, \ldots, e_m & ::= x \mid f(e_1, \ldots, e_m) \mid \bigcirc e \mid \ominus e \\ \text{ITL formulas} & Q, Q_1, Q_2 & ::= \pi \mid e_1 = e_2 \mid \text{Pred}(e_1, \ldots, e_m) \mid \neg Q \mid Q_1 \wedge Q_2 \\ & & \quad \mid \bigcirc Q \mid Q_1 \,;\, Q_2 \mid \exists x.Q \mid Q^+ \end{array}$$

where x is a variable, f ranges over a predefined set of function symbols, $\bigcirc e$ and $\ominus e$ indicate that term e is evaluated on the next and previous states respectively; π ranges over a predefined set of atomic propositions, and $\text{Pred}(e_1, \ldots, e_m)$ represents a predefined predicate constructed with $e_1, \ldots, e_m$; operators *next*($\bigcirc$), *chop*(;) and *chop plus* ($^+$) are temporal operators.

Semantics of ITL terms and formulas. ITL formulas are interpreted as the following:

$$(\sigma, i, k, j) \models_{itl} Q,$$

where an *interval* $\sigma = \langle\!\langle s_0, s_1, \ldots \rangle\!\rangle$ is a sequence of states, i, j, k are the indices of the initial, last and current states respectively, and Q is the ITL formula. The interval can be infinite and in that case we write ω for the third index. We call the tuple (σ, i, k, j) an *interpretation* $\Im$.

A state in an interval is a pair of assignments (I_{var}, I_{prop}) such that

$$I_{var} \in \mathcal{V} \to \mathcal{D} \cup \{\mathsf{nil}\}, \qquad I_{prop} \in \mathcal{P}rop \to \{\mathsf{True}, \mathsf{False}\}$$

where $\mathcal{V}$ is the set of variables, and $\mathcal{D}$ is the set of values including integers, lists, etc.. nil denotes an undefined value. $\mathcal{P}rop$ is a set of propositions. Both I_{var} and I_{prop} are total

$$
\begin{array}{lcl}
(\sigma, i, k, j)[x] & = & s_k[x] \\
\Im[f(e_1, \ldots, e_m)] & = & \begin{cases} f(\Im[e_1], \ldots, \Im[e_m]) & \text{if } \Im[e_h] \neq nil \text{ for all } h \\ nil & \text{otherwise} \end{cases} \\
(\sigma, i, k, j)[\bigcirc e] & = & \begin{cases} (\sigma, i, k+1, j)[e] & \text{if } k < j \\ nil & \text{otherwise} \end{cases} \\
(\sigma, i, k, j)[\ominus e] & = & \begin{cases} (\sigma, i, k-1, j)[e] & \text{if } i < k \\ nil & \text{otherwise} \end{cases} \\
\Im \models_{itl} \pi & \text{iff} & \Im[\pi] = \mathsf{True} \\
\Im \models_{itl} e_1 = e_2 & \text{iff} & \Im[e_1] = \Im[e_2] \\
\Im \models_{itl} \mathrm{Pred}(e_1, \ldots, e_m) & \text{iff} & \mathrm{Pred}(\Im[e_1], \ldots, \Im[e_m]) = \mathsf{True} \\
\Im \models_{itl} \neg Q & \text{iff} & \Im \not\models_{itl} Q \\
\Im \models_{itl} Q_1 \wedge Q_2 & \text{iff} & \Im \models_{itl} Q_1 \text{ and } \Im \models_{itl} Q_2 \\
(\sigma, i, k, j) \models_{itl} \bigcirc Q & \text{iff} & k < j \text{ and } (\sigma, i, k+1, j) \models_{itl} Q \\
(\sigma, i, k, j) \models_{itl} Q_1 ; Q_2 & \text{iff} & \text{there exists } r \text{ s.t.} (\sigma, i, k, r) \models_{itl} Q_1 \text{ and } (\sigma, r, r, j) \models_{itl} Q_2
\end{array}
$$

Fig. 1. Semantics for ITL Terms and Formulas.

$$
\varepsilon \stackrel{\text{def}}{=} \neg \bigcirc \mathsf{True} \qquad \mathsf{more} \stackrel{\text{def}}{=} \neg\varepsilon \qquad \Box Q \stackrel{\text{def}}{=} \neg\Diamond\neg Q
$$

$$
\mathsf{len}(n) \stackrel{\text{def}}{=} \begin{cases} \varepsilon & \text{if } n = 0 \\ \bigcirc \mathsf{len}(n-1) & \text{if } n > 1 \end{cases} \qquad \mathsf{skip} \stackrel{\text{def}}{=} \mathsf{len}(1) \qquad x := e \stackrel{\text{def}}{=} \bigcirc x = e \wedge \mathsf{skip}
$$

Fig. 2. Some derived ITL formulas.

functions. We write $s[x]$ for the value $I_{var}(x)$, and $s[\pi]$ for the boolean value $I_{prop}(\pi)$. The length of σ, denoted by $|\sigma|$, is defined as follows.

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

An empty interval is denoted by ϵ.

The interpretation $(\sigma, i, k, j) \models_{itl} Q$ says that the formula Q is satisfied by the subinterval of σ starting from state s_i and ending at s_j (j can be ω). Interpretations of ITL terms and formulas are defined in Figure 1. The semantics of $Q_1 \,;\, Q_2$ says that computation Q_2 follows Q_1, and the intervals for Q_1 and Q_2 share a common state. ε specifies intervals whose current state is the final state; an interval satisfying more requires that the current state not be the final state; $\Box Q$ means Q holds at every state after (including) the current state; $\mathsf{len}(n)$ means that the distance from the current state to the final state is n; skip specifies intervals with the length 1. $x := e$ means that at the next state $x = e$ holds and the length of the interval over which the assignment takes place is 1.

2.2 Framing and Minimal Model

Framing concerns how the value of a variable can be carried from one state to the next, which has been employed by the conventional imperative languages for years. In temporal logic programming, it offers no ready-made solution in this respect as values are not assumed to be carried forward. In the 1990s, framing issue was discussed amongst researchers interested in temporal logic programming [5, 8, 10, 11, 19]. However, no consensus is given regarding what is the best underlying semantics of framing and how to well formalize framing in a satisfiable and practicable manner. Within ITL community, Duan, Holt and Maciej [5] first bring forward a framing technique through an explicit operator ($\mathsf{frame}(x)$), which enables us to establish a flexible framed environment where framed and non-framed variables can be mixed, with frame operators being used in sequential, conjunctive and parallel manner, and an executable version of framed temporal logic programming language is developed [7, 6, 23].

The key characteristic of the frame operator can be stated as:

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

To identify an occurrence of an assignment to a variable, say x, we make use of a so-called *assignment flag*, denoted by a predicate $\mathsf{af}(x)$; it is true whenever an assignment of a value to x is encountered, and false otherwise. In order to give its definition, a new assignment called *positive immediate assignment* $x \Leftarrow e$ is needed and defined as $x \Leftarrow e \stackrel{\text{def}}{=} x = e \wedge p_x$, where p_x is an atomic proposition connected with variable x and cannot be used for other purposes. Thus the definition of the assignment flag is $\mathsf{af}(x) \stackrel{\text{def}}{=} p_x$ for every variable x. The predicate $\mathsf{af}(x)$ is associated with the 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 — $\mathsf{af}(x)$, and certainly p_x, must be true whenever such an assignment occurs. On the other side, when $\mathsf{af}(x)$ is true, an assignment must have been perceived in the execution of the program. however, if there is no assignment to x, p_x is unspecified, and in this case, we will use a *minimal model* to force it to be false.

To define $\mathsf{frame}(x)$, another state frame $\mathsf{lbf}(x)$ is introduced:

$$\mathsf{lbf}(x) \stackrel{\text{def}}{=} \neg\mathsf{af}(x) \rightarrow \exists\, b \;:\; (\ominus x = b \wedge x = b) \qquad \mathsf{frame}(x) \stackrel{\text{def}}{=} \Box(\mathsf{more} \rightarrow \bigcirc\, \mathsf{lbf}(x))$$

where b is a static variable. Intuitively, $\mathsf{lbf}(x)$ means that, when a variable is framed at a state, its value remains unchanged (same as at the previous state) if no assignment occurs at that state. We often call a program is framed if it contains $\mathsf{lbf}(x)$ or $\mathsf{frame}(x)$.

A faithful interpretation of framed programs is based on their minimal models and we briefly introduce the definition of minimal models. Details can be found in [4, 7, 6].

In logic programming languages, canonical models [2] have been used to interpret programs. A canonical interpretation on propositions is a set $I \subseteq \mathcal{P}rop$, and all propositions not in I are false. Now let $\sigma = \langle (I^0_{var}, I^0_{prop}), (I^1_{var}, I^1_{prop}), \dots \rangle$ be a model. We denote the sequence of interpretation on propositions of σ by $\sigma_{prop} = \langle I^0_{prop}, I^1_{prop}, \dots \rangle$, and call σ_{prop} canonical if each I^i_{prop} is a canonical interpretation on propositions. Moreover, if $\sigma' = \langle (I'^0_{var}, I'^0_{prop}), (I'^1_{var}, I'^1_{prop}), \dots \rangle$ is another canonical model, then we denote:

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

A framed program can be satisfied by several different canonical models, among which minimal models capture precisely the meanings of the framed program.

Definition 1 (Minimal Satisfaction Relation [6]) Let Q be a framed program, and $\Im = (\sigma, i, k, j)$ be a canonical interpretation. Then the minimal satisfaction relation $\models_m$ is defined as $(\sigma, i, k, j) \models_m Q$ iff $(\sigma, i, k, j) \models_{itl} Q$ and there is no σ' such that $\sigma' \sqsubset \sigma$ and $(\sigma', i, k, j) \models_{itl} Q$.

In the sequel, by the satisfaction relation $\models_{itl}$ we actually mean the minimal satisfaction relation $\models_m$ by default.

The framing technique is better illustrated by examples. Consider the formula $\mathsf{frame}(x) \wedge (x \Leftarrow 1) \wedge \mathsf{len}(1)$, and by convention we use the pair $(\{x : e\}, \{p_x\})$ to express the state formula $x = e \wedge p_x$ at every state. It can be checked that the formula has the following canonical models:

$$\sigma_1 = \langle (\{x{:}1\}, \{p_x\}), (\varnothing, \{p_x\}) \rangle \qquad \sigma_2 = \langle (\{x{:}1\}, \{p_x\}), (\{x{:}1\}, \varnothing) \rangle$$

The canonical model does not specify p_x at state s_1 — it can be True too, but taking True as the value of p_x causes x to be unspecified as in σ_1, which means x_1 can be an arbitrary value of its domain. The intended meaning of the formula is indeed captured by its minimal model, in this case σ_2: with this model, x is defined in both states of the interval — the value is 1 at both states.

3 Temporal Logic ITL$^+$

3.1 The Logic ITL$^+$

ITL$^+$ is defined based on the interval temporal logic ITL with formulas defined by the following grammar (ITL$^+$ terms are as in ITL):

$$\text{ITL}^+ \text{ Formulas:} \quad p, q, \ldots ::= \ldots \mid \langle Q \rangle \mid \neg p \mid p \wedge q \mid \bigcirc p \mid p \,;\, q \mid \exists x.p \mid p^+$$

where Q is an ITL formula. The novel construct $\langle \ldots \rangle$ is used to specify atomic blocks with arbitrary granularity in concurrency and we call $\langle Q \rangle$ an *atomic interval formulas*. All other constructs are as in ITL except that they can take atomic interval formulas in ITL$^+$. In particular, a valid ITL formulas is also a valid ITL$^+$ formula.

Notice that formulas inside atomic blocks $\langle \ldots \rangle$ must be an ITL formulas — we do not allow nested atomic blocks. The logic can be probably extended further to allow nested atomicity by enforcing a strict tier over atomic blocks, but it is beyond the discussion of the paper.

The essence of atomic execution are twofold: first, the concrete execution of the code block inside an atomic wrapper can take multiple state transitions; second, nobody out of the atomic block can see the internal states. This leads to an interpretation of

atomic interval formulas based on two levels of intervals — at the outer level an atomic interval formula $\langle Q \rangle$ always specify a single transition between two consecutive states, which the formula Q (which is an ITL formula) will be interpreted at another interval (the inner level), which we call an *atomic interval* and must be finite, with only the first and final states being exported to the outer level. The key point of such a two-level interval based interpretation is the exportation of values which are computed inside atomic blocks. We shall show how framing technique helps at this aspect.

A few notations are introduced to support formalizing the semantics of atomic interval formulas.

- Given an ITL formula Q, let V_Q be the set of free variables of Q. we define another ITL formula $\mathsf{FRM}(V_Q)$ as follows:

$$\mathsf{FRM}(V_Q) \stackrel{\text{def}}{=} \begin{cases} \bigwedge_{x \in V_Q} \mathsf{frame}(x) & \text{if } V_Q \neq \emptyset \\ \mathsf{True} & \text{otherwise} \end{cases}$$

  $\mathsf{FRM}(V_Q)$ says that each variable in the set V_Q is a framing variable that allows to inherit the old value from previous states. $\mathsf{FRM}(V_Q)$ is essentially used to apply the framing technique within atomic blocks, and allows values to be carried throughout an atomic block to its final state, which will be exported.
- Interval concatenation is defined by

$$\sigma \cdot \sigma' = \begin{cases} \sigma & \text{if } |\sigma| = \omega \text{ or } \sigma' = \epsilon \\ \sigma' & \text{if } \sigma = \epsilon \\ \langle\!\langle s_0, ..., s_i, s_{i+1}, ... \rangle\!\rangle & \text{if } \sigma = \langle\!\langle s_0, ..., s_i \rangle\!\rangle \text{ and } \sigma' = \langle\!\langle s_{i+1}, ... \rangle\!\rangle \end{cases}$$

- If $s = (I_{var}, I_{prop})$ is a state, we write $s|_{I'_{prop}}$ for the state (I_{var}, I'_{prop}), which has the same interpretation for normal variables as s but a different interpretation I'_{prop} for propositions. Given two intervals σ, σ', we write $\sigma \stackrel{x}{=} \sigma'$ if $|\sigma| = |\sigma'|$ and for every variable $y \in \mathcal{V}/\{x\}$ and every proposition $p \in \mathcal{P}rop$, $s_k[y] = s'_k[y]$ and $s_k[p] = s'_k[p]$, where s_k, s'_k are the k-th states in σ and σ' respectively, for all $0 \leq k \leq |\sigma|$.

The semantics of ITL$^+$ formulas, in the form of $(\sigma, i, k, j) \models p$, is defined in Figure 3 (semantics of terms and omitted formulas can be obtained from Figure 1). The interpretation of atomic interval formulas is the key novelty of the paper, which represents exactly the semantics of atomic executions: the entire formula $\langle Q \rangle$ specifies a single state transition (from s_k to s_{k+1}), and the real execution of the block, which represented by Q, can be carried over a finite interval (of arbitrary length), with the first and final state connected to s_k and s_{k+1} respectively — the internal states of Q (states of σ') are hidden from outside the atomic block.. Notice that when interpreting Q, we use $\mathsf{FRM}(V_Q)$ to carry values through the interval, however this may lead to conflict interpretations to some primitive propositions (basically assignment flags for variables that take effect both outside and inside atomic blocks), hence the final state is not exactly s_{k+1} but $s_{k+1}|_{I'_{prop}}$ with a different interpretation of primitive propositions. We shall explain the working details of framing in atomic blocks by examples.

Figure 4 presents a set of useful ITL formulas that we shall frequently use in the rest of the paper:

$$
\begin{array}{lll}
(\sigma, i, k, j) \models \langle Q \rangle & \text{iff} & \text{there exists a finite interval } \sigma' \text{ and } I'_{prop} \text{ such that} \\
 & & \langle\!\langle s_k \rangle\!\rangle \cdot \sigma' \cdot \langle\!\langle s_{k+1}|_{I'_{prop}} \rangle\!\rangle \models_{itl} Q \wedge \mathsf{FRM}(V_Q) \\
(\sigma, i, k, j) \models \bigcirc p & \text{iff} & (\sigma, i, k+1, j) \models p, \text{ if } k < j \\
\Im \models p \,;\, q & \text{iff} & \text{there exists } r \text{ such that } (\sigma, i, k, r) \models p \text{ and } (\sigma, r, r, j) \models q \\
\Im \models \neg p & \text{iff} & \Im \not\models p \\
\Im \models p \wedge q & \text{iff} & \Im \models p \text{ and } \Im \models q \\
(\sigma, i, k, j) \models \exists x.p & \text{iff} & \text{there exists } \sigma' \text{ such that } \sigma' \stackrel{x}{=} \sigma \text{ and } (\sigma', i, k, j) \models p \\
(\sigma, i, k, j) \models p^+ & \text{iff} & \text{if there are } i = r_0 \leq r_1 \leq \ldots \leq r_{n-1} \leq r_n = j \ (n \geq 1) \text{ such that} \\
 & & (\sigma, r_0, k, r_1) \models p \text{ and } (\sigma, r_{l-1}, r_{l-1}, r_l) \models p \ (1 < l \leq n)
\end{array}
$$

Fig. 3. Semantics for ITL$^+$ Formulas.

$$
\begin{array}{lll}
\Box p \stackrel{\text{def}}{=} \neg\Diamond\neg p & \Diamond p \stackrel{\text{def}}{=} \mathsf{True} \,;\, p & p^* \stackrel{\text{def}}{=} p^+ \vee \varepsilon \\
\odot p \stackrel{\text{def}}{=} \bigcirc p \vee \varepsilon & p \supset q \stackrel{\text{def}}{=} \Box(p \rightarrow q) & p \equiv q \stackrel{\text{def}}{=} \Box(p \leftrightarrow q)
\end{array}
$$

Fig. 4. Some Derived ITL$^+$ Formulas.

- $\Box p$ (*always p*): p holds at every state after (including) the current state;
- $\Diamond p$ (*sometimes p*): p eventually holds at some state after (including) the current state;
- p^* (*chop star*): either chop plus p^+ or ε;
- $\odot p$: either $\bigcirc p$ or ε;
- $p \equiv q$ (*strong equivalence*): p and q have the same truth value in all states of every model;
- $p \supset q$ (*strong implication*) : $p \rightarrow q$ always hold in all states of every model.

In the ITL programming model, we support two minimum execution unit defined as follows. They are used to construct *normal forms*[7, 6] of program executions.

State formulas. State formulas in ITL are defined as follows:.

$$ps ::= x = e \mid x \Leftarrow e \mid ps \wedge ps \mid \mathsf{lbf}(x) \mid \mathsf{True}$$

We consider True as a state formula since every states satisfies True. Note that $\mathsf{lbf}(x)$ is also a state formula when $\ominus x$ is evaluated to a value. The state frame $\mathsf{lbf}(x)$, which denotes a special assignment that keeps the old value of a variable if there are no new assignment to the variable, enables the inheritance of values from the previous state to the current state. Apart from the general assignments $x = e$, in the paper we also allow assignments such like $\mathsf{lbf}(x) \wedge x \Leftarrow e$ and $\mathsf{lbf}(x) \wedge x = e$.

Proof: In the following, we prove some logic laws in Fig.5 regarding atomic interval formulas and other logic laws can be proved in a similar way as in ITL.

$(L1)$ $\bigcirc p \equiv \odot p \wedge \mathsf{more}$
$(L2)$ $\bigcirc p \supset \mathsf{more}$
$(L3)$ $\bigcirc(p \wedge q) \equiv \bigcirc p \wedge \bigcirc q$
$(L4)$ $\bigcirc(p \vee q) \equiv \bigcirc p \vee \bigcirc q$
$(L5)$ $\bigcirc(\exists x : p) \equiv \exists x : \bigcirc p$
$(L6)$ $\Box p \equiv p \wedge \odot \Box p$
$(L7)$ $\Box p \wedge \varepsilon \equiv p \wedge \varepsilon$
$(L8)$ $\Box p \wedge \mathsf{more} \equiv p \wedge \bigcirc \Box p$
$(L9)$ $(p \vee q)\ ;\ ps \equiv (p\ ;\ ps) \vee (q\ ;\ ps)$
$(L10)$ $(ps \wedge p)\ ;\ q \equiv ps \wedge (p\ ;\ q)$
$(L11)$ $\bigcirc p\ ;\ q \equiv \bigcirc(p\ ;\ q)$
$(L12)$ $\varepsilon\ ;\ q \equiv q$
$(L13)$ $\langle Q_1 \vee Q_2 \rangle \equiv \langle Q_1 \rangle \vee \langle Q_2 \rangle$
$(L14)$ $\langle Q_1 \rangle \equiv \langle Q_2 \rangle$, *if* $Q_1 \equiv Q_2$
$(L15)$ $(p_1 \vee p_2)\ ;\ q \equiv (p_1\ ;\ q) \vee (p_2\ ;\ q)$
$(L16)$ $\exists x : p(x) \equiv \exists y : p(y)$
$(L17)$ $x = e \equiv (p_x \wedge x = e) \vee (\neg p_x \wedge x = e)$
$(L18)$ $\mathsf{lbf}(x) \equiv p_x \vee (\neg p_x \wedge (x = \ominus x))$
$(L19)$ $\mathsf{lbf}(x) \wedge x = e \equiv x \Leftarrow e \ \ (\ominus x \neq e)$
$(L20)$ $\mathsf{lbf}(x) \wedge x \Leftarrow e \equiv x \Leftarrow e$
$(L21)$ $\mathsf{frame}(x) \wedge \mathsf{more} \equiv \bigcirc(\mathsf{frame}(x) \wedge \mathsf{lbf}(x))$
$(L22)$ $\mathsf{frame}(x) \wedge \varepsilon \equiv \varepsilon$
$(L23)$ $(Qs \wedge \bigcirc p_1)\ ;\ p_2 \equiv Qs \wedge \bigcirc(p_1\ ;\ p_2)$
$(L24)$ $Q_1 \wedge \langle Q \rangle \equiv Q_2 \wedge \langle Q \rangle$, if $Q_1 \equiv Q_2$
$(L25)$ $\langle Q \rangle \equiv \langle \mathsf{frame}(x) \wedge Q \rangle$ x is a free variale in Q
$(L26)$ $\bigcirc e_1 + \bigcirc e_2 = \bigcirc(e_1 + e_2)$

Fig. 5. Some ITL$^+$ Laws

The proof of (L13):

$$
\begin{array}{ll}
 & (\sigma, i, k, j) \models \langle Q_1 \vee Q_2 \rangle \\
\Longleftrightarrow & \exists \sigma', \sigma'', I'_{prop} \text{ s.t. } \sigma'' = \langle\!\langle s_k \rangle\!\rangle \cdot \sigma' \cdot \langle\!\langle s_{k+1}|_{I'_{prop}} \rangle\!\rangle \text{ and} \\
 & \sigma'' \models_{itl} (Q_1 \vee Q_2) \wedge \mathsf{FRM}(V_{Q_1 \vee Q_2}) \\
\Longleftrightarrow & \sigma'' \models_{itl} (Q_1 \wedge \mathsf{FRM}(V_{Q_1})) \vee (Q_2 \wedge \mathsf{FRM}(V_{Q_2})) \\
\Longleftrightarrow & \sigma'' \models_{itl} Q_1 \wedge \mathsf{FRM}(V_{Q_1}) \text{ or } \sigma'' \models_{itl} Q_2 \wedge \mathsf{FRM}(V_{Q_2}) \\
\Longleftrightarrow & (\sigma, i, k, j) \models \langle Q_1 \rangle \text{ or } (\sigma, i, k, j) \models \langle Q_2 \rangle \\
\Longleftrightarrow & (\sigma, i, k, j) \models \langle Q_1 \rangle \vee \langle Q_2 \rangle
\end{array}
$$

The proof of (L14):

$$
\begin{array}{ll}
 & (\sigma, i, k, j) \models \langle Q_1 \rangle \\
\Longleftrightarrow & \exists \sigma', \sigma'', I'_{prop} \text{ s.t. } \sigma'' = \langle\!\langle s_k \rangle\!\rangle \cdot \sigma' \cdot \langle\!\langle s_{k+1}|_{I'_{prop}} \rangle\!\rangle \text{ and} \\
 & \sigma'' \models_{itl} Q_1 \wedge \mathsf{FRM}(V_{Q_1}) \\
\Longleftrightarrow & \sigma'' \models_{itl} Q_2 \wedge \mathsf{FRM}(V_{Q_2}) \quad (Q_1 \equiv Q_2) \\
\Longleftrightarrow & (\sigma, i, k, j) \models \langle Q_2 \rangle
\end{array}
$$

The proof of (L23):

$$
\begin{array}{ll}
 & (\sigma, i, k, j) \models (Qs \wedge \bigcirc p_1)\ ;\ p_2 \\
\Longleftrightarrow & \exists r \text{ s.t. } (\sigma, i, k, r) \models Qs \wedge \bigcirc p_1 \text{ and } (\sigma, r, r, j) \models p_2 \\
\Longleftrightarrow & (\sigma, i, k, r) \models Qs \text{ and } (\sigma, i, k+1, r) \models p_1 \text{ and } (\sigma, r, r, j) \models p_2 \\
\Longleftrightarrow & (\sigma, i, k, r) \models Qs \text{ and } (\sigma, i, k+1, j) \models p_1\ ;\ p_2 \\
\Longleftrightarrow & (\sigma, i, k, r) \models Qs \text{ and } (\sigma, i, k, j) \models \bigcirc(p_1\ ;\ p_2) \\
\Longleftrightarrow & (\sigma, i, k, j) \models Qs \text{ and } (\sigma, i, k, j) \models \bigcirc(p_1\ ;\ p_2) \\
\Longleftrightarrow & (\sigma, i, k, j) \models Qs \wedge \bigcirc(p_1\ ;\ p_2)
\end{array}
$$

The proof of (L24):

$$\begin{aligned}
& (\sigma, i, k, j) \models Q_1 \wedge \langle Q \rangle \\
\iff & (\sigma, i, k, j) \models Q_1 \text{ and } (\sigma, i, k, j) \models \langle Q \rangle \\
\iff & (\sigma, i, k, j) \models Q_2 \text{ and } (\sigma, i, k, j) \models \langle Q \rangle \ \ (Q_1 \equiv Q_2) \\
\iff & (\sigma, i, k, j) \models Q_2 \wedge \langle Q \rangle
\end{aligned}$$

The proof of (L25):

$$\begin{aligned}
& (\sigma, i, k, j) \models \langle Q \rangle \\
\iff & \exists \sigma', \sigma'', I'_{prop} \text{ s.t. } \sigma'' = \langle\!\langle s_k \rangle\!\rangle \cdot \sigma' \cdot \langle\!\langle s_{k+1}|_{I'_{prop}} \rangle\!\rangle \text{ and} \\
& \sigma'' \models_{itl} Q \wedge \mathsf{FRM}(V_Q) \\
\iff & \sigma'' \models_{itl} \mathsf{frame}(x) \wedge Q \wedge \mathsf{FRM}(V_Q) \ \ (\ x \text{ is a free variable in } Q.) \\
\iff & (\sigma, i, k, j) \models \langle \mathsf{frame}(x) \wedge Q \rangle
\end{aligned}$$

Extended state formulas. Since atomic interval formulas are introduced in ITL$^+$, we extend the notion of state formulas to include atomic interval formulas:

$$Qs ::= ps \mid \langle Q \rangle \mid Qs \wedge Qs,$$

where Q is arbitrary ITL formulas and ps are state formulas.

In Figure 5, we present a collection of logic laws that are useful for reasoning about ITL$^+$ formulas. For any non-atomic interval formula p in ITL$^+$, it has the same meanings under the satisfaction relations $\models$ and $\models_{itl}$, that is, $(\sigma, i, k, j) \models p$ iff $(\sigma, i, k, j) \models_{itl} p$.

3.2 Support Framing in Atomic Interval Formulas

Framing is a powerful technique that carries values through over interval states in logical programming with ITL formulas and it is also used inside atomic blocks in ITL$^+$. While in ITL frames are usually explicitly specified by users in ITL, we have made it inherent in the semantics of atomic interval formulas in ITL$^+$, which is the role of $\mathsf{FRM}(V_Q)$. However, framing inside atomic blocks must be carefully manipulated, as we use same propositions (assignment flags) to track the modification of variables outside and inside atomic blocks — conflict interpretations of assignment flags may occur at the exit of atomic blocks.

We demonstrate this by the following example:

$$(\sigma, 0, 0, |\sigma|) \models \mathsf{FRM}(\{x\}) \wedge x = 1 \wedge \langle Q \rangle \quad \text{where } Q \stackrel{\text{def}}{=} \bigcirc x = 2 \wedge \mathsf{len}(2)$$

Q requires the length of the atomic interval be 2. Inside the atomic block, which is interpreted at the atomic interval, $\mathsf{FRM}(\{x\})$ will record that x is updated with 2 at the second state and remains unchanged till the last state. The detailed interpretation is given in Figure 6.

When exiting the atomic block (at the state s_1 of σ), we need to merge the updating information with that obtained from $\mathsf{FRM}(\{x\})$ outside the block, which inherits the previous value 1 with the assignment flag denoting no assignment to x. This conflicts the value sent out from the atomic block, which says that the value of x is 2, with non-assignment flag as well. In Figure 6, we highlight the place where conflicts occur. We

$$\mathsf{FRM}(\{x\}) \wedge x = 1 \wedge \langle Q \rangle \wedge \mathsf{len}(1) \quad \text{Where } Q \stackrel{\text{def}}{=} \bigcirc x = 2 \wedge \mathsf{len}(2)$$

$$\equiv \mathsf{frame}(x) \wedge x = 1 \wedge \langle \bigcirc x = 2 \wedge \mathsf{len}(2) \rangle \wedge \bigcirc \varepsilon$$

($L21$) $$\equiv \bigcirc(\mathsf{frame}(x) \wedge \mathsf{lbf}(x)) \wedge x = 1 \wedge \langle \bigcirc(\mathsf{frame}(x) \wedge \mathsf{lbf}(x)) \wedge \bigcirc x = 2 \wedge \bigcirc\bigcirc\varepsilon \rangle \wedge \bigcirc\varepsilon$$

($L18$) $$\equiv x = 1 \wedge \bigcirc(\mathsf{frame}(x) \wedge ((x = 1 \wedge \neg p_x) \vee p_x) \wedge \varepsilon) \wedge \langle \bigcirc(\mathsf{frame}(x) \wedge ((x = 1 \wedge \neg p_x) \vee p_x)) \wedge \bigcirc x = 2 \wedge \bigcirc\bigcirc\varepsilon \rangle$$

($L19$) $$\equiv \ldots \wedge \langle \bigcirc(\mathsf{frame}(x) \wedge x = 2 \wedge p_x \wedge \bigcirc\varepsilon) \rangle \wedge \bigcirc\varepsilon$$

($L21$) $$\equiv \ldots \langle \bigcirc(\bigcirc(\mathsf{frame}(x) \wedge \mathsf{lbf}(x)) \wedge x = 2 \wedge p_x \wedge \bigcirc\varepsilon) \rangle \wedge \bigcirc\varepsilon$$

($L18$) $$\equiv \ldots \wedge \langle \bigcirc(\bigcirc(\mathsf{frame}(x) \wedge ((x = 2 \wedge \neg p_x) \vee p_x)) \wedge x = 2 \wedge p_x \wedge \bigcirc\varepsilon) \rangle \wedge \bigcirc\varepsilon$$

($L17$) $$\equiv (x = 1 \wedge p_x) \vee (x = 1 \wedge \neg p_x) \wedge \bigcirc((x = 1 \wedge \neg p_x) \vee p_x) \wedge \langle \bigcirc(x = 2 \wedge p_x) \wedge \bigcirc\bigcirc(((x = 2 \wedge \neg p_x) \vee p_x)) \wedge \bigcirc\bigcirc\varepsilon \rangle \wedge \bigcirc\varepsilon$$

$$\equiv (x = 1 \wedge \neg p_x) \wedge \bigcirc((x = 1 \wedge \neg p_x) \vee p_x) \quad \text{(conflicts occur !)}$$
$$\wedge \langle \bigcirc(x = 2 \wedge p_x) \wedge \bigcirc\bigcirc(x = 2 \wedge \neg p_x) \wedge \bigcirc\bigcirc\varepsilon \rangle \wedge \bigcirc\varepsilon$$

(Def. *in Fig* 3) $$\equiv (x = 1 \wedge \neg p_x) \wedge \bigcirc((x = 1 \wedge \neg p_x) \vee p_x) \wedge \bigcirc(x = 2) \wedge \bigcirc\varepsilon$$
I'_{prop} used inside atomic blocks is existentially quantified

(Def.1) $$\equiv (x = 1 \wedge \neg p_x) \wedge \bigcirc(x = 2 \wedge p_x) \wedge \bigcirc\varepsilon$$

Fig. 6. Resolving Framing Conflicts in Atomic Interval Formulas

solve this problem by adopting different interpretation of p_x (the assignment proposition for x) at the exporting state (s_1 in σ and s''_2 in σ''): outside the atomic block (at s_1) p_x in interpreted as True while inside the block (at $s''_2 = s_1|_{I'_{prop}}$) p_x is False, which is consistent with the interpretation of Q.

4 Programming with ITL$^+$

We use a subset of ITL$^+$ as our programming language, where for concurrent programming, we extend ITL programming language with atomic interval formulas and an *interleaving operator*, which captures the interleaving between processes with atomic blocks.

4.1 Expressions and Statements

Expressions. ITL$^+$ provides permissible arithmetic expressions and boolean expressions and both are basic terms of ITL$^+$.

Arithmetic expressions: $e ::= n \mid x \mid \bigcirc x \mid \ominus x \mid e_0 \ \mathsf{op} \ e_1$
Boolean expressions: $b ::= \mathsf{True} \mid \mathsf{False} \mid \neg b \mid b_0 \wedge b_1 \mid e_0 = e_1 \mid e_0 < e_1$

where n is an integer, x is a program variable, op represents common arithmetic operations, $\bigcirc x$ and $\ominus x$ mean that x is evaluated over the next and previous state respectively.

Termination :	ε	Assignment :	$x = e$
Positive Assignment :	$x \Leftarrow e$	State Frame :	$\mathsf{lbf}(x)$
Interval Frame :	$\mathsf{frame}(x)$	Conjuction statement :	$p \wedge q$
Selection statement:	$p \vee q$	Next statement :	$\bigcirc p$
Sequential statement :	$p \,;\, q$	Conditional statement :	$\texttt{if}\ b\ \texttt{then}\ p\ \texttt{else}\ q$
Existential quantification :	$\exists x : p(x)$	While statement :	$\texttt{while}\ b\ \texttt{do}\ p$
Atomic block :	$\langle Q \rangle$	Interleaving statement :	$p \,\|\|\|\, q$

Fig. 7. Statements in ITL$^+$

Statements. Figure 7 shows the statements of ITL$^+$, where $p, q, \ldots$ are ITL$^+$ formulas and Q is an ITL formula. ε means termination on the current state; $x = e$ represents assignment over the current state or boolean conditions; $x \Leftarrow e$, $\mathsf{lbf}(x)$ and $\mathsf{frame}(x)$ support framing mechanism and are discussed in Section 3.2; $p \wedge q$ means that the processes p and q are executed concurrently and they share all the states and variables during the execution; $p \vee q$ represents selection statements; $\bigcirc p$ means that p holds at the next state; $p \,;\, q$ means that p holds at every state from the current one till some state in the future and from that state on p holds; $\langle Q \rangle$ executes Q atomically.

The conditional and while statements are defined as below:

$$\begin{aligned} &\texttt{if}\ b\ \texttt{then}\ p\ \texttt{else}\ q \stackrel{\text{def}}{=} (b \wedge p) \vee (\neg b \wedge q) \\ &\texttt{while}\ b\ \texttt{do}\ p \quad\quad \stackrel{\text{def}}{=} (p \wedge b)^* \wedge \Box(\varepsilon \rightarrow \neg b) \end{aligned}$$

$\texttt{if}\ b\ \texttt{then}\ p\ \texttt{else}\ q$ first evaluates the boolean expression: if b is True, then the process p is executed, otherwise q is executed. The while statement $\texttt{while}\ b\ \texttt{do}\ p$ 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 is false, then the `while` statement terminates. The interleaving statement $p \,|||\, q$ is used to execute atomic blocks of concurrent threads p and q sequentially in a non-deterministic way, and its formal definition will be given in Section 4.3. We use a renaming method to reduce a program with existential quantification. Given a formula $\exists x : p(x)$ with a bound variable x, we can remove the existential quantification $(\exists x)$ from $\exists x : p(x)$ 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 $(q \wedge \exists x : p(x))$;
- 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 1 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 the definition in Fig.3, given a model σ, the following is true:

$$\sigma \models \exists y : p(y) \text{ iff there exists } \sigma', \sigma \stackrel{y}{=} \sigma', \text{ and } \sigma' \models p(y)$$

Thus, if $\sigma \models \exists y : p(y)$ then there is a σ', $\sigma' \models p(y)$. Conversely, for a model σ, if $\sigma \models p(y)$, then $\sigma \models \exists y : p(y)$ because σ 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, $\exists y : p(y) \equiv \exists x : p(x)$. Hence, Lemma 1 is true.

From Lemma 1, 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. In the following, we give a definition of *x-equivalent* programs w.r.t. $\exists x : p(x)$.

Definition 2 For a variable x, $\exists x : p(x)$ and $p(x)$ are called x-equivalent, denoted by $\exists x : p(x) \overset{x}{\equiv} p(x)$, if for each model $\sigma \models \exists x : p(x)$, there exists some σ', $\sigma \overset{x}{=} \sigma'$ and $\sigma' \models p(x)$ and vice versa.

Definition 2 tells us that the models of $\exists x : p(x)$ and $p(x)$ are the same ignoring variable x. Thus, we denote such the special equivalence relation between programs as $\overset{x}{\equiv}$, which is the counterpart of the equivalence relation $\overset{x}{=}$ over the intervals. Notice that, $\exists x : p(x) \equiv \exists y : p(y)$, and by Definition 2, $\exists y : p(y) \overset{y}{\equiv} p(y)$; so we say that $\exists x : p(x) \overset{y}{\equiv} p(y)$ 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 1 Hiding local variable using existential quantifiers.

$$Q_1 \overset{\text{def}}{=} x = 1 \wedge \exists x : (x = 2 \wedge y := 3 * x) \quad Q_2 \overset{\text{def}}{=} x = 1 \wedge (z = 2 \wedge y := 3 * z)$$

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 Q_1. Thus, we have $Q_1 \overset{z}{\equiv} Q_2$. Moreover, by Definition 2, the model of Q_1 can be obtained by hiding variable z in the model of Q_2. Therefore, we have the model of Q_2 is $\sigma_2 = \langle s_0, s_1 \rangle = \langle(\{x : 1, z : 2\}), (\{y : 6\})\rangle$; and the model of Q_1 is $\sigma_1 = \langle s_0, s_1 \rangle = \langle(\{x : 1\}), (\{y : 6\})\rangle$.

In order to avoid an excessive number of parentheses, the priority level of the interleaving operator is the lowest in ITL$^+$.

4.2 Semi-normal Forms

In the ITL$^+$ programming model, since programs are represented as formulas, the execution of programs can be treated as a kind of formulas reduction. Target programs are obtained by rewriting the original programs with logic laws presented in Figure 5, and the valid of laws ensures that original and target programs are logically equivalent. Usually, our target programs should be represented as ITL$^+$ formulas in *normal forms*.

In this section, we shall describe how ITL$^+$ programs can be transformed into their normal forms. Since we have both state formulas and atomic interval formulas as minimum execution units, we use a different regular form of formulas called *semi-normal form* (SNF for short, see Definition 3) to define the interleaving semantics, and it provides an intermediate form for transforming ITL$^+$ programs into normal forms.

Definition 3 (Semi-Normal Form) An ITL$^+$ program is *semi-normal* if it has the following form

$$(\bigvee_{i=1}^{n_1} Qs_{ci} \wedge \bigcirc p_{fi}) \vee (\bigvee_{j=1}^{n_2} Qs_{ej} \wedge \varepsilon)$$

where Qs_{ci} and Qs_{ej} are extended state formulas for all i, j, p_{fi} is an ITL$^+$ program, $n_1 + n_2 \geq 1$ and $\bigvee_{i=1}^{n} Qs_{ei} \wedge \varepsilon$ is a shorthand for $(Qs_{e1} \wedge \varepsilon) \vee \ldots \vee (Qs_{en2} \wedge \varepsilon)$. P_{fi} is an ITL$^+$ program, in which variables may refer to their previous states but not beyond the first state of the current interval over which the program is executed. If a program terminates at the current state, then it is transformed to $\bigvee_{j=1}^{n_2} Qs_{ej} \wedge \varepsilon$; otherwise, it is transformed to $\bigvee_{i=1}^{n_1} Qs_{ci} \wedge \bigcirc p_{fi}$.

For convenience, we often call $(\bigvee_{i=1}^{n_1} Qs_{ci} \wedge \bigcirc p_{fi})$ *future formulas* and $(Qs_{ci} \wedge \bigcirc p_{fi})$ *single future formulas*; whereas $(\bigvee_{j=1}^{n_2} Qs_{ej} \wedge \varepsilon)$ *terminal formulas* and $(Qs_{ej} \wedge \varepsilon)$ *single terminal formulas*.

Example 2 Some examples of semi-normal forms are given as below:

(1) $x = 1 \wedge \langle x := x + 1 \rangle \wedge \bigcirc(y = 2 \wedge \mathsf{skip})$

(2) $x = 1 \wedge y = 2 \wedge \varepsilon$ (3) $x = 1 \wedge \bigcirc(y := 2)$

In (1), the semi-normal form consists of an extended state formula which contains an atomic interval formula, and the next program. In (2) and (3), semi-normal forms are given by the state formulas associated with the termination or the next program.

Theorem 2 For any ITL$^+$ formula p, if p does not contain any interleaving operators, then there exists an ITL$^+$ formula p', such that $p \equiv p'$ and p' is in *semi-normal form*.

Proof: The proof proceeds by induction on the structure of ITL$^+$ statements. Note that program p does not contain interleaving operators.
(1) If p is state formula ps including $x = e$, $x \Leftarrow e$, $\mathsf{lbf}(x)$:

$$\begin{aligned} ps &\equiv ps \wedge \mathsf{True} \\ &\equiv ps \wedge (\mathsf{more} \vee \varepsilon) \\ &\equiv ps \wedge \mathsf{more} \vee ps \wedge \varepsilon \\ &\equiv ps \wedge \bigcirc\mathsf{True} \vee ps \wedge \varepsilon \quad (\mathsf{more} \stackrel{\text{def}}{=} \bigcirc\mathsf{True}) \end{aligned}$$

(2) If p is the termination statement ε, the conclusion is straightforward.

(3) If p is a statement in the form $\bigcirc p$, it is already in its semi-normal form.

(4) If p is the atomic block $\langle Q \rangle$:

$$\begin{aligned} & (\sigma, i, k, j) \models \langle Q \rangle \\ \Longleftrightarrow\ & \exists \sigma', \sigma'', I'_{prop} \text{ s.t. } \sigma'' = \langle\!\langle s_k \rangle\!\rangle \cdot \sigma' \cdot \langle\!\langle s_{k+1}|_{I'_{prop}} \rangle\!\rangle \text{ and } \sigma'' \models_{itl} Q \wedge \mathsf{FRM}(V_Q) \\ \Longleftrightarrow\ & (\sigma, i, k, j) \models ps_1 \wedge \bigcirc ps'_2 \\ & \text{where } ps_1 \text{ and } ps_2 \text{ be state formulas of } Q \wedge \mathsf{FRM}(V_Q), \text{ which hold at state } s_k \\ & \text{and state } \langle\!\langle s_{k+1}|_{I'_{prop}} \rangle\!\rangle \text{ respectively, and } ps'_2 = \bigwedge_{i=1}^{n}(x_i = e_i) \text{ or } \mathsf{True}. \end{aligned}$$

(5) If p is the conjunction statement $p \wedge q$, then, by the hypothesis, we have, $p \equiv w \wedge \varepsilon \vee w' \wedge \bigcirc p'$ and $q \equiv u \wedge \varepsilon \vee u' \wedge \bigcirc q'$, where w, w', u, u' are extended state formulas.

$$\begin{aligned} & p \wedge q \\ \equiv\ & (w \wedge \varepsilon \vee w' \wedge \bigcirc p') \wedge (u \wedge \varepsilon \vee u' \wedge \bigcirc q') \\ \equiv\ & (w \wedge u \wedge \varepsilon) \vee (w' \wedge u' \wedge \bigcirc(p' \wedge q')) \end{aligned}$$

(6) If p is a sequential statement $p \,;\, q$, then, by hypothesis, we have, $p \equiv w \wedge \varepsilon \vee w' \wedge \bigcirc p'$, $q \equiv u \wedge \varepsilon \vee u' \wedge \bigcirc q'$, where w, w' and u, u' are extended state formulas.

$$\begin{aligned} & p \,;\, q \\ \equiv\ & (w \wedge \varepsilon \vee w' \wedge \bigcirc p') \,;\, q \\ \equiv\ & ((w \wedge \varepsilon) \,;\, q) \vee ((w' \wedge \bigcirc p') \,;\, q) \quad (L15) \end{aligned}$$

If w is state formulae, we have:

$$\begin{aligned} & ((w \wedge \varepsilon) \,;\, q) \vee ((w' \wedge \bigcirc p') \,;\, q) \\ \equiv\ & (w \wedge q) \vee (w' \wedge \bigcirc(p' \,;\, q)) \ \ (L10, L12, L23) \\ \equiv\ & (w \wedge (u \wedge \varepsilon \vee u' \wedge \bigcirc q')) \vee (w' \wedge \bigcirc(p' \,;\, q)) \\ \equiv\ & (w \wedge u \wedge \varepsilon) \vee (w \wedge u' \wedge \bigcirc q') \vee (w' \wedge \bigcirc(p' \,;\, q)) \end{aligned}$$

Otherwise w is extended state formulae with at least one atomic block, then we have $w \wedge \varepsilon \equiv \mathsf{False}$. Thus,

$$\begin{aligned} & ((w \wedge \varepsilon) \,;\, q) \vee ((w' \wedge \bigcirc p') \,;\, q) \\ \equiv\ & (w' \wedge \bigcirc p') \,;\, q \\ \equiv\ & w' \wedge \bigcirc(p' \,;\, q) \ \ (L23) \end{aligned}$$

(7) If p is the conditional statement, by hypothesis, we have $p \equiv w \wedge \varepsilon \vee w' \wedge \bigcirc p'$ and $q \equiv u \wedge \varepsilon \vee u' \wedge \bigcirc q'$, where w, w', u, u' are extended state formulas. Thus, by the definition of conditional statement, we have

$$\texttt{if}\ b\ \texttt{then}\ p\ \texttt{else}\ q \equiv (b \wedge p) \vee (\neg b \wedge q)$$

As seen, if b is true the statement is reduced to $p \equiv w \wedge \varepsilon \vee w' \wedge \bigcirc p'$ otherwise it is reduced to $q \equiv u \wedge \varepsilon \vee u' \wedge \bigcirc q'$.

(8) If p is the while statement, we assume that $p \equiv (w \wedge \bigcirc q) \vee (u \wedge \varepsilon)$, w and u are extended state formulae. We have

$$\begin{aligned} \texttt{while}\ b\ \mathsf{do}\ p &\equiv \texttt{if}\ b\ \texttt{then}\ (p \wedge \mathsf{more} \,;\, \texttt{while}\ b\ \mathsf{do}\ p)\ \texttt{else}\ \varepsilon \\ &\equiv (b \wedge (p \wedge \mathsf{more} \,;\, \texttt{while}\ b\ \mathsf{do}\ p)) \vee (\neg b \wedge \varepsilon) \\ &\equiv (b \wedge (p \wedge \bigcirc\mathsf{True} \,;\, \texttt{while}\ b\ \mathsf{do}\ p)) \vee (\neg b \wedge \varepsilon) \\ &\equiv (b \wedge (((w \wedge \bigcirc q) \vee (w' \wedge \varepsilon)) \wedge \bigcirc\mathsf{True} \,;\, \texttt{while}\ b\ \mathsf{do}\ p)) \vee (\neg b \wedge \varepsilon) \\ &\equiv (b \wedge (w \wedge \bigcirc q \wedge \bigcirc\mathsf{True} \,;\, \texttt{while}\ b\ \mathsf{do}\ p)) \vee (\neg b \wedge \varepsilon) \\ &\equiv (b \wedge (w \wedge (\bigcirc q \wedge \bigcirc\mathsf{True} \,;\, \texttt{while}\ b\ \mathsf{do}\ p))) \vee (\neg b \wedge \varepsilon) \\ &\equiv (b \wedge (w \wedge \bigcirc(q \wedge \mathsf{True} \,;\, \texttt{while}\ b\ \mathsf{do}\ p))) \vee (\neg b \wedge \varepsilon) \\ &\equiv b \wedge w \wedge \bigcirc(q \wedge \mathsf{True} \,;\, \texttt{while}\ b\ \mathsf{do}\ p) \vee (\neg b \wedge \varepsilon) \end{aligned}$$

A special case of the while statement is $p \equiv ps \wedge \varepsilon$. In this case, while statement is reduced to $ps \wedge \varepsilon$ by its definition in Fig.7.

(9) If p is the frame statement, $\mathsf{frame}(x)$, we have

$$\begin{aligned}\mathsf{frame}(x) &\equiv \mathsf{frame}(x) \wedge (\varepsilon \vee \mathsf{more}) \\ &\equiv (\mathsf{frame}(x) \wedge \varepsilon) \vee (\mathsf{frame}(x) \wedge \mathsf{more}) \\ &\equiv \varepsilon \vee \bigcirc(\mathsf{lbf}(x) \wedge \mathsf{frame}(x)) \ \ (L21, L22)\end{aligned}$$

(10) If p is the selection statement $p \vee q$, then, by hypothesis, we have,
$p \equiv w \wedge \varepsilon \vee w' \wedge \bigcirc p'$, $q \equiv u \wedge \varepsilon \vee u' \wedge \bigcirc q'$, where w, w' and u, u' are extended state formulas.

$$\begin{aligned}p \vee q &\equiv (w \wedge \varepsilon \vee w' \wedge \bigcirc p') \vee (u \wedge \varepsilon \vee u' \wedge \bigcirc q') \\ &\equiv (w \wedge \varepsilon) \vee (u \wedge \varepsilon) \vee (w' \wedge \bigcirc p') \vee (u' \wedge \bigcirc q')\end{aligned}$$

(11) If p is the existential quantification $\exists x : p(x)$:

$$\exists x : p(x) \stackrel{y}{\equiv} p(y)$$

where $p(y)$ is a renamed formula [6] of $\exists x : p(x)$, y is a free variable in $p(x)$. Suppose that $p(y)$ has its semi-normal form, by Lemma 1 and Definition 2, $\exists x : p(x)$ can be reduced to semi-normal form.

4.3 The Interleaving Semantics with Atomic Blocks

In the following we give the interleaving semantics. As Theorem 2 shown, any program without the interleaving operators can be transformed to SNFs. Hence, we can define the interleaving statement as a derived formula based on the SNFs. We first define the interleaving semantics for the single future formula ($Qs \wedge \bigcirc p$) and the single terminal formula ($Qs \wedge \varepsilon$) in Definition 4. Note that the interleaving semantics only concerns with atomic blocks, and for non-atomic blocks, the interleaving can be regarded as the conjunction of them. Therefore, the definition is discussed depending on the extended state formula Qs. For short, we use the following abbreviation:

$$\bigwedge_{i=1}^{n} \langle Q_i \rangle \stackrel{\text{def}}{=} \begin{cases} \langle Q_1 \rangle \wedge \ldots \langle Q_n \rangle & \text{if } n \geq 1, n \in N \\ \mathsf{True} & \text{if } n = 0 \end{cases}$$

Definition 4 Let $Qs_1 \equiv ps_1 \wedge \bigwedge_{i=1}^{n} \langle Q_i \rangle$ and $Qs_2 \equiv ps_2 \wedge \bigwedge_{j=1}^{m} \langle Q_j \rangle$ be extended state formulas, p and q be single future formulas ($Qs_i \wedge \bigcirc p_i$) or single terminal formulas ($Qs_i \wedge \varepsilon$) ($i = 1, 2$) and without any interleaving operators. The interleaving semantics for ($Qs_i \wedge \bigcirc p_i$) and ($Qs_i \wedge \varepsilon$) are inductively defined as follows.

- *case* 1: Let $p \equiv Qs_1 \wedge \bigcirc p_1$ and $q \equiv Qs_2 \wedge \bigcirc p_2$
  - (1a) For $n = 0$ and $m = 0$, or $n > 0$ and $m > 0$,
    - (i) If $V_{Q_i} \cap V_{Q_j} = \varnothing$ for all i and j, that is, there are no shared variables between $\bigwedge_{i=1}^{n} \langle Q_i \rangle$ and $\bigwedge_{j=1}^{m} \langle Q_j \rangle$, then $p \,|||\, q \stackrel{\text{def}}{=} (Qs_1 \wedge Qs_2) \wedge \bigcirc(p_1 \,|||\, p_2)$

(ii) If $V_{Q_i} \cap V_{Q_j} \neq \varnothing$ for some i and j, that is, there exists at least one shared variable between $\bigwedge_{i=1}^{n}\langle Q_i\rangle$ and $\bigwedge_{j=1}^{m}\langle Q_j\rangle$, then

$$\begin{aligned} p \,|||\, q \stackrel{\text{def}}{=} & (Qs_1 \wedge ps_2) \wedge \bigcirc(p_1 \,|||\, (\textstyle\bigwedge_{j=1}^{m}\langle Q_j\rangle \wedge \bigcirc p_2) \\ & \vee \ (Qs_2 \wedge ps_1) \wedge \bigcirc(p_2 \,|||\, (\textstyle\bigwedge_{i=1}^{n}\langle Q_i\rangle \wedge \bigcirc p_1) \end{aligned}$$

(1b) For $n = 0$ and $m \geq 1$ or $m = 0$ and $n \geq 1$, $p \,|||\, q \stackrel{\text{def}}{=} (Qs_1 \wedge Qs_2) \wedge \bigcirc(p_1 \,|||\, p_2)$

– *case* 2: Let $p \equiv Qs_1 \wedge \bigcirc p_1$ and $q \equiv Qs_2 \wedge \varepsilon$

(2a) For $m = 0$, $p \,|||\, q \stackrel{\text{def}}{=} (Qs_1 \wedge Qs_2) \wedge \bigcirc p_1$; (2b) For $m \geq 1$, $p \,|||\, q \stackrel{\text{def}}{=} Qs_1 \wedge \bigcirc p_1$

– *case* 3: Let $p \equiv Qs_1 \wedge \varepsilon$ and $q \equiv Qs_2 \wedge \bigcirc p_2$,

(3a) For $n = 0$, $p \,|||\, q \stackrel{\text{def}}{=} (Qs_1 \wedge Qs_2) \wedge \bigcirc p_2$; (3b) For $n \geq 1$, $p \,|||\, q \stackrel{\text{def}}{=} Qs_2 \wedge \bigcirc p_2$

– *case* 4: Let $p \equiv Qs_1 \wedge \varepsilon$ and $q \equiv Qs_2 \wedge \varepsilon$

(4a) For $n = 0$ and $m = 0$, $p \,|||\, q \stackrel{\text{def}}{=} (Qs_1 \wedge Qs_2) \wedge \varepsilon$.

(4b) For $n = 0$ and $m \geq 1$ (resp. $m = 0$ and $n \geq 1$), $p \,|||\, q \stackrel{\text{def}}{=} Qs_1 \wedge \varepsilon$ (resp. $Qs_2 \wedge \varepsilon$).

(4c) For $n \geq 1$ and $m \geq 1$, $p \,|||\, q \stackrel{\text{def}}{=} \mathsf{False}$.

There are four cases in Definition 4. Case 1 is about the interleaving between two single future formulas, that is, $(Qs_1 \wedge \bigcirc p_1) \,|||\, (Qs_2 \wedge \bigcirc p_2)$. In case (1a), we have two subcases (i): If $V_{Q_i} \cap V_{Q_j} = \varnothing$ for all $i, j \geq 1$, which means that there are no shared variables in both of the atomic interval formulas $\bigwedge_{i=1}^{n}\langle Q_i\rangle$ and $\bigwedge_{j=1}^{m}\langle Q_j\rangle$, then we can execute them in a parallel way by means of the conjunction construct ($\wedge$). (ii): If $V_{Q_i} \cap V_{Q_j} \neq \varnothing$ for some $i, j \geq 1$, which presents that there are at least one variable that is shared with the atomic interval formulas $\bigwedge_{i=1}^{n}\langle Q_i\rangle$ and $\bigwedge_{j=1}^{m}\langle Q_j\rangle$, then we can select one of them such as $\bigwedge_{i=1}^{n}\langle Q_i\rangle$ to execute at the current state, and the other atomic interval formula such as $\bigwedge_{j=1}^{m}\langle Q_j\rangle$ is dealt with at the next state. Further, in case (1b), for Qs_1 and Qs_2, if one of them contains at least an interval atomic formula and the other does not have any interval atomic formulas, then we define them as $Qs_1 \wedge Qs_2$.

Case 2 is about the interleaving between one single future formula and one single terminal formula, that is, $(Qs_1 \wedge \bigcirc p_1) \,|||\, (Qs_2 \wedge \varepsilon)$. In case (2a), if $m = 0$, then Qs_2 is state formulae and there is no interleaving between atomic blocks. Thus, we define $(Qs_1 \wedge \bigcirc p_1) \,|||\, (Qs_2 \wedge \varepsilon)$ as $(Qs_1 \wedge Qs_2 \wedge \bigcirc p_1)$. Further, in case (2b), because the atomic interval formula $\bigwedge_{i=1}^{n}\langle Q_i\rangle$ is interpreted over an interval with at least two states, we have $\bigwedge_{i=1}^{n}\langle Q_i\rangle \wedge \varepsilon \equiv \mathsf{False}$. Hence, $(Qs_1 \wedge \bigcirc p_1) \,|||\, (Qs_2 \wedge \varepsilon)$ is defined as $(Qs_1 \wedge \bigcirc p_1)$. Case 3 can be defined as the same as case 2. Finally, case 4 discusses the interleaving between two single terminal formulas, that is, $(Qs_1 \wedge \varepsilon) \,|||\, (Qs_2 \wedge \varepsilon)$, which can be understood easily.

Example 3 We show the following examples.

(1) $\langle x := 1\rangle \wedge \bigcirc p_1 \,|||\, \langle y := 1\rangle \wedge \bigcirc p_2 \stackrel{\text{def}}{=} \langle x := 1\rangle \wedge \langle y := 1\rangle \wedge \bigcirc(p_1 \,|||\, p_2)$

(2) $\langle x := 1\rangle \wedge \bigcirc p_1 \,|||\, \langle x := 2\rangle \wedge \bigcirc p_2 \stackrel{\text{def}}{=} \langle x := 1\rangle \wedge \bigcirc(p_1 \,|||\, \langle x := 2\rangle \wedge \bigcirc p_2)$
$\vee\langle x := 2\rangle \wedge \bigcirc(p_2 \,|||\, \langle x := 1\rangle \wedge \bigcirc p_1)$

(3) $x = 1 \wedge \varepsilon \,|||\, y = 2 \wedge \varepsilon \stackrel{\text{def}}{=} x = 1 \wedge y = 2 \wedge \varepsilon$

In Definition 4, we have discussed the interleaving semantics between the single future formula ($Qs \wedge \bigcirc p$) and the single terminal formula ($Qs \wedge \varepsilon$) . Further, in Definition 5, we extend Definition 4 to a general SNFs.

Definition 5 Let $p \equiv (\bigvee_{i=1}^{n_1} Qs_{ci} \wedge \bigcirc p_{fi}) \vee (\bigvee_{j=1}^{n_2} Qs_{ej} \wedge \varepsilon)$ and $q \equiv (\bigvee_{k=1}^{m_1} Qs'_{ck} \wedge \bigcirc p'_{fk}) \vee (\bigvee_{t=1}^{m_2} Qs'_{et} \wedge \varepsilon)$ $(n_1 + n_2 \geq 1, m_1 + m_2 \geq 1)$ be the general SNFs without any interleaving operators. The interleaving semantics for the general SNFs is defined as follows.

$$
\begin{aligned}
&(\bigvee_{i=1}^{n_1} Qs_{ci} \wedge \bigcirc p_{fi}) \vee (\bigvee_{j=1}^{n_2} Qs_{ej} \wedge \varepsilon) \,|||\, (\bigvee_{k=1}^{m_1} Qs'_{ck} \wedge \bigcirc p'_{fk}) \vee (\bigvee_{t=1}^{m_2} Qs'_{et} \wedge \varepsilon) \\
\stackrel{\text{def}}{=}\ & \bigvee_{i=1}^{n_1} \bigvee_{k=1}^{m_1} \left(Qs_{ci} \wedge \bigcirc p_{fi} \,|||\, Qs'_{ck} \wedge \bigcirc p'_{fk}\right) \vee \bigvee_{i=1}^{n_1} \bigvee_{t=1}^{m_2} \left(Qs_{ci} \wedge \bigcirc p_{fi} \,|||\, Qs'_{ek} \wedge \varepsilon\right) \vee \\
& \bigvee_{j=1}^{n_2} \bigvee_{k=1}^{m_1} \left(Qs_{ej} \wedge \varepsilon \,|||\, Qs'_{ck} \wedge \bigcirc p'_{fk}\right) \vee \bigvee_{j=1}^{n_2} \bigvee_{t=1}^{m_2} \left(Qs_{ej} \wedge \varepsilon \,|||\, Qs'_{et} \wedge \varepsilon\right)
\end{aligned}
$$

For instance, $(\langle x := 1\rangle \wedge \bigcirc\varepsilon \vee \langle x := 2\rangle \wedge \bigcirc\varepsilon) \,|||\, y := 2 \stackrel{\text{def}}{=} ((\langle x := 1\rangle \wedge \bigcirc\varepsilon) \,|||\, y := 2) \vee ((\langle x := 2\rangle \wedge \bigcirc\varepsilon) \,|||\, y := 2)$.

Lemma 3 Let p and q be ITL$^+$ program without any interleaving operators, then $p \,|||\, q$ can be reduced to the semi-normal forms.

Proof: From Definition 4, we can see that in (case 1), (case 2), (case 3) and (case 4), all definitions for interleaving operator are in the semi-normal forms.

Theorem 4 For any ITL$^+$ program p, there exists a semi-normal form q such that $p \equiv q$.

Proof: The proof is straightforward by Theorem 2 and Lemma 3.

Definition 6 Let p and q be ITL$^+$ programs, and the semi-normal forms of p and q be $p \equiv (\bigvee_{i=1}^{n_1} Qs_{ci} \wedge \bigcirc p_{fi}) \vee (\bigvee_{j=1}^{n_2} Qs_{ej} \wedge \varepsilon)$ and $q \equiv (\bigvee_{k=1}^{m_1} Qs'_{ck} \wedge \bigcirc p'_{fk}) \vee (\bigvee_{t=1}^{m_2} Qs'_{et} \wedge \varepsilon)$. We have the following definition.

$$
p \,|||\, q \stackrel{\text{def}}{=} (\bigvee_{i=1}^{n_1} Qs_{ci} \wedge \bigcirc p_{fi}) \vee (\bigvee_{j=1}^{n_2} Qs_{ej} \wedge \varepsilon) \,|||\, (\bigvee_{k=1}^{m_1} Qs'_{ck} \wedge \bigcirc p'_{fk}) \vee (\bigvee_{t=1}^{m_2} Qs'_{et} \wedge \varepsilon)
$$

Up to now, the entire ITL$^+$ programming language is well defined. As discussed before, since introducing atomic interval formulas, semi-normal form is necessary as a bridge for transforming programs into normal forms. In the following, we define normal form for any ITL$^+$ program and present some relevant results.

Definition 7 (Normal Form) The normal form of formula p in ITL$^+$ is defined as follows:

$$
p \equiv (\bigvee_{i=1}^{l} p_{ei} \wedge \varepsilon) \vee (\bigvee_{j=1}^{m} p_{cj} \wedge \bigcirc p_{fj})
$$

where $l + m \geq 1$ and the following hold:

– each p_{ei} and p_{cj} is a state formula.
– p_{fj} is an ITL$^+$ program.

We can see that normal form is defined based on the state formulas, whereas semi-normal form is defined based on the extended state formulas that includes the atomic interval formulas. By the definition of atomic interval formula in Fig.3, we can unfolding the atomic interval formula into ITL formulas such as $ps \wedge \bigcirc ps'$, where ps and ps' are state formulas, which are executed over two consecutive states. Thus, the extended state formulas can be reduced to the state formulas at last. In ITL$^+$ programming, normal form plays an important role that provides us a way to execute programs. To this end, we have the following theorems.

Theorem 5 For any ITL$^+$ program p, there exists a normal form q such that $p \equiv q$.

Proof: By Theorem 4, we know that there exists a semi-normal form for any ITL$^+$ program p, i.e., $p \equiv (\bigvee_{i=1}^{n_1} Qs_{ci} \wedge \bigcirc p_{fi}) \vee (\bigvee_{j=1}^{n_2} Qs_{ej} \wedge \varepsilon)$.

1. If Qs_{ci} and Qs_{ej} are state formulae, then the semi-normal form is in normal form.
2. If $Qs_{ci} \equiv ps_{1i} \wedge \bigwedge_{k=1}^{m} \langle Q_{ki} \rangle$, that is Qs_{ci} contains at least one atomic interval formula, we have
   (a) If Qs_{ej} is state formulae, then $p \equiv (\bigvee_{i=1}^{n_1} ps_{1i} \wedge \bigwedge_{k=1}^{m} \langle Q_{ki} \rangle \wedge \bigcirc p_{fi}) \vee (\bigvee_{j=1}^{n_2} Qs_{ej} \wedge \varepsilon)$. By the definition of atomic interval formula in Fig.3, we have $\bigwedge_{k=1}^{m} \langle Q_{ki} \rangle \equiv ps_i \wedge \bigcirc ps'_i$, where ps_i and ps'_i are state formulae. Hence, $p \equiv (\bigvee_{i=1}^{n_1} ps_{1i} \wedge ps_i \wedge \bigcirc ps'_i \wedge \bigcirc p_{fi}) \vee (\bigvee_{j=1}^{n_2} Qs_{ej} \wedge \varepsilon) \equiv \bigvee_{i=1}^{n_1} ps_{1i} \wedge ps_i \wedge \bigcirc (ps'_i \wedge p_{fi}) \vee (\bigvee_{j=1}^{n_2} Qs_{ej} \wedge \varepsilon)$, which is in normal form.
   (b) If Qs_{ej} contains atomic interval formula, then it is obvious that $Qs_{ej} \wedge \varepsilon \equiv \mathsf{False}$. Therefore, $p \equiv \bigvee_{i=1}^{n_1} Qs_{ci} \wedge \bigcirc p_{fi} \equiv \bigvee_{i=1}^{n_1} ps_{1i} \wedge ps_i \wedge \bigcirc (ps'_i \wedge p_{fi})$, which is in normal form.

Theorem 6 For any satisfiable ITL$^+$ program p, there is at least one minimal model.

Proof:

1. If p has at least one finite model or p has finitely many models. The proof sketch is presented as follows: Since p is satisfiable under the canonical interpretation on propositions, and p has at least one finite model, the canonical interpretation sequences on propositions, denoted by Σ_{prop} for p is finite. Thus, let $\sigma^0_{prop} \in \Sigma_{prop}$ be an arbitrary canonical interpretation sequence on propositions, if σ^0_{prop} is not a minimal interpretation sequence on propositions, then there exists a σ^1_{prop} such that $\sigma^0_{prop} \sqsupset \sigma^1_{prop}$. Analogously, if σ^1_{prop} is not a minimal interpretation sequence

on propositions, then there exists a σ^2_{prop} such that $\sigma^1_{prop} \sqsupset \sigma^2_{prop}$. In this way, we obtain a sequence

$$\sigma^0_{prop} \sqsupset \sigma^1_{prop} \sqsupset \sigma^2_{prop} \sqsupset \cdots$$

Since Σ_{prop} is finite, so there exists σ^m_{prop} is the minimal interpretation sequence. If p has finitely many models, a similar augment to the above can be given to produce the sequence.

2. If p has infinitely many infinite models and no finite model, we can prove it using Knasker-Tarski's fixed-point theorem [20]. A similar proof is given in [6]. We omit the proof here.

Theorem 5 tells us that any ITL$^+$ program can be transformed to its normal form. Theorem 6 asserts the existence of minimal model for ITL$^+$ programs.

5 Examples

We give some examples in this section to illustrate how ITL$^+$ can be used. Basically, executing a program p is to find an interval to satisfy the program. The execution of a program consists of a series of reductions or rewriting over a sequence of states (i.e., interval). At each state, the program is logically equivalent transformed to normal form (See Definition 7), such as $ps \wedge \bigcirc p_f$ if the interval does indeed continue; otherwise it reduced to $ps \wedge \varepsilon$. And, the correctness of reduction at each state is enforced by logic laws. In the following, we give 3 simple examples to simulate transactions with our atomic interval formulas and present the reductions in details. The first example shows that the framing mechanism can be supported in ITL$^+$, and we can also hide local variables when doing logic programming. The second example shows ITL$^+$ can be used to model fine-grained algorithms as well. Finally we use a two-thread concurrent program to demonstrate reduction with the interleaving operator.

Example 4 Suppose that a transaction is aimed to increment shared variable x and y atomically and variable t is a local variable used for the computation inside atomic blocks. The transactional code is shown as below, the atomicity is enforced by the low-level implementations of transactional memory system.

$$\textbf{\_stm\_atomic} \{\ t := x\ \textbf{;}\ x := t + 1\ \textbf{;}\ y := y + 1\ \}$$

Let the initial state be $x = 0, y = 0$. The above transactional code can be defined with ITL$^+$ as follows.

$$\mathsf{Prog}_1 \stackrel{\text{def}}{=} \mathsf{frame}(x) \wedge x = 0 \wedge y = 0 \wedge \langle \exists t : \mathsf{frame}(t) \wedge (t := x; x := t+1; y := y+1) \rangle \wedge \mathsf{skip}$$

Figure 8 presents the detailed rewriting procedure for the above transactional code.

The second example is about money transfer in the bank. Suppose that the money could be transferred from the account A to the account B or from the account B to the account A atomically for n ($n \geq 1$) times. The atomicity is enforced by STM implementations. We use two concurrent threads to implement the task as below.

$$
\begin{array}{ll}
 & \mathsf{frame}(x)\wedge x=0\wedge y=0\wedge\langle\exists t:\mathsf{frame}(t)\wedge(t:=x;x:=t+1;y:=y+1)\rangle\wedge\mathsf{skip}\\
\text{(L1)} & \equiv \mathsf{frame}(x)\wedge x=0\wedge y=0\wedge\langle\exists t:\mathsf{frame}(t)\wedge(t:=x;x:=t+1;y:=y+1)\rangle\wedge\bigcirc\varepsilon\wedge\mathsf{more}\\
\text{(L21)} & \equiv x=0\wedge y=0\wedge\bigcirc(\mathsf{frame}(x)\wedge\mathsf{lbf}(x)\wedge\varepsilon)\wedge\langle\exists t:\mathsf{frame}(t)\wedge(t:=x;x:=t+1;y:=y+1)\rangle\\
\text{(L22)} & \equiv x=0\wedge y=0\wedge\bigcirc(\mathsf{lbf}(x)\wedge\mathsf{lbf}(y)\wedge\varepsilon)\wedge\langle\exists t:\mathsf{frame}(t)\wedge(t:=x;x:=t+1;y:=y+1)\rangle\\
\text{(L25)} & \equiv \ldots\wedge\langle\mathsf{frame}(x)\wedge\mathsf{frame}(y)\wedge\exists t:\mathsf{frame}(t)\wedge(t:=x;x:=t+1;y:=y+1)\rangle\\
\text{(Lem1)} & \overset{z}{\equiv} \ldots\wedge\langle\mathsf{FRM}(\{x,y,z\})\wedge(z:=x\,;\,x:=z+1\,;\,y:=y+1)\rangle\\
 & \equiv \ldots\wedge\langle\mathsf{FRM}(\{x,y,z\}\wedge((\bigcirc z=x\wedge\mathsf{skip})\,;\,(\bigcirc x=z+1\wedge\mathsf{skip})\,;\,(\bigcirc y=y+1\wedge\mathsf{skip})))\rangle\\
\text{(L11,12)} & \equiv \ldots\wedge\langle\mathsf{FRM}(\{x,y,z\}\wedge\bigcirc z=0\wedge\bigcirc(\bigcirc x=z+1\wedge\mathsf{skip}\,;\,(\bigcirc y=y+1\wedge\mathsf{skip})))\rangle\\
\text{(L21)} & \equiv \ldots\wedge\langle\bigcirc(\mathsf{FRM}(\{x,y,z\}\wedge\mathsf{lbf}(x)\wedge\mathsf{lbf}(y)\wedge\mathsf{lbf}(z)\wedge z=0\wedge\\
 & \qquad(\bigcirc x=z+1\wedge\mathsf{skip}\,;\,(\bigcirc y=y+1\wedge\mathsf{skip})))\rangle\\
\text{(L11,12,17,19)} & \equiv \ldots\wedge\langle\bigcirc(\mathsf{FRM}(\{x,y,z\}\wedge(x=0\wedge\neg p_x)\wedge(y=0\wedge\neg p_y)\wedge(z=0\wedge p_z)\\
 & \qquad\wedge\bigcirc x=z+1\wedge\bigcirc(\bigcirc y=y+1\wedge\mathsf{skip}))\rangle\\
\text{(L21)} & \equiv \ldots\wedge\langle\bigcirc((x=0\wedge\neg p_x)\wedge(y=0\wedge\neg p_y)\wedge(z=0\wedge p_z)\\
 & \qquad\wedge\bigcirc(\mathsf{FRM}(\{x,y,z\}\wedge x=1\wedge\mathsf{lbf}(x)\wedge\mathsf{lbf}(y)\wedge\mathsf{lbf}(z)\wedge\bigcirc y=y+1\wedge\bigcirc\varepsilon)))\rangle\\
\text{(L18-22)} & \equiv \ldots\wedge\langle\bigcirc((x=0\wedge\neg p_x)\wedge(y=0\wedge\neg p_y)\wedge(z=0\wedge p_z)\wedge\bigcirc((x=1\wedge p_x)\\
 & \qquad\wedge(y=0\wedge\neg p_y)\wedge(z=0\wedge\neg p_z)\wedge\bigcirc\mathsf{lbf}(x)\wedge\mathsf{lbf}(y)\wedge\mathsf{lbf}(z)\wedge y=1\wedge\varepsilon)))\rangle\\
\text{(L18-22)} & \equiv \ldots\wedge\langle\bigcirc((x=0\wedge\neg p_x)\wedge(y=0\wedge\neg p_y)\wedge(z=0\wedge p_z)\wedge\bigcirc((x=1\wedge p_x)\\
 & \qquad\wedge(y=0\wedge\neg p_y)\wedge(z=0\wedge\neg p_z)\wedge\bigcirc(x=1\wedge\neg p_x\wedge z=0\wedge\neg p_z\wedge(y=1\wedge p_y)\wedge\varepsilon))))\rangle\\
\text{(Lem1,Fig.3)} & \equiv x=0\wedge y=0\wedge\bigcirc(\mathsf{lbf}(x)\wedge\mathsf{lbf}(y)\wedge\varepsilon)\wedge\bigcirc(x=1\wedge y=1)\\
\text{(Def.1)} & \equiv (x=0\wedge\neg p_x)\wedge(y=0\wedge\neg p_y)\wedge\bigcirc((x=1\wedge p_x)\wedge(y=1\wedge p_y)\wedge\varepsilon)
\end{array}
$$

Fig. 8. Rewriting Procedure for Prog_1.

Example 5 Let T_1 denote the money transfer from the account A to the account B and T_2 denote the money transfer from the account B to the account A. This example can be simulated in ITL$^+$ as follows.

In the above example, salary_1, salary_2, m_1, m_2 and n are constants, where salary_1 and salary_2 are money need to transfer; m_1 and m_2 are money in the account, and n denotes the transfer times. The transaction for money transfer in each thread is encoded in the atomic interval formula. To implement Prog_2, by Theorem 2, we first reduce threads T_1 and T_2 into their semi-normal forms, which are presented as follows.

$$
\begin{array}{l}
T_1 \equiv (i=0)\wedge(\mathsf{tmp}_1=\mathsf{salary}_1)\wedge\bigcirc p_1\wedge\langle(\mathsf{accA}:=\mathsf{accA}-\mathsf{tmp}_1);(\mathsf{accB}:=\mathsf{accB}+\mathsf{tmp}_1)\rangle\\
T_2 \equiv (j=0)\wedge(\mathsf{tmp}_2=\mathsf{salary}_2)\wedge\bigcirc p_2\wedge\langle(\mathsf{accB}:=\mathsf{accB}-\mathsf{tmp}_2);(\mathsf{accA}:=\mathsf{accA}+\mathsf{tmp}_2)\rangle
\end{array}
$$

where p_1 and p_2 are programs that will be executed at the next state and we omit their reduction details here. Further, by (case1a(ii)) in Definition 4, we have Now $T_1 \,|||\, T_2$ is reduced to the semi-normal form like $Q_s \wedge \bigcirc p$. Hence, based on the semi-normal form, we can further reduce Prog_2 to its normal form by interpreting the atomic interval formulas. We have presented the detailed reductions for atomic interval formulas in

$$\mathsf{Prog}_2 \stackrel{\text{def}}{=} \mathsf{frame}(i, \mathsf{tmp}_1, \mathsf{tmp}_2, \mathsf{accA}, \mathsf{accB}) \wedge (\mathsf{accA} = m_1) \wedge (\mathsf{accB} = m_2) \wedge (T_1 \;|||\; T_2)$$

$$\begin{array}{l} T_1 \stackrel{\text{def}}{=} (i = 0) \wedge (\mathsf{tmp}_1 = \mathsf{salary}_1) \wedge \\ \quad \texttt{while}\ (i \leq n \wedge \mathsf{tmp}_1 \leq m_1)\ \mathsf{do} \\ \quad \{ \\ \quad\ i := i + 1\ ; \\ \quad\ \langle (\mathsf{accA} := \mathsf{accA} - \mathsf{tmp}_1)\ ; \\ \quad\ (\mathsf{accB} := \mathsf{accB} + \mathsf{tmp}_1) \rangle \\ \quad \} \end{array} \qquad \begin{array}{l} T_2 \stackrel{\text{def}}{=} (j = 0) \wedge (\mathsf{tmp}_2 = \mathsf{salary}_2) \wedge \\ \quad \texttt{while}\ (i \leq n \wedge \mathsf{tmp}_2 \leq m_2)\ \mathsf{do} \\ \quad \{ \\ \quad\ j := j + 1\ ; \\ \quad\ \langle (\mathsf{accB} := \mathsf{accB} - \mathsf{tmp}_2); \\ \quad\ (\mathsf{accA} := \mathsf{accA} + \mathsf{tmp}_2) \rangle \\ \quad \} \end{array}$$

Fig. 9. Money Transfer using Transactions.

Example 4 and the whole reductions of Prog_2 can be found in technique report [24].

$$\begin{array}{rl} T_1 \;|||\; T_2 \stackrel{\text{def}}{=} & (i = 0) \wedge (\mathsf{tmp}_1 = \mathsf{salary}_1) \wedge (j = 0) \wedge (\mathsf{tmp}_2 = \mathsf{salary}_2) \wedge \\ & \quad \langle (\mathsf{accA} := \mathsf{accA} - \mathsf{tmp}_1); (\mathsf{accB} := \mathsf{accB} + \mathsf{tmp}_1) \rangle \wedge \\ & \quad\quad \bigcirc (p_1 \;|||\; (\bigcirc p_2 \wedge \langle (\mathsf{accB} := \mathsf{accB} - \mathsf{tmp}_2); (\mathsf{accA} := \mathsf{accA} + \mathsf{tmp}_2) \rangle)) \\ \vee & (i = 0) \wedge (\mathsf{tmp}_1 = \mathsf{salary}_1) \wedge (j = 0) \wedge (\mathsf{tmp}_2 = \mathsf{salary}_2) \wedge \\ & \quad \langle (\mathsf{accB} := \mathsf{accB} - \mathsf{tmp}_2)\ ; (\mathsf{accA} := \mathsf{accA} + \mathsf{tmp}_2) \rangle \wedge \\ & \quad\quad \bigcirc (p_2 \;|||\; (p_1 \wedge \langle (\mathsf{accA} := \mathsf{accA} - \mathsf{tmp}_1); (\mathsf{accB} := \mathsf{accB} + \mathsf{tmp}_1) \rangle)) \end{array}$$

In addition to simulating transactional codes, ITL$^+$ is able to model fine-grained concurrent algorithms as well. Fine-grained concurrency algorithms are usually implemented with basic machine primitives, whose atomicity is enforced by the hardware. The behaviors of these machine primitives can be modeled with *atomic interval formulas* in ITL$^+$ like this: "$\langle \bigvee_{i=1}^{n} (ps_i \wedge \bigcirc ps'_i \wedge \mathsf{skip}) \rangle$". For instance, the machine primitive $\mathsf{CAS}(\mathsf{x}, \mathsf{old}, \mathsf{new}; \mathsf{ret})$ can be defined in ITL$^+$ as follows, which has the meaning : if the value of the shared variable x is equivalent to old then to update it with new and return 1, otherwise keep x unchanged and return 0.

$$\begin{array}{rl} \mathsf{CAS}(x, \mathsf{old}, \mathsf{new}; \mathsf{ret}) \stackrel{\text{def}}{=} & \langle\ \texttt{if}\ (x = \mathsf{old})\ \texttt{then}\ x := \mathsf{new} \wedge \mathsf{ret} := 1)\ \texttt{else}\ \mathsf{ret} := 0\ \rangle \\ \equiv & \langle (x = \mathsf{old}\ \wedge x := \mathsf{new} \wedge \mathsf{ret}\ := 1) \vee (\neg (x = \mathsf{old}\ ) \wedge \mathsf{ret} := 0) \rangle \\ \equiv & \langle (x = \mathsf{old}\ \wedge \bigcirc (x = \mathsf{new} \wedge \mathsf{ret}\ = 1 \wedge \varepsilon)) \vee (\neg (x = \mathsf{old}\ ) \wedge \bigcirc (\mathsf{ret}\ = 0 \wedge \varepsilon)) \rangle \end{array}$$

The framing mechanism can help us to inherit the old values inside and outside the atomic blocks. Then we can simulate fine-grained concurrent programs using CAS commands in ITL$^+$.

6 Related Work and Conclusions

Several researchers have proposed extending logic programming with temporal logic: Tempura [15] and MSVL [7, 6] based on interval temporal logic, Cactus [18] based on branching-time temporal logic, XYZ/E [19], Templog [1] based on linear-time temporal logic. While most of these temporal languages adopt a Prolog-like syntax and programming style due to their origination in logic programming, interval temporal logic

programming languages such like Tempura can support imperative programming style. Not only does interval temporal logic allow itself to be executed [15, 7] (imperative constructs like sequential composition and while-loop can be derived straightforwardly in ITL), but more importantly, the imperative execution of ITL programs are well founded on the solid theory of framing and minimal model semantics. Indeed, ITL languages have narrowed the gap between the logic and imperative programming languages, and facilitate specifying, verifying and developing reactive systems in a more efficient and uniform way.

The fact that none of existing temporal logic programming language supports concurrent logic programming with atomic blocks, motivates our work on ITL$^+$ — the new logic allows us to model programs with the fundamental mechanism in modern concurrent programming. We have choosed to base our work on interval temproral logic and introduce a new form of formulas — atomic interval formulas, which is powerful enough to specify synchronization primitives with arbitrary granularity, as well as their interval-based semantics. The choice of ITL also enables us to make use of the *framing* technique and develop an executive language on top of the logic consequently. Indeed, we show that framing works smoothly with our interval semantics for atomicity.

In future work, we shall focus on the practical use of ITL$^+$ and plan to extend the theory and the existing tool MSVL [12, 6] to support simulation and verification of algorithms with more complex data structure such as the linked list queue [14] and the array-based queue [3].

References

[1] M. Abadi and Z. Manna. Temporal logic programming. *Journal of Symbolic Computation*, 8:277–295, 1989.

[2] N. Bidoit. Negation in rule-based data base languages: a survey. *Theoretical Computer Science*, 78(1):3–83, 1991.

[3] R. Colvin and L. Groves. Formal verification of an array-based nonblocking queue. In *Proc. 10th International Conference on Engineering of Complex Computer Systems*, pages 507–516, 2005.

[4] Z. Duan and C. Holt. Negation by default for framing in temporal logic programming. Technical Report 441, Computing Science Department, University of Newcastle upon Tyne, 1993.

[5] Z. Duan and M. Koutny. A framed temporal logic programming language. *Journal of Computer Science and Technology*, 19:341–351, 2004.

[6] Z. Duan, X. Yang, and M. Koutny. Framed temporal logic programming. *Science of Computer Programming*, 70(1):31–61, 2008.

[7] Z. Duan, X. Yang, and M.Koutny. Semantics of framed temporal logic programs. In *Proc. ICLP 2005*, pages 356–370, 2005.

[8] R. Hale. *Programming in temporal logic*. PhD thesis, Trinity College Computer Laboratory, Cambridge University, Cambridge, England, 1989.

[9] T. L. Harris and K. Fraser. Language support for lightweight transactions. In *Proc. OOPSLA 2003*, pages 388–402, 2003.

[10] E. C. R. Hehner. A practical theory of programming. *Science of Computer Programming*, 14:133–158, 1990.

[11] L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

[12] Y. Ma, Z. Duan, X. Wang, and X. Yang. An interpreter for framed tempura and its application. In *Proc. 1st Joint IEEE/IFIP Symp. on Theoretical Aspects of Soft. Eng. (TASE 2007)*, pages 251–260, 2007.

[13] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems - specification*. Springer, 1992.

[14] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distrib. Comput.*, 51(1):1–26, 1998.

[15] B. C. Moszkowski. *Executing temporal logic programs*. Cambridge University Press, 1986.

[16] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and implementation of transactional constructs for C/C++. In *Proceedings OOPSLA'08*, pages 195–212, sep 2008.

[17] A. Pnueli. The temporal semantics of concurrent programs. In *Proc. Int'l Symp. Semantics of Concurrent Computation*, volume 70, pages 1–20, 1979.

[18] P. Rondogiannis, M. Gergatsoulis, and T. Panayiotopoulos. Branching-time logic programming: The language cactus and its applications. *Computer language*, 24(3):155–178, 1998.

[19] C. Tang. A temporal logic language oriented toward software engineering – introduction to XYZ system (I). *Chinese Journal of Advanced Software Research*, 1:1–27, 1994.

[20] A. Tarski. A lattice theoretical fixpoint theroem and its application. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[21] V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In *Proc. CONCUR'07*, volume 4703, pages 256–271, 2007.

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

[23] X. Yang and Z. Duan. Operational semantics of framed tempura. *Journal of Logic and Algebraic Programming*, 78(1):22–51, 2008.

[24] X. Yang, Y. Zhang, M. Fu, and X. Feng. ITL+: A temporal programming model with atomic blocks (extended version). Technical report, Institute of Software, Chinese Academy of Sciences, 2011. `http://home.ustc.edu.cn/~fuming/itlplustr.pdf`.

[25] F. Zyulkyarov, T. Harris, O. S. Unsal, A. Cristal, and M. Valero. Debugging programs that use atomic blocks and transactional memory. In *Proc. PPoPP '10*, pages 57–66, 2010.