Front. Comput. Sci., 2014, 8(6): 958–976
DOI 10.1007/s11704-014-3342-0

RESEARCH ARTICLE

A temporal programming model with atomic blocks based on projection temporal logic

Xiaoxiao YANG (✉)[1], Yu ZHANG[1], Ming FU[2], Xinyu FENG[2]

1 State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China
2 Suzhou Institute for Advanced Study University of Science & Technology of China, SuZhou 215123, China

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 (αPTL) which extends the projection temporal logic (PTL) to support concurrent 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 in projection temporal logic also works in the new model, which consequently supports our development of an executive language. The language supports concurrency by introducing a loose interleaving semantics which tracks only the mutual exclusion between atomic blocks. We demonstrate the usage of αPTL by modeling and verifying both the fine-grained and coarse-grained concurrency.

Keywords atomic blocks, semantics, temporal logic programming, verification, framing

Received September 9, 2013; accepted June 24, 2014

E-mail: xxyang@ios.ac.cn

1 Introduction

Parallelism has become a challenging domain for processor architectures, programming, and formal methods [1]. Based on the multiprocessor systems, threads communicate via shared memory, which employs some kind of synchronization mechanisms to coordinate access to a shared memory location. Verification of such synchronization mechanisms is the key to assure the correctness of shared-memory programs. 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. 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 and realize the synchronization mechanisms in a multi-core era, and have been used in both theoretical study of fine-grained concurrency verification [2] and in modern programming languages [3–5] to support transactional programming.

On the other side, temporal logic has proved very useful in specifying and verifying concurrent programs [6] and has seen particular success in the temporal logic programming model, where both algorithms and their properties can be specified in the same language [7]. 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)

[8], Cactus [9], Tempura [10], MSVL [11], etc. All these make a good foundation for applying temporal logic to implement and verify concurrent algorithms.

However, to our best knowledge, none of these languages supports 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 the 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.

For instance, modern concurrent programs running on multi-core machines often involve machine-based memory primitives such as CAS. However, to describe such programs in traditional temporal logics, one has to explore the implementation details of CAS, which is roughly presented as the following temporal logic formula ("$\bigcirc$" is the next temporal operator):

$$\text{lock}(x) \wedge \bigcirc(\text{ if } (x = \text{old}), \text{ then } \bigcirc (x = \text{new} \wedge ret = 1)$$
$$\text{else } \bigcirc (ret = 0) \wedge \bigcirc\bigcirc \text{unlock}(x)).$$

To avoid writing such details in programs, a naive solution is to introduce all the low-level memory operations as language primitives, but this is clearly not a systematic way, not to mention that different architecture has different set of memory primitives. With atomic blocks, one can simply define these primitives by wrapping the implementation into atomic blocks:

$$CAS \overset{\text{def}}{=} \langle \text{if } (x = old), \text{ then } \bigcirc (x = new \wedge ret = 1)$$
$$\text{else } \bigcirc (ret = 0)\rangle.$$

It is sufficient that the language should support the consistent abstraction of atomic blocks and related concurrency.

Similar examples can be found in software memory transactions. For instance, the following program illustrates a common transaction of reading a value from a variable x, doing computation locally (via t) and writing the result back to x:

$$\textbf{\_stm\_atomic} \{ \, t := x \;\textbf{;}\; compute\_with(t) \;\textbf{;}\; x := t \,\}.$$

When reasoning about programs with such transactions, we would like to have a mechanism to express the atomicity of transaction execution.

Therefore, we are motivated to define the notation of *atomicity* in the framework of temporal logic and propose a new temporal logic programming language αPTL. Our work is based on the projection temporal logic (PTL) [11,12], which is a variant of interval temporal logic (ITL) [10]. We further extend interval temporal logic programming languages with the mechanism of executing code blocks *atomically*, together with a novel parallel operator that tracks only the interleaving of atomic blocks. αPTL could facilitate specifying, verifying and developing reactive systems in a more efficient and uniform way. It not only allows us to express various low-level memory primitives like CAS, but also makes it possible to model transactions of arbitrary granularity. The framing operators and the minimal model semantics inherited from interval temporal logic programming may enable us to narrow the gap between temporal logic and programming languages in a realistic way.

This paper extends the abstract conference paper [13] in the following three-fold: 1) introduces the pointer operators for modeling pointer programs in non-blocking concurrency, and completes all the proofs of lemmas and theorems; 2) investigates the verification method based on αPTL for non-blocking concurrency; 3) implements and verifies a bounded total concurrent queue within αPTL framework.

αPTL can support 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. However, 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 [11,12], 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 αPTL.

In addition, 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, $x = 1 \;|||\; y = 1$ will be translated into $x = 1 \wedge y = 1$, which reflects the fact that access 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 rest of the paper is organized as follows: in Section 2, we briefly review the temporal logic PTL and Duan's framing technique, then we extend the logic and define αPTL in Section 3, as well as its semantics in the interval model. Section 4 develops an executable subset of αPTL as a programming language. In Section 5, we present the verification method and give a case study to illustrate the implementation and verification within αPTL framework. Finally, we conclude in Section 6.

2 Temporal logic

In this section, we start with a brief review to PTL, which was proposed for reasoning about intervals of time for hardware and software systems [12,14–16]. Further, we introduce the framing technique in PTL, which is a key crux in writing and verifying temporal logic programs.

2.1 Projection temporal logic

PTL terms and formulas are defined by the following grammar:

$$\begin{array}{lll}\text{PTL terms:} & e, e_1, \ldots, e_m & ::= x \mid f(e_1, e_2, \ldots, e_m) \mid \bigcirc e \mid \ominus e,\\ \text{PTL formulas :} & p, q, p_1, p_m & ::= \pi \mid e_1 = e_2 \mid \\ & & \quad \text{Pred}(e_1, e_2, \ldots, e_m)\\ & & \quad \mid \neg p \mid p \wedge q \mid \bigcirc p \mid \exists x.\ p\\ & & \quad \mid (p_1, p_2, \ldots, p_m)\ \text{prj}\ q \mid p^+,\end{array}$$

where x is a static or dynamic 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 $\text{Pred}(e_1, e_2, \ldots, e_m)$ represents a predefined predicate constructed with $e_1, e_2, \ldots, e_m$; operators *next*($\bigcirc$), *projection*(prj) and *chop plus* ($^+$) are temporal operators. A static variable remains the same over an interval whereas a dynamic variable can have different values at different states.

Let $\mathcal{V}$ be the set of variables, $\mathcal{D}$ be the set of values including integers, lists, etc., and $\mathcal{P}rop$ be the set of primitive propositions. A *state* s is a pair of assignments (I_{var}, I_{prop}), where $I_{var} \in \mathcal{V} \to \mathcal{D} \cup \{\text{nil}\}$ (nil denotes undefined values) and $I_{prop} \in \mathcal{P}rop \to \{\text{True}, \text{False}\}$. We often write $s[x]$ for the value $I_{var}(x)$, and $s[\pi]$ for the boolean value $I_{prop}(\pi)$. An *interval* $\sigma = \langle s_0, s_1, \ldots\rangle$ is a sequence of states, of which the length, denoted by $|\sigma|$, is n if $\sigma = \langle s_0, s_1, \ldots, s_n\rangle$ and ω if σ is infinite. An empty interval is denoted by ϵ, $|\epsilon| = -1$. Interpretation of PTL formulas takes the following form:

$$(\sigma, i, j) \models p,$$

where σ is an interval, p is a PTL formula. We call the tuple (σ, i, j) an *interpretation*. Intuitively, $(\sigma, i, j) \models p$ means that the formula p is interpreted over the subinterval of σ starting from state s_i and ending at s_j.

Given two intervals σ, σ', we write $\sigma \overset{x}{=} \sigma'$ 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 $0 \leqslant k \leqslant |\sigma|$. The operator $(\cdot)$ means the concatenation of two intervals σ and σ', that is, for $\sigma = \langle s_0, s_1, \ldots, s_n\rangle$ and $\sigma' = \langle s_{n+1}, \ldots\rangle$, we have $\sigma \cdot \sigma' = \langle s_0, \ldots s_n, s_{n+1}, \ldots\rangle$. Interpretations of PTL terms and formulas are defined in Fig. 1.

Below is a set of syntactic abbreviations that we shall use frequently:

$$\varepsilon \overset{\text{def}}{=} \neg\bigcirc\text{True}, \quad \text{more} \overset{\text{def}}{=} \neg\varepsilon, \quad p_1\ ;\ p_2 \overset{\text{def}}{=} (p_1, p_2)\ \text{prj}\ \varepsilon,$$
$$\Diamond p \overset{\text{def}}{=} \text{True}\ ;\ p, \qquad \Box p \overset{\text{def}}{=} \neg\Diamond\neg p, \qquad \text{skip} \overset{\text{def}}{=} \text{len}(1),$$
$$x := e \overset{\text{def}}{=} \bigcirc x = e \wedge \text{skip}, \qquad \odot p \overset{\text{def}}{=} \bigcirc p \vee \varepsilon,$$
$$\text{len}(n) \overset{\text{def}}{=} \begin{cases}\varepsilon, & \text{if } n = 0,\\ \bigcirc\text{len}(n-1), & \text{if } n > 1.\end{cases}$$

ε specifies intervals whose current state is the final state; an interval satisfying more requires that the current state not be the final state. The semantics of $p_1\ ;\ p_2$ says that computation p_2 follows p_1, and the intervals for p_1 and p_2 share a common state. Note that *chop* (;) formula can be defined directly by the projection operator. $\Diamond p$ says that p holds eventually in the future; $\Box p$ means p holds at every state after (including) the current state; 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. $\odot p$ (*weak next*) tells us that either the current state is the final one or p holds at the next state of the present interval.

2.2 Framing issue

Framing concerns how the value of a variable can be carried from one state to the next. Within ITL community, Duan and Maciej [12] proposed a framing technique through an explicit operator (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

$$
\begin{array}{ll}
(\sigma, i, j)[x] & = s_i[x] \\
(\sigma, i, j)[f(e_1, e_2, \ldots, e_m)] & = \begin{cases} f((\sigma, i, j)[e_1], \ldots, (\sigma, i, j)[e_m]) \text{ if } (\sigma, i, j)[e_h] \neq nil \text{ for all } h \\ nil \qquad \text{otherwise} \end{cases} \\
(\sigma, i, j)[\bigcirc e] & = \begin{cases} (\sigma, i+1, j)[e_1] \text{ if } i < j \\ nil \text{ otherwise} \end{cases} \\
(\sigma, i, j)[\bigcirc e] & = \begin{cases} (\sigma, i-1, j)[e] \text{ if } i > 0 \\ nil \text{ otherwise} \end{cases} \\
(\sigma, i, j) \models \pi & \textbf{iff } s_i[\pi] = \text{True} \\
(\sigma, i, j) \models e_1 = e_2 & \textbf{iff } (\sigma, i, j)[e_1] = (\sigma, i, j)[e_2] \\
(\sigma, i, j) \models \text{Pred}(e_1, e_2, \ldots, e_m) & \textbf{iff } \text{Pred}((\sigma, i, j)[e_1], \cdots, (\sigma, i, j)[e_m]) = \text{True} \\
(\sigma, i, j) \models \neg p & \textbf{iff } (\sigma, i, j) \not\models p \\
(\sigma, i, j) \models p_1 \wedge p_1 & \textbf{iff } (\sigma, i, j) \models p_1 \text{ and } (\sigma, i, j) \models p_2 \\
(\sigma, i, j) \models \bigcirc p & \textbf{iff } i < j \text{ and } (\sigma, i+1, j) \models p \\
(\sigma, i, j) \models \exists x.p & \textbf{iff } \text{there exists } \sigma' \text{ such that } \sigma' \, x{=}\sigma \text{ and } (\sigma', i, j) \models p \\
(\sigma, i, j) \models (p_1, p_2, \ldots, p_m) \text{ prj q} & \textbf{iff } \text{if there are } i = r_0 \leqslant r_1 \leqslant \cdots \leqslant r_m \leq j \text{ such that} \\
 & (\sigma, r_0, r_1) \models p_1 \text{ and } (\sigma, r_{l-1}, r_l) \models p_l \text{ for all } 1 < l \leqslant m \text{ and} \\
 & (\sigma', 0, |\sigma'|) \models q \text{ for } \sigma' \text{ given by :} \\
 & \text{1) } r_m < j \text{ and } \sigma' = \sigma \downarrow (r_0, r_1, \ldots, r_m) \cdot \sigma(r_m{+}1..j) \\
 & \text{2) } r_m = j \text{ and } \sigma' = \sigma \downarrow (r_0, r_1, \ldots, r_h) \text{ for some } 0 \leqslant h \leqslant m. \\
(\sigma, i, j) \models p^{+} & \textbf{iff } \text{if there are } i = r_0 \leqslant r_1 \leqslant \cdots \leqslant r_{n-1} \leqslant r_n = j \ (n > 1) \text{ such that} \\
 & (\sigma, r_0, r_1) \models p \text{ and } (\sigma, r_{l-1}, r_l) \models p \ (1 < l \leqslant n)
\end{array}
$$

Fig. 1 Semantics for PTL terms and formulas

[11,14]. The key characteristic of the frame operator can be stated as: frame(x) means that variable x keeps its old value over an interval if no assignment to x has been encountered.

The framing technique defines a primitive proposition aflag$_x$ for each variable x: intuitively aflag$_x$ denotes an assignment of a new value to x — whenever such an assignment occurs, aflag$_x$ must be true; however, if there is no assignment to x, aflag$_x$ is unspecified, and in this case, we will use a *minimal model* [11,12] to force it to be false. Formally, frame(x) is defined as follows:

$$\text{frame}(x) \overset{\text{def}}{=} \Box(\text{more} \rightarrow \bigcirc\, \text{lbf}(x)),$$
$$\text{where lbf}(x) \overset{\text{def}}{=} \neg\text{aflag}_x \rightarrow \exists\, b \ :\ (\ominus x = b \wedge x = b).$$

Intuitively, lbf(x) (looking back framing) 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 say a program is framed if it contains lbf(x) or frame(x).

We will interpret a framed program p using minimal (canonical) models. There are three main ways of interpreting propositions in p, namely the canonical, complete and partial, as in the semantics of logic programming languages [17]. Here we use the canonical one to interpret programs. That is, each I^k_{var} in a model $\sigma = \langle (I^0_{var}, I^0_{prop}), (I^1_{var}, I^1_{prop}), \ldots \rangle$ is used as in the underlying logic, but I^k_{prop} is changed to the canonical interpretation.

A *canonical interpretation* on propositions is a set $I \subseteq \mathcal{P}rop$, and all propositions not in I are false. Let $\sigma = \langle (I^0_{var}, I^0_{prop}), (I^1_{var}, I^1_{prop}), \ldots \rangle$ be a model. We denote the sequence of interpretation on propositions of σ by $\sigma_{prop} = \langle I^0_{prop}, I^1_{prop}, \ldots \rangle$, and call σ_{prop} canonical if each I^i_{prop} is a canonical interpretation on propositions. Moreover, if $\sigma' = \langle (I'^0_{var}, I'^0_{prop}), (I'^1_{var}, I'^1_{prop}), \ldots \rangle$ is another canonical model, then we denote:

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

For instance, $\langle (\{x{:}1\}, \varnothing) \rangle \sqsubset \langle (\varnothing, \{\text{aflag}_x\}) \rangle$.

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 [11]) Let p be a PTL formula, and (σ, i, j) be a canonical interpretation. Then the minimal satisfaction relation $\models_m$ is defined as $(\sigma, i, j) \models_m p$ iff $(\sigma, i, j) \models p$ and there is no σ' such that $\sigma' \sqsubset \sigma$ and $(\sigma', i, j) \models p$.

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

$$\sigma_1 = \langle (\{x{:}1\}, \{\text{aflag}_x\}), (\varnothing, \{\text{aflag}_x\}) \rangle,$$
$$\sigma_2 = \langle (\{x{:}1\}, \{\text{aflag}_x\}), (\{x{:}1\}, \varnothing) \rangle.$$

The canonical model does not specify aflag$_x$ at state s_1 — it can be True too, but taking True as the value of aflag$_x$ 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 with atomic blocks

3.1 The logic αPTL

In the following, we discuss the temporal logic αPTL, which augments PTL with the reference operator (&) and the dereference operator ($*$) and atomic blocks ($\langle\rangle$). The introduction of pointer operations ($\&, *$) into PTL facilitates describing a multitude of concurrent data structures that involve pointer operations. αPTL terms and formulas can be defined by the following grammar.

$$\begin{array}{lll}\alpha\text{PTL terms:} & e, e_1, \ldots, e_m & ::= x \mid \&x \mid *Z \mid f(e_1, e_2, \ldots, e_m) \\ & & \mid \bigcirc e \mid \ominus e, \text{ where } Z ::= x \mid \&x, \\ \alpha\text{PTL formulas:} & p, q, p_1, p_m & ::= \pi \mid e_1 = e_2 \mid \text{Pred}(e_1, e_2, \ldots, e_m) \\ & & \mid \neg p \mid p \wedge q \mid Q \mid \langle p \rangle \mid \bigcirc p \\ & & \mid (p_1, p_2, \ldots, p_m) prj\ q \mid \exists x.\ p \mid p^+,\end{array}$$

where x is an integer variable, $\&x$ is a reference operation and $*Z$ is a dereference operation. The novel construct $\langle\cdot\rangle$ is used to specify atomic blocks with arbitrary granularity in concurrency and we call $\langle p \rangle$ an *atomic interval formula*. All other constructs are as in PTL except that they can take atomic interval formulas. The interpretation of αPTL terms can be found in Fig. 2, where we only show the interpretation of the pointer operations. Let $\Im$ denote the interpretation (σ, i, j). The interpretation of other terms are the same as in PTL.

$$\Im[\&x] = x.$$
$$\Im[*Z] = \begin{cases} \Im[\Im[Z]], & \text{where } \Im[Z] \in \mathcal{V}; \\ nil, & \text{otherwise}. \end{cases}$$

Fig. 2 Interpretation of pointer operations

From Fig. 2, we can see that if x is a pointer pointing to y, then $\Im[x] = y$ ($y \in \mathcal{V}$) and $\Im[*x] = \Im[y]$. Otherwise, if x is not a pointer, then $\Im[x] \in D$, but $\Im[x] \notin \mathcal{V}$, from the interpretation, we know that $\Im[*x] = nil$, which is unspecified.

Example 1 In the following, we give an example to illustrate how the pointer operations can be interpreted in our underlying logic.

$$\begin{aligned}\text{Prog} &= (x = 5) \wedge (y := 7) \wedge (p := \&y) \wedge (t := *p) \\ &\Longleftrightarrow (x = 5) \wedge \bigcirc(y = 7 \wedge p = \&y \wedge t = *p \wedge \varepsilon) \\ &\Longleftrightarrow (x = 5) \wedge \bigcirc((y = 7) \wedge (p = \Im[\&y]) \wedge (t = \Im[*p]) \wedge \varepsilon) \\ &\Longleftrightarrow (x = 5) \wedge \bigcirc((y = 7) \wedge (p = y) \wedge (t = \Im[*p]) \wedge \varepsilon) \\ &\Longleftrightarrow (x = 5) \wedge \bigcirc((y = 7) \wedge (p = y) \wedge (t = \Im[\Im[p]]) \wedge \varepsilon) \\ &\Longleftrightarrow (x = 5) \wedge \bigcirc((y = 7) \wedge (p = y) \wedge (t = \Im[y]) \wedge \varepsilon) \\ &\Longleftrightarrow (x = 5) \wedge \bigcirc((y = 7) \wedge (p = y) \wedge (t = 7) \wedge \varepsilon).\end{aligned}$$

Thus, the model of Prog is $\sigma = \langle(\{x : 5\}, \emptyset), (\{y : 7, p : y, t : 7\}, \emptyset)\rangle$.

The essence of atomic execution is 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 $\langle p \rangle$ always specifies a single transition between two consecutive states. The formula p will be interpreted at another interval (the inner level), which we call an *atomic interval*. It is assumed that the atomic intervals must be finite and only the first and final states of the atomic intervals can be 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 a formula p, let V_p be the set of free variables of p. We define formula FRM(V_p) as follows:

$$\text{FRM}(V_p) \overset{\text{def}}{=} \begin{cases} \bigwedge_{x \in V_p} \text{frame}(x), & \text{if } V_p \neq \emptyset; \\ \text{True}, & \text{otherwise}. \end{cases}$$

  FRM(V_p) says that each variable in the set V_p is a framed variable that allows to inherit the old value from previous states. FRM(V_p) 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

$$\sigma \cdot \sigma' = \begin{cases} \sigma, & \text{if } |\sigma| = \omega \text{ or } \sigma' = \epsilon; \\ \sigma', & \text{if } \sigma = \epsilon; \\ \langle s_0, ..., s_i, s_{i+1}, ...\rangle, & \text{if } \sigma = \langle s_0, s_1, ..., s_i\rangle \text{ and} \\ & \quad \sigma' = \langle s_{i+1}, ...\rangle. \end{cases}$$

- 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.

Definition 2 (interpretation of atomic interval formulas) Let (σ, i, j) be an interpretation and p be a formula. Atomic interval formula $\langle p\rangle$ is defined as:

$$(\sigma, i, j) \models \langle p\rangle \text{ iff there exists a finite interval } \sigma' \text{ and } I'_{prop} \text{ such that } \langle s_i\rangle \cdot \sigma' \cdot \langle s_{i+1}|_{I'_{prop}}\rangle \models p \wedge \mathrm{FRM}(V_p).$$

The interpretation of atomic interval formulas is the key novelty of the paper, which represents exactly the semantics of atomic executions: the entire formula $\langle p\rangle$ specifies a single state transition (from s_i to s_{i+1}), and the real execution of the block, which represented by p, can be carried over a finite interval (of arbitrary length), with the first and final state connected to s_i and s_{i+1} respectively — the internal states of p (states of σ') are hidden from outside the atomic block. Notice that when interpreting p, we use $\mathrm{FRM}(V_p)$ to carry values through the interval, however this may lead to conflict interpretations to some primitive propositions (basically primitive propositions like p_x for variables that take effect both outside and inside atomic blocks), hence the final state is not exactly s_{i+1} but $s_{i+1}|_{I'_{prop}}$ with a different interpretation of primitive propositions. In Section 3.2, we shall explain the working details of framing in atomic blocks by examples.

In the following, we present some useful αPTL formulas that are frequently used in the rest of the paper. Note that other derived formulas such as ε, more, len(n), skip, $\Box p$ and $\Diamond p$ are the same as defined in PTL.

- $p^* \overset{\text{def}}{=} p^+ \vee \varepsilon$ (*chop star*): either chop plus p^+ or ε;
- $p \equiv q \overset{\text{def}}{=} \Box(p \leftrightarrow q)$ (*strong equivalence*): p and q have the same truth value in all states of every model;
- $p \supset q \overset{\text{def}}{=} \Box(p \rightarrow q)$ (*strong implication*): $p \rightarrow q$ always holds in all states of every model.

In the temporal programming model, we support two minimum execution unit defined as follows. They are used to construct *normal forms* [11] of program executions.

State formulas are defined as follows:.

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

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

Since atomic interval formulas are introduced in αPTL, we extend the notion of state formulas to include atomic interval formulas:

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

where p is arbitrary PTL formulas and ps is state formulas.

Theorem 1 The logic laws in Fig. 3 are valid.

In Fig. 3, we present a collection of logic laws that are useful for reasoning about αPTL formulas. Their proofs can be found in the Appendix A.

3.2 Support framing in atomic interval formulas

Framing is a powerful technique that carries values through over interval states in temporal logic programming and it is also used inside atomic blocks in αPTL. While in PTL, framing is usually explicitly specified by users, we have made it inherent in the semantics of atomic interval formulas in αPTL, which is the role of $\mathrm{FRM}(V_Q)$. However, framing inside atomic blocks must be carefully manipulated, as we use same primitive propositions (such as aflag_x) to track the modification of variables outside and inside atomic blocks —

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

Fig. 3 Some αPTL Laws: where p, q, p_1 and p_2 are αPTL formulas; Q, Q_1 and Q_2 are PTL formulas

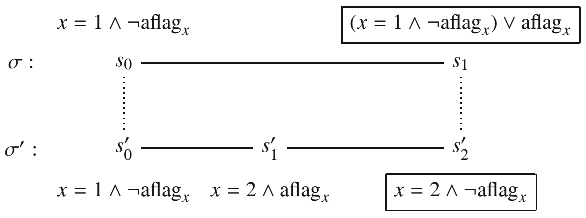
conflict interpretations of primitive propositions may occur at the exit of atomic blocks. We demonstrate this by the following example:

$$(\sigma, 0, |\sigma|) \models \mathrm{FRM}(\{x\}) \wedge x = 1 \wedge \langle Q \rangle,$$

$$\text{where } Q \overset{\text{def}}{=} \bigcirc x = 2 \wedge \mathrm{len}(2).$$

Q requires the length of the atomic interval (that is σ' in the following diagram) to be 2. Inside the atomic block, which is interpreted at the atomic interval, FRM($\{x\}$) will record that x is updated with 2 at the second state (s'_1) and remains unchanged till the last state (s'_2).

When exiting the atomic block (at state s'_2 of σ'), we need to merge the updating information with that obtained from FRM($\{x\}$) outside the block, which intend to inherit the previous value 1 if there is no assignment to x. This conflicts the value sent out from the atomic block, which says that the value of x is 2 with a negative proposition $\neg\mathrm{aflag}_x$. The conflict is well illustrated by the following diagram:


$x = 1 \wedge \neg\mathrm{aflag}_x$
$\boxed{(x = 1 \wedge \neg\mathrm{aflag}_x) \vee \mathrm{aflag}_x}$
σ :
s_0
s_1
σ' :
s'_0
s'_1
s'_2
$x = 1 \wedge \neg\mathrm{aflag}_x$
$x = 2 \wedge \mathrm{aflag}_x$
$\boxed{x = 2 \wedge \neg\mathrm{aflag}_x}$

A naive merge of the last states (s_1 and s'_2) in both intervals will produce a false formula: $((x = 1 \wedge \neg\mathrm{aflag}_x) \vee \mathrm{aflag}_x) \wedge (x = 2 \wedge \neg\mathrm{aflag}_x)$. We solve this problem by adopting different interpretation of aflag_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) we have $(x = 1 \wedge \neg\mathrm{aflag}_x \vee \mathrm{aflag}_x)$ while inside the block we replace s'_2 by $s_1|_{I'_{prop}}$ with a reinterpretation of p_x in I'_{prop}, which gives rise to $(x = 1 \wedge \neg\mathrm{aflag}_x \vee \mathrm{aflag}_x) \wedge (x = 2) \equiv (x = 2) \wedge \mathrm{aflag}_x$ at state s_1. This defines consistent interpretations of formulas outside and inside atomic blocks. It also conforms to the intuition: as long as the value of global variable x is modified inside atomic blocks, then the primitive proposition aflag_x associated with x must be set True when seen from outside the block, even though there is no assignment outside or at the last internal state, as in the above example. The detailed interpretations can refer to the Appendix B.

4 Temporal programming with atomic blocks

MSVL is a modeling simulation and verification language. In our previous work [11,14], we have given a complete operational semantics and minimal model semantics for MSVL such that it can be well used in verification of software systems.

In this section, to specify concurrent programs with pointer and atomic blocks, we need to extend the MSVL with the pointer operators, atomic blocks and interleaving operator. The pointer operators and atomic blocks are defined as primitive terms and formulas in αPTL, whereas interleaving operator can be derived from the basics of αPTL, which captures the interleaving between processes with atomic blocks. In the following, we present expressions and statements in our temporal programming model.

4.1 Expressions and statements

αPTL provides permissible arithmetic expressions and boolean expressions and both are basic terms of αPTL.

$$\begin{array}{ll} \text{Arithmetic expressions:} & e ::= n \mid x \mid \&x \mid *Z \mid \bigcirc x \mid \ominus x \\ & \quad\mid e_0 \ \text{op} \ e_1, \\ \text{Boolean expressions:} & b ::= \text{True} \mid \text{False} \mid \neg b \mid b_0 \wedge b_1 \\ & \quad\mid e_0 = e_1 \mid e_0 < e_1, \end{array}$$

where $Z ::= x \mid \&x$, n is an integer, x is a program variable including the integer variables and pointer variables, $\&x$ and $*Z$ are pointer operations, op represents common arithmetic operations, $\bigcirc x$ and $\ominus x$ mean that x is evaluated over the next and previous state respectively.

Figure 4 shows the statements of αPTL, where $p, q, \ldots$ are αPTL formulas. ε means termination on the current state; $x = e$ represents unification over the current state or boolean conditions; $x \Leftarrow e$, lbf(x) and frame(x) support framing mechanism and are discussed in Section 2.2; the assignment $x := e$ is as defined in Section 2.2; $p \wedge q$ 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. The conditional and while statements are defined as below:

$$\text{if } b \text{ then } p \text{ else } q \overset{\text{def}}{=} (b \wedge p) \vee (\neg b \wedge q),$$
$$\text{while } b \text{ do } p \overset{\text{def}}{=} (p \wedge b)^* \wedge \Box(\varepsilon \rightarrow \neg b).$$

If b then p else q first evaluates the boolean expression: if b is True, then the process p is executed, otherwise q is executed. The while statement 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 is false, then the while statement terminates. We use a renaming method [14]

Termination :	ε	Unification :	x=e
Positive unification :	$x \Leftarrow e$	Unit Assignment :	x:=e
State frame :	lbf(x)	Interval frame :	frame(x)
Conjuction statement :	$p \wedge q$	Selection statement:	$p \vee q$
Next statement :	$\bigcirc p$	Sequential statement :	$p\ ;p$
Conditional statement :	if b then p else q	Existential quantification :	$\exists x : p(x)$
While statement :	while b do p	Atomic block :	$\langle p \rangle$
Parallel statement :	$p \parallel\mid q$	Pointer assignment :	$*Z = e$

Fig. 4 Statements in αPTL

to reduce a program with existential quantification.

The last three statements are new in αPTL. $\langle p\rangle$ executes p atomically. $p \parallel\mid q$ executes programs p and q in parallel and we distinguish it from the standard concurrent programs by defining a novel interleaving semantics which tracks only the interleaving between atomic blocks. Intuitively, p and q must be executed at independent processors or computing units, and when neither of them contains atomic blocks, the program can immediately reduce to $p \wedge q$, which indicates that the two programs are executed in a truly concurrent manner. The formal definition of the interleaving semantics will be given in Section 4.3.

However, the new interpretation of parallel operator will force programs running at different processors to execute synchronously as if there is a global clock, which is certainly not true in reality. For instance, consider the program

$$(x := x + 1; y := y + 1) \parallel\mid (y := y + 2; x := x + 2).$$

In practice, if the two programs run on different processors, there will be data race between them, when we intend to interpret such program as a false formula and indicate a programming fault. But with the new parallel operator $\parallel\mid$, since there is no atomic blocks, the program is equivalent to

$$(\bigcirc x = x + 1 \wedge \bigcirc(\bigcirc y = y + 1)) \wedge (\bigcirc y = y + 2 \wedge \bigcirc(\bigcirc x = x + 2)),$$

which can still evaluate to true.

The latency assignment $x :=^{+} e$ is introduced to represent the non-deterministic delay of assignments:

$$x :=^{+} e \overset{\text{def}}{=} \bigvee_{n\in[1,2,\ldots,N]} \text{len}(n) \wedge \text{fin}(x = e),$$

where $\text{fin}(p) \overset{\text{def}}{=} \Box(\varepsilon \rightarrow p)$ and N is a constant denoting a latency bound. The intuition of latency assignment is that an assignment can have an arbitrary latency up to the bound before it takes effect. We explicitly require that every assignment outside atomic blocks must be a latency assignment. In order to avoid an excessive number of parentheses, the priority level of the parallel operator is the lowest in αPTL. Pointer statement $*Z = e$ means that the value of e is assigned to the memory cell pointed by Z. To cope with the existential quantification, we need Lemma 1 and Definition 3 to reduce $\exists x : p(x)$ in programs.

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 $\exists x : p(x)$.

Definition 3 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.

4.2 Semi-normal form

In the αPTL programming model, the execution of programs can be treated as a kind of formula reduction. Target programs are obtained by rewriting the original programs with logic laws (see Fig. 3), and the laws ensure that original and target programs are logically equivalent. Usually, our target programs should be represented as αPTL formulas in *normal forms*.

In this section, we shall describe how αPTL 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 4) to define the interleaving semantics, and it provides an intermediate form for transforming αPTL programs into normal forms.

Definition 4 (semi-normal form) An αPTL program is semi-normal form if it has the following form

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

where Qs_{ci} and Qs_{ej} are extended state formulas for all i, j, p_{fi} is an αPTL program, $n_1 + n_2 \geqslant 1$.

If a program terminates at the current state, then it is transformed to $\bigvee_{j=1}^{n_2} \text{Qs}_{ej} \wedge \varepsilon$; otherwise, it is transformed to $\bigvee_{i=1}^{n_1} \text{Qs}_{ci} \wedge \bigcirc p_{fi}$. For convenience, we often call $(\bigvee_{i=1}^{n_1} \text{Qs}_{ci} \wedge \bigcirc p_{fi})$ *future formulas* and $(\text{Qs}_{ci} \wedge \bigcirc p_{fi})$ *single future formu-*

las; whereas $(\bigvee_{j=1}^{n_2} \mathrm{Qs}_{ej} \wedge \varepsilon)$ *terminal formulas* and $(\mathrm{Qs}_{ej} \wedge \varepsilon)$ *single terminal formulas*.

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

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

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

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

4.3 The interleaving semantics with atomic blocks

In the following we define the parallel statement $(p \mid\mid\mid q)$ based on the SNFs. We first define the interleaving semantics for the single future formula $(\mathrm{Qs} \wedge \bigcirc p)$ and the single terminal formula $(\mathrm{Qs} \wedge \varepsilon)$ in Definition 5. 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:

$$\bigwedge_{i=1}^{l} \langle Q_i\rangle \overset{\text{def}}{=} \begin{cases} \langle Q_1\rangle \wedge \ldots \langle Q_l\rangle, & \text{if } l \geqslant 1, l \in N; \\ \text{True}, & \text{if } l = 0. \end{cases}$$

Definition 5 Let ps_1 and ps_2 be state formulas, $\mathrm{Qs}_1 \equiv ps_1 \wedge \bigwedge_{i=1}^{l_1} \langle Q_i\rangle$ and $\mathrm{Qs}_2 \equiv ps_2 \wedge \bigwedge_{i=1}^{l_2} \langle Q'_i\rangle$ be extended state formulas $(l_1, l_2 \geqslant 0)$, and T_i denote $\bigcirc p_i$ or ε $(i = 1, 2)$. The interleaving semantics for $(\mathrm{Qs}_1 \wedge T_1) \mid\mid\mid (\mathrm{Qs}_2 \wedge T_2)$ is inductively defined as follows.

- **Case 1**

$$(\mathrm{Qs}_1 \wedge \bigcirc p_1) \mid\mid\mid (\mathrm{Qs}_2 \wedge \bigcirc p_2)$$
$$\overset{\text{def}}{=} \begin{cases} \text{i)} \ (\mathrm{Qs}_1 \wedge \mathrm{Qs}_2) \wedge \bigcirc(p_1 \mid\mid\mid p_2), \ \text{if } (V_{Q_i} \cap V_{Q'_i} = \varnothing); \\ \text{ii)} \ (\mathrm{Qs}_1 \wedge ps_2) \wedge \bigcirc(p_1 \mid\mid\mid (\bigwedge_{i=1}^{l_2} \langle Q'_i\rangle \wedge \bigcirc p_2)) \vee \\ \quad (\mathrm{Qs}_2 \wedge ps_1) \wedge \bigcirc(p_2 \mid\mid\mid (\bigwedge_{i=1}^{l_1} \langle Q_i\rangle \wedge \bigcirc p_1)); \\ \quad \text{if } (V_{Q_i} \cap V_{Q'_i} \neq \varnothing). \end{cases}$$

- **Case 2**

$$(\mathrm{Qs}_1 \wedge \varepsilon) \mid\mid\mid (\mathrm{Qs}_2 \wedge T_2)$$
$$\overset{\text{def}}{=} \begin{cases} \text{i)} \ (\mathrm{Qs}_1 \wedge \mathrm{Qs}_2) \wedge T_2, \\ \quad \text{if } ((l_1 = l_2 = 0) \text{ or } (l_1 = 0 \text{ and } l_2 > 0 \text{ and } T_2 = \bigcirc p_2)); \\ \text{ii)} \ \mathrm{Qs}_1 \wedge \varepsilon, \quad \text{if } (l_2 > 0 \text{ and } T_2 = \varepsilon); \\ \text{iii)} \ \mathrm{Qs}_2 \wedge T_2, \quad \text{if } (l_1 \neq 0). \end{cases}$$

- **Case 3** $(\mathrm{Qs}_1 \wedge T_1) \mid\mid\mid (\mathrm{Qs}_2 \wedge \varepsilon)$ can be defined similarly as Case 2.

In Definition 5, there are three cases. Case 1 is about the interleaving between two single future formulas. We have two subcases. On one hand, in Case 1 i): if $V_{Q_i} \cap V_{Q'_i} = \varnothing$, which means that there are no shared variables in both of the atomic interval formulas, then we can execute them in a parallel way by means of the conjunction construct $(\wedge)$. On the other hand, in Case 1 ii): if $V_{Q_i} \cap V'_{Q_i} \neq \varnothing$, which presents that there are at least one variable that is shared with the atomic interval formulas $\bigwedge_{i=1}^{l_1} \langle Q_i\rangle$ and $\bigwedge_{i=1}^{l_2} \langle Q'_i\rangle$, then we can select one of them (such as $\bigwedge_{i=1}^{l_1} \langle Q_i\rangle$) to execute at the current state, and the other atomic interval formulas (such as $\bigwedge_{i=1}^{l_2} \langle Q'_i\rangle$) are reduced at the next state. Case 2 is about the interleaving between one single terminal formula and one single future formula or between two single terminal formulas. In Case 2 i), if Qs_1 and Qs_2 do not contain any atomic interval formulas (i.e., $l_1 = l_2 = 0$) or there are at least one atomic interval formula in Qs_2 (i.e., $l_2 > 0$) and T_2 is required to be the next statement $(\bigcirc p_2)$, then we define $(\mathrm{Qs}_1 \wedge \varepsilon) \mid\mid\mid (\mathrm{Qs}_2 \wedge T_2)$ as $(\mathrm{Qs}_1 \wedge \mathrm{Qs}_2 \wedge T_2)$. Since the atomic interval formula $\bigwedge_{i=1}^{n} \langle Q_i\rangle$ is interpreted over an interval with at least two states, we have $\bigwedge_{i=1}^{n} \langle Q_i\rangle \wedge \varepsilon \equiv \text{False}$. We discuss it in Case 2 ii) and iii) respectively. In Case 2 ii), $l_2 > 0$ and $T_2 = \varepsilon$ mean that $\mathrm{Qs}_2 \wedge T_2$ is False and we define $(\mathrm{Qs}_1 \wedge \varepsilon) \mid\mid\mid (\mathrm{Qs}_2 \wedge T_2)$ as $(\mathrm{Qs}_1 \wedge \varepsilon)$; In Case 2 iii), $(l_1 \neq 0)$ implies that $\mathrm{Qs}_1 \wedge \varepsilon$ is False and $(\mathrm{Qs}_1 \wedge \varepsilon) \mid\mid\mid (\mathrm{Qs}_2 \wedge T_2)$ is defined as $(\mathrm{Qs}_2 \wedge T_2)$. Further, Case 3 can be defined and understood similarly.

Example 3 The following examples illustrate different cases in Definition 5.

1) $\langle x := 1\rangle \wedge \bigcirc p_1 \mid\mid\mid \langle y := 1\rangle \wedge \bigcirc p_2 \overset{\text{def}}{=} \langle x := 1\rangle \wedge \langle y := 1\rangle \wedge \bigcirc(p_1 \mid\mid\mid p_2)$: because there are no shared variables in both of the atomic interval formulas, by the definition in Case 1 i), we can execute them in parallel by means of conjunction.

2) $\langle x := 1\rangle \wedge \bigcirc p_1 \mid\mid\mid \langle x := 2\rangle \wedge \bigcirc p_2 \overset{\text{def}}{=} \langle x := 1\rangle \wedge \bigcirc(p_1 \mid\mid\mid \langle x := 2\rangle \wedge \bigcirc p_2) \vee \langle x := 2\rangle \wedge \bigcirc(p_2 \mid\mid\mid \langle x := 1\rangle \wedge \bigcirc p_1)$: because variable x is shared by both atomic blocks, by Case 1 ii), one of them will be executed before the other and both cases are included in the definition by disjunction.

3) $\langle x := 1\rangle \wedge \bigcirc p \mid\mid\mid y = 2 \wedge \varepsilon \overset{\text{def}}{=} \langle x := 1\rangle \wedge y = 2 \wedge \bigcirc p$: immediately from Case 2 i).

4) $\langle y := 2\rangle \wedge \varepsilon \,|||\, \langle x := 1\rangle \wedge \bigcirc p_1 \overset{\text{def}}{=} \langle x := 1\rangle \wedge \bigcirc p_1$: this is obtained according to Case 2 iii), where $\langle y := 2\rangle \wedge \varepsilon \equiv$ False, because the atomic interval formula $\langle y := 2\rangle$ is interpreted over an interval with two states, which conflicts with ε.

Theorem 2 The interleaving semantics guarantees the mutual exclusion property of atomic blocks with shared variables.

Proof The proof is straightforward. From Definition 5 (Case 1 ii)), we can see that if two atomic interval formulas $\bigwedge_{i=1}^{l_1}\langle Q_i\rangle$ and $\bigwedge_{i=1}^{l_2}\langle Q'_i\rangle$ have shared variables, then only one atomic interval formula is allowed to execute at the current state, whereas the other atomic interval formula will be executed at the next state, which guarantees the mutual exclusion. For other cases (Case 1 i), Case 2, and Case 3), i.e., there are no shared variables between the two atomic interval formulas, we can reduce the parallel statement into the conjunction of Qs_1 and Qs_2.

In Definition 5, 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 6, we extend Definition 5 to general SNFs.

Definition 6 Let $p \equiv p_1 \,|||\, p_2$, where p_1 and p_2 are general SNFs, which are defined as follows:

$$p_1 \equiv (\bigvee_{i=1}^{n_1} \mathrm{Qs}_{ci} \wedge \bigcirc p_{fi}) \vee (\bigvee_{j=1}^{n_2} \mathrm{Qs}_{ej} \wedge \varepsilon),$$
$$p_2 \equiv (\bigvee_{k=1}^{m_1} \mathrm{Qs}'_{ck} \wedge \bigcirc p'_{fk}) \vee (\bigvee_{t=1}^{m_2} \mathrm{Qs}'_{et} \wedge \varepsilon)$$
$$(n_1 + n_2 \geqslant 1, m_1 + m_2 \geqslant 1).$$

We have the following definition.

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

From Definition 6, we can see that if programs p_1 and p_2 have semi-normal forms, then the interleaving semantics between p_1 and p_2 is the interleaving between the semi-normal forms. In the following Theorem 3, we show that there is a semi-normal form for any αPTL program p.

Theorem 3 For any αPTL program p, there exists a semi-normal form q such that $p \equiv q$.

The proof of Theorem 4 can be found in the Appendix C. Up to now, the entire αPTL 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 αPTL program and present some relevant results.

Definition 7 (normal form) The normal form of formula p in αPTL 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 $l + m \geqslant 1$ and each p_{ei} and p_{cj} is state formulas, p_{fj} is an αPTL 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 Definition 2, 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 normal form at last. In αPTL programming, normal form plays an important role that provides us a way to execute programs. To this end, we have the following theorems.

Theorem 4 For any αPTL program p, there exists a normal form q such that $p \equiv q$.

Theorem 5 For any satisfiable αPTL program p, there is at least one minimal model.

Theorem 4 tells us that any αPTL program can be reduced to its normal form. Therefore, one way to execute programs in αPTL is to transform them logically equivalently to their normal forms. Theorem 5 asserts the existence of minimal model for αPTL programs. The proofs of Theorem 4 and Theorem 5 can be found in the Appendix D and E.

5 Simulation and verification

In the previous work [13], we present the semantic framework of the temporal program model αPTL. In this section, we will discuss the verification method based on αPTL for verifying concurrent programs with any granularity. We take concurrent queues as an example to illustrate how to use αPTL to simulate a non-blocking queue and verify the linearizability property of the fine-grained concurrent queue.

5.1 Simulation

We first give some examples to illustrate how the logic αPTL 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, such as $ps \wedge \bigcirc p_f$ if the interval does indeed continue; otherwise it is reduced to $ps \wedge \varepsilon$. And, the correctness of reduction at each state is enforced by logic laws. In the following, we give two 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 αPTL, and we can also hide local variables when doing logic programming. The second example shows that αPTL can be used to model fine-grained algorithms as well.

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.

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

Let the initial state be $x = 0$ and $y = 0$. The above transactional code can be defined with αPTL as follows.

$$\begin{aligned}&\text{Prog}_1 \overset{\text{def}}{=} \text{frame}(x) \wedge x = 0 \wedge y = 0 \wedge \\ &\langle \exists t : (t := x\ ;\ x := t+1\ ;\ y := y+1) \rangle \wedge \text{skip}\end{aligned}$$

Figure 5 presents the detailed rewriting procedure for the above transactional code. For simplicity, we use the propositional primitive p_x associated with variable x to denote aflag$_x$. To interpret a formula with an existential quantification, we use the renaming method in Lemma 1 and Definition 3.

In addition to simulating transactional codes, αPTL is able to model fine-grained concurrent algorithms as well. Fine-grained concurrency algorithms are usually implemented with basic machine primitives. The behaviors of these machine primitives can be modeled with *atomic interval formulas* in αPTL like this: "$\langle \bigvee_{i=1}^{n} (ps_i \wedge \bigcirc ps'_i \wedge \text{skip}) \rangle$". For instance, the machine primitive CAS(x, old, new; ret) can be defined in αPTL 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.

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

In real systems, the atomicity of CAS is enforced by the hardware. Note that, in our logic, we use the atomic interval formula ($\langle p \rangle$) to assure the atomicity. Let us see the following example.

$$\begin{aligned}&x = 0 \wedge\\ &\text{while(True) do } \{t_1 := x\ ;\ \text{CAS}(x, t_1, t_1+1; \text{ret})\}\ |||\end{aligned}$$

$$\begin{array}{ll}
 & \text{frame}(x) \wedge x = 0 \wedge y = 0 \wedge \langle \exists t : (t := x ; x := t+1 ; y := y+1) \rangle \wedge \text{ skip}\\
\text{(L1)} & \equiv \text{frame}(x) \wedge x = 0 \wedge y = 0 \wedge \langle \exists t : (t := x ; x := t+1 ; y := y{+}1) \rangle \wedge \bigcirc\varepsilon \wedge \text{more}\\
\text{(L19)} & \equiv x{=}0 \wedge y{=}0 \wedge \bigcirc(\text{frame}(x) \wedge \text{lbf}(x) \wedge \varepsilon) \wedge \langle \exists t : (t := x ; x := t{+}1 ; y := y{+}1) \rangle\\
\text{(L20)} & \equiv x = 0 \wedge y = 0 \wedge \bigcirc(\text{lbf}(x) \wedge \varepsilon) \wedge \langle \exists t : (t := x ; x := t+1 ; y := \text{y} + 1) \rangle\\
\text{(L23)} & \equiv \cdots \wedge \langle \text{frame}(x) \wedge \text{frame}(y) \wedge \langle \exists t : (t := x ; x := t{+}1 ; y := y{+}1) \rangle\\
 & \overset{z}{\equiv} \cdots \wedge \langle \text{FRM}(\{x, y, z\}) \ \wedge (z := x ; x := z+1 ; y := y+1) \rangle\\
 & \equiv \cdots \wedge \langle \text{FRM}(\{x, y, z\} \wedge ((\bigcirc z = x \ \wedge \ \text{skip}) ; (\bigcirc x = z{+}1 \wedge \text{skip}) ; (\bigcirc y = y{+}1 \wedge \text{skip})) \rangle\\
\text{(L11,12)} & \equiv \cdots \wedge \langle \text{FRM}(\{x, y, z\} \wedge \bigcirc z = 0 \ \wedge \ \bigcirc(\bigcirc x = \text{A}z + 1 \wedge \text{skip} ; (\bigcirc y = y{+}1 \wedge \text{skip})) \rangle\\
\text{(L19)} & \equiv \cdots \wedge \langle (\text{FRM}(\{x, y, z\} \wedge \text{lbf}(x) \wedge \text{lbf}(y) \wedge \text{lbf}(z) \wedge z = 0 \wedge\\
 & (\bigcirc x = z + 1 \wedge \text{skip} ; (\bigcirc y = y{+}1 \wedge \text{skip})) \rangle\\
\text{(L11,12,15,17)} & \equiv \cdots \wedge \langle \bigcirc(\text{FRM}(\{x, y, z\} \wedge (x = 0 \ \wedge \ \neg p_x) \wedge (y = 0 \ \wedge \ \neg p_y) \wedge (z = 0 \ \wedge \ p_z)\\
 & \wedge \bigcirc x = z + 1 \wedge \bigcirc (\bigcirc y = y{+}1 \wedge \text{skip}) \rangle\\
\text{(L19)} & \equiv \cdots \wedge \langle \bigcirc((x = 0 \ \wedge \ \neg p_x) \wedge (y = 0 \ \wedge \ \neg p_y) \wedge (z = 0 \ \wedge \ p_z)\\
 & \wedge \bigcirc (\text{FRM}(\{x, y, z\} \wedge x = 1 \wedge \text{lbf}(x) \wedge \text{lbf}(y) \wedge \text{lbf}(z) \wedge \bigcirc y = y{+}1 \wedge \bigcirc\varepsilon)) \rangle\\
\text{(L16-20)} & \equiv \cdots \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)\\
 & \wedge (\text{y} = 0 \wedge \neg p_y) \wedge (\text{z} = 0 \wedge \neg p_z) \wedge \bigcirc \text{lbf}(x) \wedge \text{lbf}(y) \wedge \text{lbf}(z) \wedge y = 1 \wedge \varepsilon)) \rangle\\
\text{(L16-20)} & \equiv \cdots \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)\\
 & \wedge (\text{y} = 0 \wedge \neg p_y) \wedge (\text{z} = 0 \wedge \neg p_z) \wedge \bigcirc(\text{x} = 1 \wedge \neg p_x \wedge \text{z} = 0 \wedge \neg p_z \wedge (\text{y} = 1 \wedge p_y) \wedge \varepsilon))) \rangle\\
\text{(Minimal-model)} & \equiv x = 0 \wedge y = 0 \wedge \bigcirc(\text{lbf}(x) \wedge \varepsilon) \wedge \bigcirc(x = 1 \wedge y = 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. 5 Rewriting procedure for Prog$_1$

while(True) do $\{t_2 := x\ ;\ \text{CAS}(x, t_2, t_2 + 1; \text{ret})\}$

The above is a lock free program that increases the value of x by two threads. With the definition of atomic interval formulas and Theorem 3, we can force the two threads access to variable x exclusively.

5.2 Verification method

We employ Propositional PTL (PPTL) as the specification language to specify various temporal properties of concurrent systems, and αPTL as programming language to write concurrent programs with atomic blocks. Thus, within one notation (i.e., within PTL framework), we can verify whether an αPTL program satisfies a given property. In the following, we first introduce the basics of PPTL and its model normal form graph (NFG). Then, we introduce a verification method for the purpose of modeling simulation and verification of concurrent programs with atomic blocks.

Definition 8 PPTL formulas are defined as follows:

$$Q ::= \pi \mid \neg Q \mid Q_1 \wedge Q_2 \mid \bigcirc Q \mid (Q_1, Q_2, \ldots, Q_n)\,\text{prj}\ Q \mid Q^+,$$

where $\pi \in \mathcal{P}rop$, $Q_1, Q_2, \ldots, Q_n$ and Q are well-formed PPTL formulas. The semantics of PPTL formulas can be referred to [18].

Theorem 6 Some results for the PPTL logic [18]:

1) For any PPTL formula Q, there exists a normal form Q' such that $Q \equiv Q'$.
2) The satisfiability of PPTL formulas is decidable.

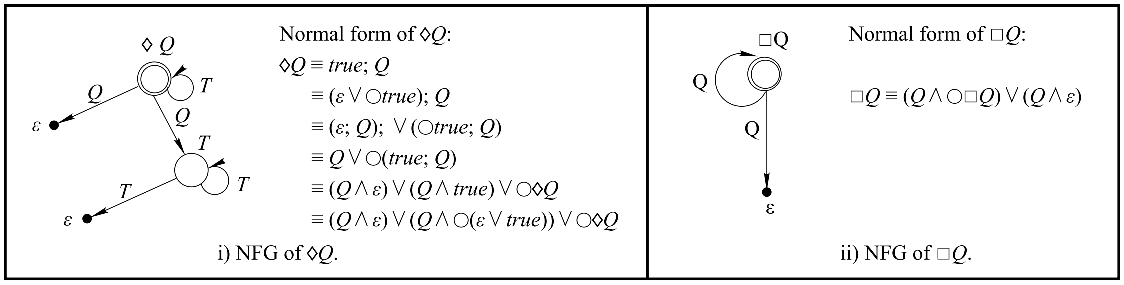
Based on the normal form of PPTL [18], we can construct a semantically equivalent graph, called *normal form graph* (NFG) [18]. For a PPTL formula Q, the NFG of Q is a directed graph, $G = (CL(Q), EL(Q))$, where $CL(Q)$ denotes the set of nodes, and $EL(Q)$ denotes the set of edges. The $CL(Q)$ and $EL(Q)$ of G are inductively defined as in Definition 9.

Definition 9 For a PPTL formula Q, the set of $CL(Q)$ of nodes and the set of $EL(Q)$ of edges connecting nodes in $CL(Q)$ are inductively defined as follows:

1) $Q \in CL(Q)$.
2) For all $Q' \in CL(Q)/\{\varepsilon, \text{False}\}$, if $Q' \equiv \bigvee_{j=1}^{h}(Q_{ej} \wedge \varepsilon) \vee \bigvee_{i=1}^{k}(Q_{ci} \wedge \bigcirc Q_i')$, then $\varepsilon \in CL(Q)$, and $(Q', Q_{ej}, \varepsilon) \in EL(Q)$ for each j, $1 \leqslant j \leqslant h$; $Q_i' \in CL(Q)$, $(Q', Q_{ci}, Q_i') \in EL(Q)$ for all i, $1 \leqslant i \leqslant k$.

The NFG of formula Q is the directed graph $G = (CL(Q), EL(Q))$. In the NFG of Q, the root node Q is denoted by a double circle; ε node by a small black dot, and each of other nodes by a single circle. Each edge is denoted by a directed arc connecting two nodes. For instance, the normal form and NFG of formulas $\Box Q$ and $\Diamond Q$ are presented in Fig. 6.

A finite path in the NFG is a sequence of nodes from the root to the ε node, while an infinite path, is a sequence of nodes from the root to some node that appear in the path for infinitely many times. An infinite (finite) interval that satisfies a PPTL formula will correspond to an infinite (finite) path in NFG. Compared with Buchi automata, NFGs have the following advantages that are more suitable for verification for interval-based temporal logics: i) NFGs are beneficial for unified verification approaches based on the same formal notation. Thus, programs and their properties can be written in the same language, which avoids the transformation between different notations. ii) NFGs can accept both finite words and infinite words. But Buchi automata can only accept infinite words. Further, temporal operators *chop* ($Q_1\ ;\ Q_2$), *chop star* (Q^*), and *projection* can be readily transformed to NFGs. iii) NFGs and PPTL formulas are semantically equivalent. That is, every path in NFGs corresponds to a model of PPTL formula. If some formula is false, then its NFG will be a false node. Thus, satisfiability in PPTL formulas can be reduced to NFGs construction. But for any LTL formula, the satisfiability problem needs to check the emptiness problem of



Normal form of $\Diamond Q$:

$$\begin{aligned}\Diamond Q &\equiv true; Q\\ &\equiv (\varepsilon \vee \bigcirc true); Q\\ &\equiv (\varepsilon; Q);\ \vee (\bigcirc true; Q)\\ &\equiv Q \vee \bigcirc(true; Q)\\ &\equiv (Q \wedge \varepsilon) \vee (Q \wedge true) \vee \bigcirc\Diamond Q\\ &\equiv (Q \wedge \varepsilon) \vee (Q \wedge \bigcirc(\varepsilon \vee true)) \vee \bigcirc\Diamond Q\end{aligned}$$

i) NFG of $\Diamond Q$.

Normal form of $\Box Q$:

$$\Box Q \equiv (Q \wedge \bigcirc\Box Q) \vee (Q \wedge \varepsilon)$$

ii) NFG of $\Box Q$.

Fig. 6 NFGs for liveness property $\Diamond Q$ and safety property $\Box Q$

Buchi automata. Recently, some promising formal verification techniques based on NFGs have been developed, such as [16,18,19].

In the following, Theorem 7 tells us that any αPTL programs, conditional on the finite domain of variables, can be expressed by PPTL formulas.

Theorem 7 Any αPTL programs can be defined in PPTL formulas, under the condition that the domain of variables is finite.

The proof can be found in the Appendix F. Based on the model checking approach, given a program p in αPTL and a property Q in PPTL, p satisfying Q can be formalized by $p \rightarrow Q$. We can check the satisfiability of $\neg(p \rightarrow Q) \equiv p \wedge \neg Q$. If a model satisfies $p \wedge \neg Q$, then there is an error trace that violates property Q; otherwise, we can conclude that Q is satisfied by p. We now give the following theorem.

Corollary 1 Let p be a program in αPTL, and Q be a PPTL formula. Satisfiability problem of $p \wedge \neg Q$ is decidable.

Proof From Theorem 6, the satisfiability of PPTL formula Q is decidable. From Theorem 7, we can see that program p can be expressed by PPTL formula Q. Therefore, the satisfiability of $p \wedge \neg Q$ is decidable.

5.3 Implementation and verification of a bounded total queue

In this section, we give a fine-grained and a coarse-grained concurrent queue written in αPTL respectively. Then we give a verification method to prove the linearizability property of the fine-grained concurrent queue.

We have threads t_1 and t_2 to access a shared queue with enq( ) and deq( ) functions. The queue provides us the *first-in-first-out* (FIFO) fairness and a limited capacity. We call a function is *total* if calls do not wait for certain conditions to become true. When a deq( ) call tries to remove an item from an empty queue, it immediately throws an exception. Also, if a total enq( ) tries to add an item to a full queue, it immediately returns an exception. Figure 7 shows us a queue, where mutex.locks ensure that at most one enqueuer, and at most one dequeuer at a time can manipulate the queue.

```
mutex.lock();        mutex.lock();
try {                try {
queue.enq(x)         queue.deq()
} finally {          } finally {
mutex.unlock;        mutex.unlock;
}                    }
```

Fig. 7 A queue with enq( ) and deq( ) functions

In the following, we will implement the queue in two ways using αPTL. One way is the coarse-grained implementation based on atomic blocks, the other way is the fine-grained implementation based on the CAS primitive. Both the atomic blocks and the CAS primitive are formalized in αPTL.

Firstly, in our underlying logic, instead of using locks, we employ atomic blocks to realize a bounded coarse-grained queue, which is presented in Fig. 8. We can see that Prog I consists of two threads t_1 and t_2 to execute $Que.enq_I$ and $Que.deq_I$ concurrently. Let Que be an array-based queue with a given capacity denoted by *size* (*size* is a constant), variables *head* and *tail* respectively refer to the first and last elements in the queue. $Que.enq_I$ and $Que.deq_I$ describe the operations enqueue and dequeue, both of which are formalized in atomic interval formulas. The $Que.enq_I$ function works as follows. If the length of *Que* is equal to the capacity *size*, then we throw a full exception, which is expressed using skip; otherwise, the enqueuer may proceed by adding a value b in the location *Que*[*tail mod size*] and the length of *Que* is increased. The $Que.deq_I$ function can be similarly understood. When $head = tail$, the queue will be empty and throw an empty exception that is expressed by skip as well. Otherwise, the length of *Que* is decreased.

```
Prog I ≝ frame(head, tail, Que) ∧ (head = tail = 0) ∧ (Que = ε)
         ∧ (while (True) do t_1 ||| while (True) do t_2)

Que.enq_I (b) ≝ ∃b : ⟨if (tail − head = size)          Que.enq_I ≝ ⟨if (tail = head)
                     then skip                                      then skip
                     else Que[tail mod size] := b ;                 else head := (head + 1)⟩
                     tail := tail + 1⟩
```

Fig. 8 A bounded coarse-grained queue implementation in αPTL

In Fig. 9, we present a fine-grained queue implementation in αPTL. Linearizability is the standard correctness condition for lock-free concurrent data structure implementations. Informally, a function is linearizable if it is observationally equivalent to an atomic execution of the function at some point. To verify linearizability property, we use auxiliary variables to capture the location of the linearisation point [20]. In the following, we first formalize the primitive CAS_lin by introducing an auxiliary variable to CAS. The new variable lin means that if variable x is updated, then lin $= 1$; otherwise, lin $= 0$. We introduce one such variable per method, which denotes the linearisation point.

$$\begin{aligned}&CAS\_\mathrm{lin}(x, old, new; ret, \mathrm{lin}) \overset{\mathrm{def}}{=}\\&\langle \text{ if } (x = old) \text{ then } (x := new \wedge ret := 1 \wedge \mathrm{lin} := 1)\\&\text{else } ret := 0 \wedge \mathrm{lin} := 0 \,\rangle\end{aligned}$$

To track the atomicity, we use auxiliary variables lineq_1, linenq_2 and $\mathrm{lindeq}_1, \mathrm{lindeq}_2$ to identify the linearization

```
PROG II ≝ frame(head, tail, Que, linenq_i, lindeq_i) ∧ (head = tail = 0) ∧
          (linenq_i = 0) ∧ (lindeq_i = 0) ∧ (Que = ε) ∧
          (while (True) do t_1 ||| while (True) do t_2) (i = 1, 2)

Que.enq_II (b) ≝ (tl_i = tail) ∧ if (tail − head = size) then skip
                 else { CAS _lin(tail, tl_i, tl_i + 1; ret_i, linenq_i) ∧ skip ;
                 if (ret_i = 1) then ∃b : ⟨Que[tl_i mod size] := b⟩ ∧ linenq_i := 0
                 else ε }
Que.deq_II = (hl_i = head) ∧ if (tail = head) then skip
             else { CAS _lin(head, hl_i, hl_i + 1; ret_i, lindeq_i) ∧ skip ; lindeq_i := 0 }
```

$$t_i \overset{\text{def}}{=} Que.enq_{II}(b) \vee Que.deq_{II} \quad \text{where } i = 1, 2$$

Fig. 9 A bounded fine-grained queue implementation in αPTL

points in functions $Que.enq_{II}$ and $Que.deq_{II}$ respectively. For convenience, we abbreviate these variables by lineq_i and lindeq_i $(i = 1, 2)$. At the beginning, linenq_i and lindeq_i are initialized to 0. Once the enqueue operation ($Que.enq_{II}$) called by some thread is successful (i.e., without full exception), we set variable linenq_i to 1 to indicate that the operation is completed at current state. After that, when the thread leaves the atomic block, we immediately reset linenq_i to 0 at the next state, which indicates that the effect of linearization point is finished. We assume that threads t_1 and t_2 implement $Que.enq_{II}$ and $Que.deq_{II}$ concurrently.

One possible implementation trace of Prog II is shown in Figure 10, where threads t_1 and t_2 enqueue or dequeue the integers $\{1, 2, 3, ...\}$ concurrently into the array. The execution sequence is like

$$t_1(enq), t_1(enq), t_2(enq), t_2(enq), t_1(deq), t_1(deq), t_2(deq), t_2(deq) \ldots$$

where $t_0(enq)$ (resp. $t_0(deq)$) means that thread t_0 calls operation $Que.enq_{II}$ (resp. $Que.deq_{II}$), and $(tail - head)$ denotes the current array capacity. Because the execution of CAS _lin is atomically, threads t_1 and t_2 access the shared variable $tail$ (or $head$) in an interleaving way. From Fig. 10, we can see that these linearization points marked by $lindeq_i$ and $linenq_i$ take effect at different time.

	Head	tail	Que	$Lindeq_1$	$Lindeq_2$	$Linenq_1$	$Linenq_2$
s_0	0	0	ε	0	0	0	0
s_1	0	1	[1]	0	0	1	0
s_2	0	2	[1, 2]	0	0	0	1
s_3	0	3	[1, 2, 3]	0	0	1	0
s_4	0	4	[1, 2, 3, 4]	0	0	0	1
s_5	1	4	[2, 3, 4]	1	0	0	0
s_6	1	5	[2, 3, 4, 5]	0	0	1	0
s_7	2	5	[3, 4, 5]	0	1	0	0
s_8	3	5	[4, 5]	1	0	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	

Fig. 10 One possible implemention of Prog II

Based on the linearisation point, we make use of PPTL to specify the linearizability property as follows, where Q_1 (or Q_2) means that the linearization points in the method $Que.enq_{II}$ (or $Que.deq_{II}$) can not take effect simultaneously.

$$Q_1 \overset{\text{def}}{=} \neg\Diamond(\text{linenq}_1 = 1 \wedge \text{linenq}_2 = 1),$$
$$Q_2 \overset{\text{def}}{=} \neg\Diamond(\text{lindeq}_1 = 1 \wedge \text{lindeq}_2 = 1).$$

Based on the verification method in Section 2, we can prove that program (Prog II) satisfies property $(Q_1 \wedge Q_2)$. As we discussed before, (Prog I) is a coarse-grained implementation based on atomic blocks, which allows us access to the queue in a sequential way. Therefore, we can regard (Prog I) as an abstraction specification, and (Prog II) as a concrete implementation. Following the verification method in Section 2, we can do the refinement checking that the set of execution traces of a concrete implementation is a subset of that of an abstraction specification. Further, the verification of linearizability property and refinement checking can be finished automatically by extending MSVL tool with atomic interval formulas and the parallel statement in the underlying logic.

6 Related work and conclusions

Several researchers have proposed extending logic programming with temporal logic: Tempura [10] and MSVL [11] based on interval temporal logic, Cactus [9] based on branching-time temporal logic, XYZ/E [21], Templog [7] based on linear-time temporal logic. Most of these temporal languages adopt a Prolog-like syntax and programming style due to their origination in logic programming, whereas interval temporal logic programming languages such like Tempura and MSVL can support imperative programming style. Not only does interval temporal logic allow itself to be executed (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. ITL languages have narrowed the gap between the logic and imperative programming languages, which are widely used in specification and verification [18,19,22,23].

The new logic αPTL allows us to model programs with the fundamental mechanism – atomic blocks in the modern concurrent programming. We have chosen to base our work on interval temporal logic and introduce a new form of formulas, i.e., 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 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. Therefore, αPTL can facilitate specifying, verifying and developing reactive systems in a more efficient and uniform way.

There are some existing works that are related to the formal definition of atomicity. For example, an atomizing bracket based on the process algebra (PA) is defined in [24], in which if these brackets are placed around a statement then this statement can be treated as an atomic action. In [25], authors propose an occurrence graph as a formal model to define the atomicity occurrence of a program. However, they do not present a well-formed verification framework to prove the correctness of programs with atomic blocks. Compared to these works, αPTL not only specifies the atomicity, but also realizes the simulation modeling and verification of concurrent programs with atomic blocks. Further, in TLA [8], an atomic operation of a concurrent program can be specified by an action, but the internal structures in the action are not explicitly interpreted. In contrast, within αPTL, we can track the internal sequence of states in the code blocks. Thus we can check the correctness of the complicated internal structures in the atomic blocks.

Our current work does not permit deadlock nor divergence within atomic blocks. In the future work, we will explore the operational semantics of αPTL to define the unsuccessful transitions like deadlock and divergence in atomic blocks. In addition, we shall focus on the practical use of αPTL and extend the theory and the existing verifier MSVL to support the verification of various temporal properties in synchronization algorithms with more complex data structure such as the linked list queue [26].

Acknowledgements We thank for anonymous referees for their suggestions and comments. This research was based on work supported by grants from Science Foundation of China Project (60833001, 61100063, 61073040 and 61103023), and by a Humboldt Fellowship (X.Y.) from Alexander von Humboldt Foundation.

References

1. Herlihy M, Shavit N. The Art of Multiprocessor Programming. Elsevier, 2009
2. Vafeiadis V, Parkinson M J A. Marriage of rely/guarantee and separation logic. In: Proceedings of the 18th International Conference on Concurrency Theory. 2007, 4703: 256–271
3. Zyulkyarov F, Harris T, Unsal O S, Cristal A, Valeroh M. Debugging programs that use atomic blocks and transactional memory. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2010, 57–66
4. Harris T, Fraser K. Language support for lightweight transactions. In: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications. 2003, 388–402
5. Ni Y, Welc A, Adl-Tabatabai A, Bach M, Berkowits S, Cownie J, Geva R, Kozhukow S, Narayanaswamy R. Design and implementation of transactional constructs for C/C++. In: Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications. 2008, 195–212
6. Pnueli A. The temporal logic of programs. In: Proceedings of the 18th IEEE Annual Symposium on Foundations of Computer Science. 1977, 46–57
7. Abadi M, Manna Z. Temporal logic programming. Journal of Symbolic Computation, 1989, 8(1–3): 277–295
8. Lamport L. The temporal logic of actions. ACM Transactions on Programming Languages Systems, 1994, 16(3): 872–923
9. Rondogiannis P, Gergatsoulis M, Panayiotopoulos T. Branching-time logic programming: the language cactus and its applications. Computer Language, 1998, 24(3): 155–178
10. Moszkowski B C. Executing Temporal Logic Programs. Cambridge University Press, 1986
11. Duan Z, Yang X, Koutny M. Semantics of framed temporal logic programs. In: Proceedings of the 21st International Conference on Logic Programming. 2005, 3668: 356–370
12. Duan Z, Koutny M. A framed temporal logic programming language. Journal of Computer Science and Technology, 2004, 19(1): 341–351
13. Yang X, Zhang Y, Fu M, Feng X. A concurrent temporal programming model with atomic blocks. In: Proceedings of the 14th International Conference on Formal Engineering Methods. 2012, 7635: 22–37
14. Yang X, Duan Z. Operational semantics of framed tempura. Journal of Logic and Algebraic Programming, 2008, 78(1): 22–51
15. Zhang N, Duan Z, Tian C. A cylinder computation model for many-core parallel computing. Theoretical Computer Science, 2013, 497: 68–83
16. Tian C, Duan Z. Complexity of propositional projection temporal logic with Star. Mathematical Structures in Computer Science, 2009, 19(1): 73–100
17. Bidoit. N. Negation in rule-based data base languages: a survey. Theoretical Computer Science, 1991, 78(1): 3–83
18. Duan Z, Tian C, Zhang L. A decision procedure for propositional projection temporal logic with infinite models. Acta Informatica, 2008, 45: 43–78
19. Duan Z, Zhang N, Koutny M. A complete proof system for propositional projection temporal logic. Theoretical Computer Science, 2013, 497: 84–107
20. Vafeiadis V. Modular Fine-Grained Concurrency Verification. Cambridge University, 2008
21. Tang C S. A temporal logic language oriented toward software engineering — introduction to XYZ system (I). Chinese Journal of Advanced Software Research, 1994, 1: 1–27
22. Schellhorn G, Tofan B, Ernst G, Reif W. Interleaved programs and rely-guarantee reasoning with ITL. In: Proceedings of 18th International Symposium on Temporal Representation and Reasoning. 2011, 99–106
23. Derrick J, Schellhorn G, Wehrheim H. Verifying linearisability with potential linearisation points. In: Proceedings of the 17th International

Symposium on Formal Methods, 2011, 6664, 327–337

24. Knijnenburg P, Kok J. The semantics of the combination of atomized statements and parallel choice. Formal Aspects of Computing, 1997, 9: 518–536
25. Best E, Randell B. A formal model of atomicity in asyn chronous systems. Acta Informatica, 1981, 16: 93–124
26. Michael M M, Scott M L. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. Journal of Parallel and Distributed Computing, 1998, 51(1): 1–26

Appendix

A) Proofs of logic laws in Fig. 3

Proof In the following, we prove some logic laws in Fig. 3 regarding atomic interval formulas and other logic laws can be proved in a similar way as in ITL.

The proof of ($L9$):

$$
\begin{aligned}
&(\sigma, i, j) \models \langle Q_1 \vee Q_2 \rangle\\
&\iff \exists\, \sigma', \sigma'', I'_{prop} \text{ s.t. } \sigma'' = \langle s_i \rangle \cdot \sigma' \cdot \langle s_{i+1}|_{I'_{prop}} \rangle \text{ and}\\
&\qquad \sigma'' \models (Q_1 \vee Q_2) \wedge \mathrm{FRM}(V_{Q_1 \vee Q_2})\\
&\iff \sigma'' \models (Q_1 \wedge \mathrm{FRM}(V_{Q_1})) \vee (Q_2 \wedge \mathrm{FRM}(V_{Q_2}))\\
&\iff \sigma'' \models Q_1 \wedge \mathrm{FRM}(V_{Q_1}) \text{ or } \sigma'' \models Q_2 \wedge \mathrm{FRM}(V_{Q_2})\\
&\iff (\sigma, i, j) \models \langle Q_1 \rangle \text{ or } (\sigma, i, j) \models \langle Q_2 \rangle\\
&\iff (\sigma, i, j) \models \langle Q_1 \rangle \vee \langle Q_2 \rangle.
\end{aligned}
$$

The proof of ($L21$):

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

The proof of ($L22$):

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

The proof of ($L23$):

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

B) Resolving framing conflicts in atomic interval formulas

The detailed interpretation of the example in Section 3.2 is presented in Fig. 11.

C) The Proof of Theorem 3

Proof The proof proceeds by induction on the structure of αPTL statements. Note that program p does not contain parallel operators.

1) If p is state formula ps including $x = e$, $x \Leftarrow e$, $\mathrm{lbf}(x)$:

$$
\begin{aligned}
ps &\equiv ps \wedge \mathrm{True}\\
&\equiv ps \wedge (\mathrm{more} \vee \varepsilon)\\
&\equiv ps \wedge \mathrm{more} \vee ps \wedge \varepsilon\\
&\equiv ps \wedge \bigcirc\mathrm{True} \vee ps \wedge \varepsilon \quad (\mathrm{more} \overset{\mathrm{def}}{=} \bigcirc\mathrm{True}).
\end{aligned}
$$

$$
\begin{array}{ll}
 & \mathrm{FRM}(\{x\}) \wedge x = 1 \wedge \langle Q \rangle \wedge \mathrm{len}(1) \quad \text{Where } Q = \bigcirc x = 2 \wedge \mathrm{len}(2)\\
 & \equiv \mathrm{frame}(x) \wedge x = 1 \wedge \langle \bigcirc x = 2 \wedge \mathrm{len}(2) \rangle \wedge \bigcirc\varepsilon\\
(L19) & \equiv \bigcirc(\mathrm{frame}(x) \wedge \mathrm{lbf}(x)) \wedge x = 1 \wedge \langle \bigcirc(\mathrm{frame}(x) \wedge \mathrm{lbf}(x)) \wedge \bigcirc x = 2 \wedge \bigcirc\varepsilon \rangle \wedge \bigcirc\varepsilon\\
(L16) & \equiv x = 1 \wedge \bigcirc(\mathrm{frame}(x) \wedge ((x = 1 \wedge \neg p_x) \vee p_x) \wedge \varepsilon) \wedge\\
 & \qquad \langle \bigcirc(\mathrm{frame}(x) \wedge ((x = 1 \wedge \neg p_x) \vee p_x)) \wedge \bigcirc x = 2 \wedge \bigcirc\bigcirc\varepsilon \rangle\\
(L17) & \equiv \cdots \wedge \langle \bigcirc(\mathrm{frame}(x) \wedge x = 2 \wedge p_x \wedge \bigcirc\varepsilon) \rangle \wedge \bigcirc\varepsilon\\
(L19) & \equiv \cdots \langle \bigcirc(\bigcirc(\mathrm{frame}(x) \wedge \mathrm{lbf}(x)) \wedge x = 2 \wedge p_x \wedge \bigcirc\varepsilon) \rangle \wedge \bigcirc\varepsilon\\
(L16) & \equiv \cdots \wedge \langle \bigcirc(\bigcirc(\mathrm{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\\
(L15) & \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\\
 & \qquad \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\\
(\text{Def. } \mathit{Minimalmodel}) & \equiv (x = 1 \wedge \neg px) \wedge \bigcirc((x = 1 \wedge \neg p_x) \vee p_x) \quad (\text{conflicts occur !})\\
 & \qquad\qquad \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\\
(\text{Def. } 2) & \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\\
 & \qquad\qquad\qquad I'_{\mathrm{prop}} \text{ used inside atomic blocks is existentially quantified}\\
 & \equiv (x = 1 \wedge \neg p_x) \wedge \bigcirc(x = 2 \wedge p_x) \wedge \bigcirc\varepsilon
\end{array}
$$

Fig. 11 Resolving framing conflicts in atomic interval formulas

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 $\langle p\rangle$:

$$\begin{aligned}
&(\sigma, i, j) \models \langle p\rangle\\
&\quad\Longleftrightarrow \exists \sigma', \sigma'', I'_{prop} \text{ s.t. } \sigma'' = \langle s_i\rangle \cdot \sigma' \cdot \langle s_{i+1}|_{I'_{prop}}\rangle, \text{ and}\\
&\sigma'' \models p \wedge \mathrm{FRM}(V_p)\\
&\quad\Longleftrightarrow (\sigma, i, j) \models ps_1 \wedge \bigcirc ps_2,\\
&\qquad\text{where } ps_1 \text{ and } ps_2 \text{ be state formulas of } p \wedge \mathrm{FRM}(V_p),\\
&\qquad\text{which hold at state } s_i \text{ and state } s_{i+1} \text{ respectively,}\\
&\qquad\text{and } ps_2 = \textstyle\bigwedge_{i=1}^{n}(x_i = e_i) \text{ or True.}
\end{aligned}$$

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

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

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

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

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

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

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

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

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.

9) If p is the frame statement, frame(x), we have

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

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 $\exists x : p(x)$:

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

where $p(y)$ is a renamed formula [11] of $\exists x : p(x)$, y is a free variable in $p(x)$. Suppose that $p(y)$ has its semi-normal form, by Lemma 2 and Definition 3 as follows, $\exists x : p(x)$ can be reduced to semi-normal form.

12) If p is the latency assignment statement $x :=^{+} e$:

$$\begin{aligned}
&x :=^{+} e\\
&\overset{\text{def}}{=} \textstyle\bigvee_{n\in[1,2,\ldots,N]} \text{len}(n) \wedge \text{fin}(x = e)\\
&\equiv \textstyle\bigvee_{n\in[1,2,\ldots,N]} \text{len}(n) \wedge \Box(\varepsilon \to x = e)\\
&\equiv \bigcirc(\textstyle\bigvee_{n\in[1,2,\ldots,N]} \text{len}(n-1)) \wedge ((\varepsilon \to x = e) \wedge (\bigcirc\Box(\varepsilon \to x = e) \vee \varepsilon))\\
&\equiv \bigcirc(\textstyle\bigvee_{n\in[1,2,\ldots,N]} \text{len}(n-1)) \wedge (\varepsilon \to x = e) \wedge \bigcirc\Box(\varepsilon \to x = e)\vee\\
&\quad \bigcirc(\textstyle\bigvee_{n\in[1,2,\ldots,N]} \text{len}(n-1)) \wedge (\varepsilon \to x = e) \wedge \varepsilon\\
&\equiv (\varepsilon \to x = e) \wedge \bigcirc(\textstyle\bigvee_{n\in[1,2,\ldots,N]} \text{len}(n-1) \wedge \Box(\varepsilon \to x = e)).
\end{aligned}$$

13) If p is the parallel statement $p\ |||\ q$: then by the hypothesis, we assume that p and q have semi-normal forms,

by Definitions 6, we have

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

By Definition 5, it is obvious that all the interleaving cases are in the semi-normal forms. Therefore, $p \,|||\, q$ can be reduced to semi-normal forms.

D) The Proof of Theorem 4

Proof By Theorem 3, we know that there exists a semi-normal form for any αPTL program p, i.e., $p \equiv (\bigvee_{i=1}^{n_1} \mathrm{Qs}_{ci} \wedge \bigcirc p_{fi}) \vee (\bigvee_{j=1}^{n_2} \mathrm{Qs}_{ej} \wedge \varepsilon)$.

1) If Qs_{ci} and Qs_{ej} are state formulae, then the semi-normal form is in normal form.
2) If $\mathrm{Qs}_{ci} \equiv ps_{1i} \wedge \bigwedge_{k=1}^{m} \langle Q_{ki}\rangle$, that is Qs_{ci} contains at least one atomic interval formula, we have

   a) If Qs_{ej} is state formulae, then $p \equiv (\bigvee_{i=1}^{n_1} ps_{1i} \wedge \bigwedge_{k=1}^{m} \langle Q_{ki}\rangle \wedge \bigcirc p_{fi}) \vee (\bigvee_{j=1}^{n_2} \mathrm{Qs}_{ej} \wedge \varepsilon)$. By the definition of atomic interval formula, 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, $p \equiv (\bigvee_{i=1}^{n_1} ps_{1i} \wedge ps_i \wedge \bigcirc ps'_i \wedge \bigcirc p_{fi}) \vee (\bigvee_{j=1}^{n_2} \mathrm{Qs}_{ej} \wedge \varepsilon) \equiv \bigvee_{i=1}^{n_1} ps_{1i} \wedge ps_i \wedge \bigcirc(ps'_i \wedge p_{fi}) \vee (\bigvee_{j=1}^{n_2} \mathrm{Qs}_{ej} \wedge \varepsilon)$, which is in normal form.

   b) If Qs_{ej} contains atomic interval formula, then it is obvious that $\mathrm{Qs}_{ej} \wedge \varepsilon \equiv$ False. Therefore, $p \equiv \bigvee_{i=1}^{n_1} \mathrm{Qs}_{ci} \wedge \bigcirc p_{fi} \equiv \bigvee_{i=1}^{n_1} ps_{1i} \wedge ps_i \wedge \bigcirc(ps'_i \wedge p_{fi})$, which is in normal form.

E) The Proof of Theorem 5

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. A similar proof is given in Ref. [11]. We omit the proof here.

F) The Proof of Theorem 7

Proof We assume that the domain of variables is finite. The proof proceeds by induction on the structure of αPTL program.

1) ε: it is already in PPTL.
2) $x = e$: it is a primitive proposition π in PPTL .
3) $x \Leftarrow e$: $x \Leftarrow e \overset{\text{def}}{=} (x = e) \wedge p_x$. Because $x = e$ and ε are in PPTL. Hence $x \Leftarrow e$ is in PPTL.
4) $x := e$: $x := e \overset{\text{def}}{=}$ skip $\wedge \bigcirc(x = e) \equiv \bigcirc(x = e \wedge \varepsilon)$. Because next operator ($\bigcirc$) is in PPTL, it is obvious that $x := e$ is in PPTL.
5) $p \wedge q$: suppose p and q are in PPTL. By induction, we have $p \wedge q$ is in PPTL.
6) $p \vee q$: suppose p and q are in PPTL. By induction, we have $p \vee q$ is in PPTL.
7) $\bigcirc p$: suppose p is in PPTL. By induction, we have $\bigcirc p$ is in PPTL.
8) p **;** q: suppose p and q are in PPTL. By definition, we have p **;** $q \overset{\text{def}}{=} \varepsilon\,\mathrm{prj}\,(p, q)$. Because projection operator (prj) is in PPTL. Hence p **;** q is in PPTL.
9) if b then p else q: suppose p and q are in PPTL. By definition, we have if b then p else $q \overset{\text{def}}{=} (b \wedge p) \vee (\neg b \wedge q)$, which is obvious in PPTL.
10) $\exists x : p(x)$: suppose $p(x)$ is in PPTL. Because the domain of x is finite, i.e., $x \in \{d_1, d_2, \ldots, d_k\}$, $k \in N$.

Hence we have $\exists x : p(x) \equiv p(d_1) \vee \cdots \vee p(d_k)$. By induction, we have $p(d_1) \vee \cdots \vee p(d_k)$ is in PPTL. Therefore, $\exists x : p(x)$ is in PPTL.

11) while b do p: suppose p is in PPTL. By definition, we have while b do $p \overset{\text{def}}{=} (b \wedge p)^* \wedge \Box(\varepsilon \rightarrow \neg b)$. Because $(b \wedge p)^*$ and $\Box(\varepsilon \rightarrow \neg b)$ can be defined in PPTL. Hence, while b do p is in PPTL.

12) $x :=^+ e$: $x :=^+ e \overset{\text{def}}{=} \bigvee_{n \in [1..\ N]} \text{len}(n) \wedge \text{fin}(x = e)$. Because $\bigvee_{n \in [1..\ N]} \text{len}(n)$ and $\text{fin}(x = e)$ can be defined in PPTL. Hence $x :=^+ e$ is in PPTL.

13) lbf(x): $\text{lbf}(x) \overset{\text{def}}{=} \neg p_x \rightarrow \exists b : (\circleddash x = b \wedge x = b)$. Suppose the domain of static variables is finite, i.e., $b \in \{n_1, n_2, \ldots, n_l\}$, $l \in N$. Because the equation $x = b$ and $\circleddash x = b$ are primitive propositions in PPTL. Hence, lbf(x) can be defined in PPTL.

14) frame(x): $\text{frame}(x) \overset{\text{def}}{=} \Box(\mathit{more} \rightarrow \bigcirc \text{lbf}(x))$. Because lbf($x$), *more* and temporal operator ($\Box$) can be defined in PPTL. Hence, frame(x) is in PPTL.

15) $\langle p \rangle$: suppose p is in PPTL. From Definition 2, program p can be reduced to two state formulas which hold at s_i and s_{i+1} respectively. By induction, the two state formulas are in PPTL. Therefore, $\langle p \rangle$ is in PPTL.

16) $p \mathbin{|||} q$: suppose p and q are in PPTL. By Definitions 5, 6, 7, we can see that $p \mathbin{|||} q$ can be rewritten into the basic constructs in PPTL. Therefore $p \mathbin{|||} q$ can be defined in PPTL.

Xiaoxiao Yang got her PhD degree from Xidian University in 2009. She is now an assistant professor in State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences. Her main research interests are in the areas of concurrent theory, temporal logic, logic programming, software modeling, and verification.

Yu Zhang got his PhD degree from école Normale Supérieure de Cachan, France in 2005 and has been an associate research professor of Institute of Software, Chinese Academy of Sciences since 2009. His research interests include formal methods, concurrent systems, information security, and programming languages.

Ming Fu got his PhD degree from University of Science and Technology of China in 2009. He is currently a postdoctoral researcher in USTC-Yale Joint Research Center for high-confidence software, University of Science and Technology of China. His main research interests lie in the areas of formal methods, concurrent program verification, and concurrency theory.

Xinyu Feng got his PhD degree from Yale University in 2007. He is now a professor in computer science at University of Science and Technology of China. His research interests are in the area of formal program verification. In particular, he is interested in developing theories, programming languages and tools to build formally certified system software, with rigorous guarantees of safety and correctness.