A Concurrent Temporal Programming Model with Atomic Blocks*

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

[1] State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences, Beijing, China
[2] University of Science and Technology of China, Hefei, 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 interval 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 practical concurrent programs.

1 Introduction

Atomic blocks in the forms of **atomic**$\{C\}$ or $\langle C \rangle$ are a high-level language construct that allows programmers to explicitly specify the atomicity of the operation C, without worrying how the atomicity is achieved by the underlying language implementation. They can be used to model architecture-supported atomic instructions, such as compare-and-swap (CAS), or to model high-level transactions implemented by software transactional memory (STM). They are viewed as a promising approach to simplifying concurrent programming in a multi-core era, and have been used in both theoretical study of fine-grained concurrency verification [16] and in modern programming languages [6,17,19] to support transactional programming.

On the other side, temporal logic has proved very useful in specifying and verifying concurrent programs [11] and has seen particular success in the temporal logic programming model, where both algorithms and their properties can be specified in the same language [1]. Indeed, a number of temporal logic programming languages have

* This research was supported by NSFC 61100063, 61161130530, 61073040, 61103023, and China Postdoctoral Science Foundation 201104162, 2012M511420.

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 22–37, 2012.
© Springer-Verlag Berlin Heidelberg 2012

been developed for the purpose of simulation and verification of software and hardware systems, such as Temporal Logic of Actions (TLA) [7], Cactus [12], Tempura [10], MSVL [8], 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 ("○" is a basic temporal operator):

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

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:

$$\mathit{CAS} \stackrel{\text{def}}{=} \langle \texttt{if}\ (x = \mathit{old})\ \texttt{then}\ \bigcirc(x = \mathit{new} \wedge \mathit{ret} = 1)\ \texttt{else}\ \bigcirc(\mathit{ret} = 0)\rangle$$

It is sufficient that the language should support 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:

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

When reasoning about programs with such transactions, we would like to have a mechanism to express the atomicity of executing transactions, without considering how the atomicity is implemented.

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 interval-based temporal logics [10,3] and we 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. 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.

Our contributions are summarized as follows:

- We extend Projection Temporal Logic (PTL) with the notion of *atomicity*, which supports the formulation of atomic executions of code blocks. The semantics is defined in the interval model and an atomic block is always interpreted over two consecutive states, with internal state transitions abstracted away. Such a formalization respects the essence of atomic execution: the environment must not interfere with the execution of the code block, and the internal execution states of atomic blocks must not be visible from outside. In fact, formulas inside atomic blocks are interpreted over separate intervals in our model, and the connection between the two levels of intervals is precisely defined.
- How values are carried through intervals is a central concern of temporal logic programming. We adopt Duan's framing technique using assignment flags and minimal models [3,4], 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.
- We define a novel notion of parallelism by considering only the interleaving of atomic blocks — parallel composition of two programs (formulas) without any atomic blocks will be translated directly into their conjunctions. For instance, $x = 1 \parallel y = 1$ will be translated into $x = 1 \wedge y = 1$, which reflects the fact that accesses from different threads to different memory locations can indeed occur simultaneously. Such a translation enforces that access to shared memory locations must be done in atomic blocks, otherwise the formulas can result in false, which indicates a flaw in the program.
- αPTL not only allows us to express various low-level memory primitives like CAS, but also makes it possible to model transactions of arbitrary granularity. We illustrate the practical use of αPTLby examples.

2 Temporal Logic

We start with a brief introduction to projection temporal logic (PTL), which was proposed for reasoning about intervals of time for hardware and software systems.

2.1 Projection Temporal Logic

PTL terms and formulas are defined by the following grammar:

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

where x is a variable, f ranges over a predefined set of function symbols, $\bigcirc e$ and $\ominus e$ indicate that term e is evaluated on the next and previous states respectively; π ranges over a predefined set of atomic propositions, and $\text{Pred}(e_1, \ldots, e_m)$ represents a

predefined predicate constructed with $e_1, \ldots, e_m$; operators *next*($\bigcirc$), *projection*(prj) and *chop plus* ($^+$) are temporal operators.

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 $I_{var} \in \mathcal{V} \to \mathcal{D} \cup \{\mathsf{nil}\}$ (nil denotes undefined values) and $I_{prop} \in \mathcal{P}rop \to \{\mathsf{True}, \mathsf{False}\}$. 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 $\sigma = \langle s_0, \ldots, s_n \rangle$ and ω if σ is infinite. An empty interval is denoted by ϵ. 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 (j can be ω). Interpretations of PTL terms and formulas are defined in Fig.1.

$$
\begin{array}{lcl}
(\sigma,i,j)[x] & = & s_i[x] \\
(\sigma,i,j)[f(e_1,\ldots,e_m)] & = & \begin{cases} f((\sigma,i,j)[e_1],\ldots,(\sigma,i,j)[e_m]) & \text{if } (\sigma,i,j)[e_h] \neq nil \text{ for all } h \\ nil & \text{otherwise} \end{cases} \\
(\sigma,i,j)[\bigcirc e] & = & \begin{cases} (\sigma,i+1,j)[e] & \text{if } i < j \\ nil & \text{otherwise} \end{cases} \\
(\sigma,i,j)[\ominus e] & = & \begin{cases} (\sigma,i-1,j)[e] & \text{if } i > 0 \\ nil & \text{otherwise} \end{cases}
\end{array}
$$

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

Fig. 1. Semantics for PTL Terms and Formulas

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

$$
\varepsilon \stackrel{\text{def}}{=} \neg \bigcirc \mathsf{True} \quad \mathsf{more} \stackrel{\text{def}}{=} \neg\varepsilon \quad p_1 \,;\, p_2 \stackrel{\text{def}}{=} (p_1, p_2)\ \mathsf{prj}\ \varepsilon \quad \Diamond p \stackrel{\text{def}}{=} \mathsf{True} \,;\, p \quad \Box p \stackrel{\text{def}}{=} \neg\Diamond\neg p
$$

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

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; $\mathsf{len}(n)$ means that the distance from the current state to the final state is n; skip specifies intervals with the length 1. $x := e$ means that at the next state $x = e$ holds and the length of the interval over which the assignment takes place is 1.

2.2 Framing Issue

Framing concerns how the value of a variable can be carried from one state to the next. Within ITL community, Duan and Maciej [3] proposed a framing technique through an explicit operator ($\mathsf{frame}(x)$), which enables us to establish a flexible framed environment where framed and non-framed variables can be mixed, with frame operators being used in sequential, conjunctive and parallel manner, and an executable version of framed temporal logic programming language is developed [4,18]. The key characteristic of the frame operator can be stated as: $\mathsf{frame}(x)$ *means that variable x keeps its old value over an interval if no assignment to x has been encountered.*

The framing technique defines a primitive proposition p_x for each variable x: intuitively p_x denotes an assignment of a new value to x — whenever such an assignment occurs, p_x must be true; however, if there is no assignment to x, p_x is unspecified, and in this case, we will use a *minimal model* [3,4] to force it to be false. We also call p_x an *assignment flag*. Formally, $\mathsf{frame}(x)$ is defined as follows:

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

Intuitively, $\mathsf{lbf}(x)$ (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 $\mathsf{lbf}(x)$ or $\mathsf{frame}(x)$.

3 Temporal Logic with Atomic Blocks

3.1 The Logic αPTL

αPTLextends projection temporal logic with atomic blocks. It can be defined by the following grammar (αPTLterms are the same as in PTL):

$$\begin{aligned} \alpha\text{PTL Formulas:}\quad p, q, p_1, p_m ::= {} & \langle p \rangle \mid \neg p \mid p \wedge q \mid \bigcirc p \mid (p_1, \ldots, p_m) prj\, q \mid \exists x.p \mid p^+ \\ & \pi \mid e_1 = e_2 \mid \mathrm{Pred}(e_1, \ldots, e_m) \end{aligned}$$

The novel construct $\langle \ldots \rangle$ 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 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 specify a single transition between two consecutive states, which the formula p will be interpreted at another interval (the inner level), which we call an *atomic interval* and must be finite, with only the first and final states being exported

to the outer level. The key point of such a two-level interval based interpretation is the exportation of values which are computed inside atomic blocks. We shall show how framing technique helps at this aspect. A few notations are introduced to support formalizing the semantics of atomic interval formulas.

- Given a formula p, let V_p be the set of free variables of p. we define formula $\mathsf{FRM}(V_p)$ as follows:

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

  $\mathsf{FRM}(V_p)$ says that each variable in the set V_p is a framing variable that allows to inherit the old value from previous states. $\mathsf{FRM}(V_p)$ 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

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

- 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 1 (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:*

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

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 $\mathsf{FRM}(V_p)$ 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.

- $p^* \stackrel{\text{def}}{=} p^+ \vee \varepsilon$ (*chop star*): either chop plus p^+ or ε;
- $p \equiv q \stackrel{\text{def}}{=} \Box(p \leftrightarrow q)$ (*strong equivalence*): p and q have the same truth value in all states of every model;
- $p \supset q \stackrel{\text{def}}{=} \Box(p \rightarrow q)$ (*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*[4] of program executions.

State formulas. State formulas are defined as follows:.

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

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

Extended state formulas. 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. 2 are valid, where Q, Q_1 and Q_2 are PTL formulas.*

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

Fig. 2. Some αPTL Laws

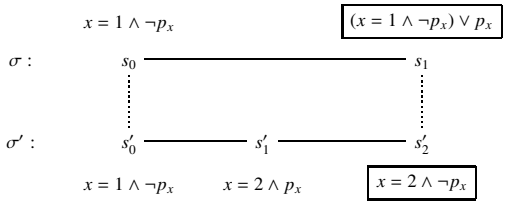
3.2 Support Framing in Atomic Interval Formulas

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 $\mathsf{FRM}(V_Q)$. However, framing inside atomic blocks must be carefully manipulated, as we use same primitive propositions (such as p_x) to track the modification of variables outside and inside

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

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

Q requires the length of the atomic interval (that is σ' in the following diagram) to be 2. Inside the atomic block, which is interpreted at the atomic interval, $\mathsf{FRM}(\{x\})$ will record that x is updated with 2 at the second state (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 $\mathsf{FRM}(\{x\})$ 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 $\neg p_x$. The conflict is well illustrated by the following diagram:



A naive merge of the last states (s_1 and s'_2) in both intervals will produce a false formula: $((x = 1 \wedge \neg p_x) \vee p_x) \wedge (x = 2 \wedge \neg p_x)$. We solve this problem by adopting different interpretation of p_x (the assignment proposition for x) at the exporting state (s_1 in σ and s'_2 in σ'): outside the atomic block (at s_1) we have $(x = 1 \wedge \neg p_x \vee p_x)$ 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 $(x = 1 \wedge \neg p_x \vee p_x) \wedge (x = 2) \equiv (x = 2) \wedge p_x$ 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 p_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.

4 Temporal Programming with Atomic Blocks

This section introduces a temporal programming language based on the logic αPTL, where for concurrent programming, we extend the interval temporal programming with atomic interval formulas, and a *parallel operator*, which captures the interleaving between processes with atomic blocks.

4.1 Expressions and Statements

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

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

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

Statements. Figure 3 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

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

Fig. 3. Statements in αPTL

state or boolean conditions; $x \Leftarrow e$, $\mathsf{lbf}(x)$ and $\mathsf{frame}(x)$ support framing mechanism and are discussed in Section 3.2; the assignment $x := e$ is as defined in Section 2.1; $p \wedge q$ 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:

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

We use a renaming method [18] to reduce a program with existential quantification.

The last three statements are new in αPTL. $\langle p \rangle$ executes p atomically. $p \,|||\, q$ 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

$$(x := x + 1; y := y + 1) \,|||\, (y := y + 2; x := x + 2).$$

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 $|||$, 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:

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

where $\mathsf{fin}(p) \stackrel{\text{def}}{=} \Box(\varepsilon \rightarrow p)$ 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.

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.2), 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) to define the interleaving semantics, and it provides an intermediate form for transforming αPTL programs into normal forms.

Definition 2 (Semi-normal Form). An αPTL program is *semi-normal* if it has the following form

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

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

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

4.3 The Interleaving Semantics with Atomic Blocks

In the following we define the parallel statement $(p \,|||\, q)$ based on the SNFs. We first define the interleaving semantics for the single future formula $(Qs \wedge \bigcirc p)$ and the single terminal formula $(Qs \wedge \varepsilon)$ in Definition 3. 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, where $q_1, \ldots, q_l$ are αPTLformulas:

$$\bigwedge_{i=1}^{l} \langle q_i \rangle \stackrel{\text{def}}{=} \begin{cases} \langle q_1 \rangle \wedge \ldots \wedge \langle q_l \rangle & \text{if } l \geq 1, l \in N \\ \mathsf{True} & \text{if } l = 0 \end{cases}$$

Definition 3. Let ps_1 and ps_2 be state formulas, $Qs_1 \equiv ps_1 \wedge \bigwedge_{i=1}^{l_1} \langle q_i \rangle$ and $Qs_2 \equiv ps_2 \wedge \bigwedge_{i=1}^{l_2} \langle q'_i \rangle$ be extended state formulas $(l_1, l_2 \geq 0)$, T_i denote $\bigcirc p_i$ or ε $(i = 1, 2)$. The interleaving semantics for $(Qs_1 \wedge T_1) \,|||\, (Qs_2 \wedge T_2)$ is inductively defined as follows.

– *case* 1:

$$(Qs_1 \wedge \bigcirc p_1) \,|||\, (Qs_2 \wedge \bigcirc p_2) \stackrel{\text{def}}{=} \begin{cases} (i)\ \ (Qs_1 \wedge Qs_2) \wedge \bigcirc (p_1 \,|||\, p_2), & \text{if } V_{q_i} \cap V_{q'_i} = \varnothing \\ (ii)\ (Qs_1 \wedge ps_2) \wedge \bigcirc (p_1 \,|||\, (\bigwedge_{i=1}^{l_2} \langle q'_i \rangle \wedge \bigcirc p_2)) & \\ \quad \vee & \text{if } V_{q_i} \cap V_{q'_i} \neq \varnothing \\ \quad (Qs_2 \wedge ps_1) \wedge \bigcirc (p_2 \,|||\, (\bigwedge_{i=1}^{l_1} \langle q_i \rangle \wedge \bigcirc p_1)) & \end{cases}$$

– *case* 2:

$$(Qs_1 \wedge \varepsilon) \,|||\, (Qs_2 \wedge T_2) \stackrel{\text{def}}{=} \begin{cases} (i)\ (Qs_1 \wedge Qs_2) \wedge T_2, & \text{if } (l_1 = l_2 = 0) \text{ or } (l_1 = 0 \text{ and } l_2 > 0 \text{ and } T_2 = \bigcirc p_2) \\ (ii)\ Qs_1 \wedge \varepsilon, & \text{if } (l_2 > 0 \text{ and } T_2 = \varepsilon) \\ (iii)\ Qs_2 \wedge T_2, & \text{if } (l_1 \neq 0) \end{cases}$$

– *case* 3: $(Qs_1 \wedge T_1) \,|||\, (Qs_2 \wedge \varepsilon)$ can be defined similarly as *case* 2.

In Definition 3, 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 $(Qs_1 \wedge \varepsilon) \,|||\, (Qs_2 \wedge T_2)$ as $(Qs_1 \wedge Qs_2 \wedge T_2)$. 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 $\bigwedge_{i=1}^{n} \langle q_i \rangle \wedge \varepsilon \equiv \mathsf{False}$. We discuss it in *case 2*(*ii*) and (*iii*) respectively. In *case 2*(*ii*), $l_2 > 0$ and $T_2 = \varepsilon$ means that $Qs_2 \wedge T_2$ is False and we define $(Qs_1 \wedge \varepsilon) \,|||\, (Qs_2 \wedge T_2)$ as $(Qs_1 \wedge \varepsilon)$; In *case 2*(*iii*), $(l_1 \neq 0)$ implies that $Qs_1 \wedge \varepsilon$ is False and $(Qs_1 \wedge \varepsilon) \,|||\, (Qs_2 \wedge T_2)$ is defined as $(Qs_2 \wedge T_2)$. Further, *case 3* can be defined and understood similarly.

In Definition 3, 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 4, we extend Definition 3 to general SNFs.

Definition 4. Let $(\bigvee_{i=1}^{n_1} Qs_{ci} \wedge \bigcirc p_{fi}) \vee (\bigvee_{j=1}^{n_2} Qs_{ej} \wedge \varepsilon)$ and $(\bigvee_{k=1}^{m_1} Qs'_{ck} \wedge \bigcirc p'_{fk}) \vee (\bigvee_{t=1}^{m_2} Qs'_{et} \wedge \varepsilon)$ $(n_1 + n_2 \geq 1, m_1 + m_2 \geq 1)$ be general SNFs. We have the following definition.

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

Definition 5. Let p and q be αPTL programs. If $p \equiv (\bigvee_{i=1}^{n_1} Qs_{ci} \wedge \bigcirc p_{fi}) \vee (\bigvee_{j=1}^{n_2} Qs_{ej} \wedge \varepsilon)$ and $q \equiv (\bigvee_{k=1}^{m_1} Qs'_{ck} \wedge \bigcirc p'_{fk}) \vee (\bigvee_{t=1}^{m_2} Qs'_{et} \wedge \varepsilon)$, then we have

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

Definition 5 means that if programs p and q have SNFs, then the interleaving semantics between p and q is the interleaving between their SNFs.

Theorem 2. For any αPTL program p, there exists a SNF q such that $p \equiv q$.

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 6 (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 $l + m \geq 1$ 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 1, we can unfolding the atomic interval formula into formulas such as $ps \wedge \bigcirc ps'$, where ps and ps' are state formulas, which are executed over two consecutive states. Thus, the extended state formulas can be reduced to the state formulas at last.

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

Theorem 4. For any satisfiable αPTL program p, there is at least one minimal model.

5 Examples

We give some examples in this section 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 reduced to $ps \wedge \varepsilon$. And, the correctness of reduction at each state is enforced by logic laws. In the following, we give three 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. In the second example, we use a two-thread concurrent program to demonstrate the reduction with the interleaving operator. Finally, the third example shows αPTL can be used to model fine-grained algorithms as well.

$$
\begin{array}{ll}
 & \mathsf{frame}(x) \wedge x = 0 \wedge y = 0 \wedge \langle \exists t : \mathsf{frame}(t) \wedge (t := x; x := t + 1; y := y + 1) \rangle \wedge \mathsf{skip} \\
\text{(L1)} & \equiv \mathsf{frame}(x) \wedge x = 0 \wedge y = 0 \wedge \langle \exists t : \mathsf{frame}(t) \wedge (t := x; x := t + 1; y := y + 1) \rangle \wedge \bigcirc \varepsilon \wedge \mathsf{more} \\
\text{(L21)} & \equiv x = 0 \wedge y = 0 \wedge \bigcirc(\mathsf{frame}(x) \wedge \mathsf{lbf}(x) \wedge \varepsilon) \wedge \langle \exists t : \mathsf{frame}(t) \wedge (t := x; x := t + 1; y := y + 1) \rangle \\
\text{(L22)} & \equiv x = 0 \wedge y = 0 \wedge \bigcirc(\mathsf{lbf}(x) \wedge \mathsf{lbf}(y) \wedge \varepsilon) \wedge \langle \exists t : \mathsf{frame}(t) \wedge (t := x; x := t + 1; y := y + 1) \rangle \\
\text{(L25)} & \equiv \ldots \wedge \langle \mathsf{frame}(x) \wedge \mathsf{frame}(y) \wedge \exists t : \mathsf{frame}(t) \wedge (t := x; x := t + 1; y := y + 1) \rangle \\
 & \stackrel{z}{\equiv} \ldots \wedge \langle \mathsf{FRM}(\{x, y, z\}) \wedge (z := x \,;\, x := z + 1 \,;\, y := y + 1) \rangle \\
 & \equiv \ldots \wedge \langle \mathsf{FRM}(\{x, y, z\} \wedge ((\bigcirc z = x \wedge \mathsf{skip}) \,;\, (\bigcirc x = z + 1 \wedge \mathsf{skip}) \,;\, (\bigcirc y = y + 1 \wedge \mathsf{skip}))) \rangle \\
\text{(L11,12)} & \equiv \ldots \wedge \langle \mathsf{FRM}(\{x, y, z\} \wedge \bigcirc z = 0 \wedge \bigcirc(\bigcirc x = z + 1 \wedge \mathsf{skip} \,;\, (\bigcirc y = y + 1 \wedge \mathsf{skip}))) \rangle \\
\text{(L21)} & \equiv \ldots \wedge \langle \bigcirc(\mathsf{FRM}(\{x, y, z\} \wedge \mathsf{lbf}(x) \wedge \mathsf{lbf}(y) \wedge \mathsf{lbf}(z) \wedge z = 0 \wedge \\
 & \qquad (\bigcirc x = z + 1 \wedge \mathsf{skip} \,;\, (\bigcirc y = y + 1 \wedge \mathsf{skip}))) \rangle \\
\text{(L11,12,17,19)} & \equiv \ldots \wedge \langle \bigcirc(\mathsf{FRM}(\{x, y, z\} \wedge (x = 0 \wedge \neg p_x) \wedge (y = 0 \wedge \neg p_y) \wedge (z = 0 \wedge p_z) \\
 & \qquad \wedge \bigcirc x = z + 1 \wedge \bigcirc (\bigcirc y = y + 1 \wedge \mathsf{skip})) \rangle \\
\text{(L21)} & \equiv \ldots \wedge \langle \bigcirc((x = 0 \wedge \neg p_x) \wedge (y = 0 \wedge \neg p_y) \wedge (z = 0 \wedge p_z) \\
 & \qquad \wedge \bigcirc (\mathsf{FRM}(\{x, y, z\} \wedge x = 1 \wedge \mathsf{lbf}(x) \wedge \mathsf{lbf}(y) \wedge \mathsf{lbf}(z) \wedge \bigcirc y = y + 1 \wedge \bigcirc \varepsilon))) \rangle \\
\text{(L18-22)} & \equiv \ldots \wedge \langle \bigcirc((x = 0 \wedge \neg p_x) \wedge (y = 0 \wedge \neg p_y) \wedge (z = 0 \wedge p_z) \wedge \bigcirc((x = 1 \wedge p_x) \\
 & \qquad \wedge (y = 0 \wedge \neg p_y) \wedge (z = 0 \wedge \neg p_z) \wedge \bigcirc \mathsf{lbf}(x) \wedge \mathsf{lbf}(y) \wedge \mathsf{lbf}(z) \wedge y = 1 \wedge \varepsilon))) \rangle \\
\text{(L18-22)} & \equiv \ldots \wedge \langle \bigcirc((x = 0 \wedge \neg p_x) \wedge (y = 0 \wedge \neg p_y) \wedge (z = 0 \wedge p_z) \wedge \bigcirc((x = 1 \wedge p_x) \\
 & \qquad \wedge (y = 0 \wedge \neg p_y) \wedge (z = 0 \wedge \neg p_z) \wedge \bigcirc(x = 1 \wedge \neg p_x \wedge z = 0 \wedge \neg p_z \wedge (y = 1 \wedge p_y) \wedge \varepsilon)))) \rangle \\
\text{(Def.1)} & \equiv x = 0 \wedge y = 0 \wedge \bigcirc(\mathsf{lbf}(x) \wedge \mathsf{lbf}(y) \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. 4. Rewriting Procedure for Prog_1

Example 1. 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 $x = 0, y = 0$. The above transactional code can be defined with αPTL as follows.

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

Fig. 4 presents the detailed rewriting procedure for the above transactional code.

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

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

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

Fig. 5. Money Transfer using Transactions

Example 2. Let T_1 denote the money transfer from the account A to the account B and T_2 denote the money transfer from the account B to the account A. This example can be simulated in αPTL shown in Fig. 5.

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

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

where p_1 and p_2 are programs that will be executed at the next state and we omit their reduction details here. Further, by Definition 3, we have

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

Now $T_1 \,|||\, T_2$ is reduced to the semi-normal form like $Q_s \wedge \bigcirc p$. Hence, based on the semi-normal form, we can further reduce Prog_2 to its normal form by interpreting the atomic interval formulas.

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, whose atomicity is enforced by the hardware. The behaviors of these machine primitives can be modeled with *atomic interval formulas* in αPTL like this: "$\langle \bigvee_{i=1}^{n} (ps_i \wedge \bigcirc ps'_i \wedge \mathsf{skip}) \rangle$". 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.

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

6 Related Works and Conclusions

Several researchers have proposed extending logic programming with temporal logic: Tempura [10] and MSVL [8] based on interval temporal logic, Cactus [12] based on branching-time temporal logic, XYZ/E [14], Templog [1] based on linear-time temporal logic. While most of these temporal languages adopt a Prolog-like syntax and programming style due to their origination in logic programming, interval temporal logic programming languages such like Tempura 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 PTL), but more importantly, the imperative execution of interval temporal programs are well founded on the solid theory of framing and minimal model semantics. Indeed, interval temporal logic programming languages have narrowed the gap between the logic and imperative programming languages, and facilitate specifying, verifying and developing reactive systems in a more efficient and uniform way [2,5,13,15].

The fact that none of existing temporal logic programming language supports concurrent logic programming with atomic blocks, motivates our work on αPTL— the new logic allows us to model programs with the fundamental mechanism in modern concurrent programming. We have chosen to base our work on interval temporal logic and introduce a new form of formulas — atomic interval formulas, which is powerful enough to specify synchronization primitives with arbitrary granularity, as well as their interval-based semantics. The choice of PTL also enables us to make use of the *framing* technique and develop an executive language on top of the logic consequently. Indeed, we show that framing works smoothly with our interval semantics for atomicity. In the near future work, we shall focus on the practical use of αPTL and extend the theory and the existing tool MSVL [8] to support the specification and verification of various temporal properties in synchronization algorithms with complex data structure such as the linked list queue [9].

References

1. Abadi, M., Manna, Z.: Temporal logic programming. Journal of Symbolic Computation 8, 277–295 (1989)
2. Derrick, J., Schellhorn, G., Wehrheim, H.: Verifying Linearisability with Potential Linearisation Points. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 323–337. Springer, Heidelberg (2011)
3. Duan, Z., Koutny, M.: A framed temporal logic programming language. Journal of Computer Science and Technology 19, 341–351 (2004)

4. Duan, Z., Yang, X., Koutny, M.: Semantics of Framed Temporal Logic Programs. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 356–370. Springer, Heidelberg (2005)
5. Duan, Z., Zhang, N.: A complete axiomatization of propositional projection temporal logic. In: TASE 2008, pp. 271–278. IEEE Computer Science (2008)
6. Harris, T.L., Fraser, K.: Language support for lightweight transactions, pp. 388–402 (2003)
7. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. 16(3), 872–923 (1994)
8. Ma, Y., Duan, Z., Wang, X., Yang, X.: An interpreter for framed tempura and its application. In: Proc. 1st Joint IEEE/IFIP Symp. on Theoretical Aspects of Soft. Eng. (TASE 2007), pp. 251–260 (2007)
9. Michael, M.M., Scott, M.L.: Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. J. Parallel Distrib. Comput. 51(1), 1–26 (1998)
10. Moszkowski, B.C.: Executing temporal logic programs. Cambridge University Press (1986)
11. Pnueli, A.: The Temporal Semantics of Concurrent Programs. In: Kahn, G. (ed.) Semantics of Concurrent Computation. LNCS, vol. 70, pp. 1–20. Springer, Heidelberg (1979)
12. Rondogiannis, P., Gergatsoulis, M., Panayiotopoulos, T.: Branching-time logic programming: The language cactus and its applications. Computer Language 24(3), 155–178 (1998)
13. Schellhorn, G., Tofan, B., Ernst, G., Reif, W.: Interleaved programs and rely-guarantee reasoning with itl. In: Proc. TIME 2011, pp. 99–106. IEEE Computer Science (2011)
14. Tang, C.: A temporal logic language oriented toward software engineering – introduction to XYZ system (I). Chinese Journal of Advanced Software Research 1, 1–27 (1994)
15. Tian, C., Duan, Z.: Model Checking Propositional Projection Temporal Logic Based on SPIN. In: Butler, M., Hinchey, M.G., Larrondo-Petrie, M.M. (eds.) ICFEM 2007. LNCS, vol. 4789, pp. 246–265. Springer, Heidelberg (2007)
16. Vafeiadis, V., Parkinson, M.: A Marriage of Rely/Guarantee and Separation Logic. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 256–271. Springer, Heidelberg (2007)
17. Yang, N., Adam, W., Adl-Tabatabai, Bach, M., Berkowits, S.: Design and implementation of transactional constructs for C/C++. In: Proceedings OOPSLA 2008, pp. 195–212 (September 2008)
18. Yang, X., Duan, Z.: Operational semantics of framed tempura. Journal of Logic and Algebraic Programming 78(1), 22–51 (2008)
19. Zyulkyarov, F., Harris, T., Unsal, O.S., Cristal, A., Valeroh, M.: Debugging programs that use atomic blocks and transactional memory. In: Proc. PPoPP 2010, pp. 57–66 (2010)