The Specification and Verification of Concurrent Systems with Chronolog(MC)

Chuchang Liu  Mehmet A. Orgun

Department of Computing
Macquarie University
NSW 2109, Australia

E-mail: {cliu,mehmet}@krakatoa.mpce.mq.edu.au

Abstract

Chronolog(MC) is an extension of logic programming based on a temporal logic with granularity of time, which is suitable for describing those systems where dynamic changes are essential. This paper presents an approach to using Chronolog(MC) as a specification language to describe concurrent systems. The approach provides a general formalism for describing the behavioral properties of concurrent systems through using Chronolog(MC) programs, and it also provides a means for specifying safety and liveness properties of a system based on TLC, the temporal logic of Chronolog(MC). We also develop a verification methodology based on TLC for proving that a system meets its temporal specification.

1 Introduction

Temporal logic is in particular suitable for reasoning about a changing world. Recently, there is a substantial interest in using temporal logic for the formal specification of concurrent systems, and it has been shown that temporal logic is a successful tool in reasoning about the behavior of concurrent systems [1, 3, 4, 5, 10].

Chronolog(MC) [7] is an extension of logic programming based on a temporal logic with the granularity of time. In Chronolog(MC), granularity of time is supported by multiple clocks. This paper presents an approach to using Chronolog(MC) as a specification language to describe concurrent systems. The approach provides a general formalism for describing the behavioral properties of concurrent systems through using Chronolog(MC) programs, and it also provides a means for specifying safety and liveness properties of a system based on TLC. TLC is an extension of temporal logic in which each predicate symbol, hence each formula, is associated with a clock. In this approach, the behavior of concurrent systems can be specified as raw Chronolog(MC) programs, and the properties which any proposed implementation of the system should satisfy are expressed as TLC formulas. We also develop a verification methodology based on TLC, the temporal logic of Chronolog(MC), for proving that a system meets its temporal specification. We show that timing properties of a system can be directly verified based on its corresponding raw Chronolog(MC) program.

The temporal assertions we use in specifications are simple as well as intuitive. Therefore, not only can the specifications be easily understood by the user without knowledge of the details of temporal logic, but they can also be easy to write by the designer.

The structure of this paper is as follows. A brief introduction to Chronolog(MC) programming is provided in section 2. Section 3 gives the framework of our approach for specifying and verifying timing properties of a concurrent system. In section 4, we discuss the use of raw Chronolog(MC) programs to specify concurrent systems, and, in section 5, we discuss the use of TLC formulas to represent timing properties of concurrent systems. In section 6, several verification techniques are presented. Last section concludes the paper with several remarks.

2 Temporal Programming

Chronolog [9] is an extension of logic programming based on a linear-time temporal logic [2], in which linear time is represented by a global clock, namely the sequence of natural numbers < 0, 1, 2, ... >. All formulas are actually defined on the same global clock. Intending to extend the temporal logic Chronolog is based on, we proposed that in a temporal logic, formulas could be allowed to be de-
defined on different clocks [7]. That is, every predicate symbol can be assigned a local clock, so that each formula can also be clocked.

In [7], a local clock is defined as a subsequence of the global clock, that is, a strictly increasing sequence of natural numbers. In particular, the global clock and the empty clock, denoted by $gck$ and $<>$, respectively, are also local clocks.

This extended logic is called Temporal Logic with Clocks (TLC). Based on TLC, we furthermore proposed an extension of temporal language Chronolog, called Chronolog(MC) (i.e., Chronolog with Multiple Clocks) [7, 6], in which the presentation of multiple granularity of time in a program is explicitly given by a clock definition and a clock assignment. The user is free to choose the granularity for each predicate symbol through clock assignments and definitions. Therefore, the multiple granularity of time is more flexible in time representation and describing timing properties of systems.

A Chronolog(MC) program consists of three components: a clock definition, a clock assignment and a program body. The program body consists of rules and facts. The clock definition is an ordinary Chronolog program and it specifies all the local clocks involved in the program body. The clock assignment assigns clocks for all the predicate symbols in the program body. The following is a very simple Chronolog(MC) program, which specifies a computational system consisting of two independent processes $p$ and $q$ running on their own local clocks.

```plaintext
% CLOCK DEFINITION (ck1, ck2) %
first ck1(0).
next ck1(N) ← ck1(N), N is M+2.
first ck2(1).
next ck2(N) ← ck2(N), N is M+3.

% CLOCK ASSIGNMENT (ck) %
is_ck(p, ck1).
is_ck(q, ck2).

% PROGRAM BODY %
first p(0).
next p(X) ← p(Y), X is Y+2.
first q(2).
next q(X) ← q(Y), X is Y+2.
```

In Chronolog(MC), there are two temporal operators, first and next, which refer to the initial and the next moment in time with respect to given clocks respectively. Now we may pose the following fixed-time goal (goals within the scope of the operator first are called fixed-time):

$$< - \text{first } p(X), \text{first } q(Y).$$

By the clocked temporal resolution [7], the answer is "$X=4$ and $Y=4$". The conclusion can easily be obtained by the following idea: Due to the fact that $ck1=\langle 0, 2, 4, 6, \ldots \rangle$ and $ck2=\langle 1, 4, 7, 10, \ldots \rangle$, the local clock of the above goal is the greatest lower bound of $ck1$ and $ck2$, that is, $\langle 4, 10, \ldots \rangle$. Therefore, to answer the above query, what we need to do is to obtain answers to the following two queries:

$$< - \text{first } \text{next } p(X).$$
$$< - \text{first } q(Y).$$

We can easily obtain that the answer to the first query is "$X=4$", and the answer to the second query is "$Y=4$". Therefore the answer to the original goal is "$X=4$ and $Y=4$".

### 3 Framework

Concurrent programs can be specified in terms of their ongoing behavior; the temporal logic TLC provides an expressive and natural formalism for specifying their behavior.

We first give the concept of raw Chronolog(MC) programs. As shown above, in a Chronolog(MC) program, the clock definition and assignment are given by ordinary Chronolog program clauses. If the clock definition and/or the clock assignment of a Chronolog(MC) program are not given by ordinary program clauses, but only some requirement on clocks and/or an explicit map which assigns local clocks to predicate symbols are provided in it, then we say that the Chronolog(MC) program is raw.

Our framework for specifying and verifying timing properties of a concurrent system is based on the following components:

- A system description which expresses proposed implementations that may be involved in an implementation environment and program code of a certain form. In our method, program code can be written in any programming language.

- A raw Chronolog(MC) program which describes the behavior of the concurrent system and its implementation environment.

- TLC formulas to express properties that can be derived from the raw Chronolog(MC) program. Such properties should be satisfied by any proposed implementation that satisfies
the specification provided by the raw Chronolog(MC) program.

- Verification techniques to prove that the properties expressed as TLC formulas can be derived from the raw Chronolog(MC) program. That is, we need to provide verification techniques to prove that the proposed implementation meets its specification.

In summary, proposed implementations of a concurrent system can be specified as a raw Chronolog(MC) program, and the requirements expressed as the properties that any proposed implementation should satisfy can be represented by TLC formulas. Therefore, if we can derive such properties from the specification of the system, i.e., the raw Chronolog(MC) program, then any proposed implementation which satisfies the specification must satisfy the requirements we want.

4 Specification by Raw Chronolog(MC) Programs

Let us consider a system which starts with variables X and Y both equal to 0 and repeatedly increments either X or Y by 1 (but not both at the same time), choosing nondeterministically between them. That is, the variables are incremented in an arbitrary order, but each incremented infinitely often.

This system might be viewed as consisting of two parallel processes, p and q, which are running in a time-shared system and repeatedly executing the instructions r and s. In a conventional programming language, we may have the following program:

initially X = 0, Y = 0
cobegin
  loop r: X := X + 1 end loop
  loop s: Y := Y + 1 end loop
coend

To specify the system, we now define the following predicates:

state.X(X): X is the current value of the state variable X.

state.Y(Y): Y is the current value of the state variable Y.

change.X: The variable X gets a change.

change.Y: The variable Y gets a change.

active.p(L): Instruction L is currently fetched and executed by the process p.

active.q(L): Instruction L is currently fetched and executed by the process q.

Thus, we obtain the following raw Chronolog(MC) program which specifies the behavior of the system:

{clock definition}
ck1, ck2: infinite.
ck1∩ck2<>.

{clock assignment}
ck(active.p)=ck1.
ck(active.q)=ck2.
ck(p)=gck, when p ≠ active.p or active.q.

{program body}
first state.x(0).
first state.y(0).
first active.p(r).
first active.q(s).
next active.p(r) <-- active.p(r).
next active.q(s) <-- active.q(s).
change.x <-- active.p(r).
change.y <-- active.q(s).
next state.x(X1) <-- state.x(X),
                 change.x,
                 X1=X+1.
next state.y(Y1) <-- state.y(Y),
                 change.y,
                 Y1=Y+1.
next state.x(X) <-- state.x(X),
                 ~change.x.
next state.y(Y) <-- state.y(Y),
                 ~change.y.

In the raw Chronolog(MC) program, the clock definition and clock assignment are used to specify the environment of the proposed implementation, whereas the program body gives a description of initial conditions and rules which are involved in the behavior of the system. In Chronolog(MC), we are allowed to use negations in the body of a rule clause [8]. Negated atoms are evaluated using the negation as failure proof rule.

Intuitively, the program describes the behavior of the system as follows: From the view based on the global clock, the variable X is initialized to 0, and at any moment, if the process p is activated to execute the instruction r, then at the next moment the value of X will be incremented by 1, otherwise there is no change for the value; on other hand, from the view based on the local clock of the process p, at the initial moment in time, p has been activated to execute r and at any moment if it is activated to execute r, then at next moment it will be also activated to execute r. A similar explanation can be made for the variable y and process q.
5 Representing Properties with TLC Formulas

In this section, we present a method to describe requirements for implementing a system. The requirements are usually expressed as several properties which can be represented by TLC formulas.

In order to enrich TLC, we assume recursive definitions of two modalities, $\Box$ and $\Diamond$, as follows:

$$\Box A \equiv (\text{first } A) \land (\Box \text{next } A)$$
$$\Diamond A \equiv (\text{first } A) \lor (\Diamond \text{next } A)$$

The intuitive meanings of the modalities are as follows. The formula $\Box A$ is true at time $t \in ck(A)$, the local clock of $A$ just in the case $A$ is true at all moments in time on $ck(A)$; the formula $\Diamond A$ is true at time $t \in ck(A)$, the local clock of $A$ just in the case $A$ is true at some moment in time on $ck(A)$. Therefore, we read $\Box$ as "always" and $\Diamond$ as "sometimes".

As an example, we now discuss the safety and liveness properties of the system given in previous section. To do this, we first introduce an auxiliary predicate as follows:

$$\text{state}(X,Y) \equiv \text{state}_x(X) \land \text{state}_y(Y)$$

Thus, we have the following property which should be satisfied by any proposed implementation:

$$(P_1) \quad \Box((\forall X)(\forall Y)(\text{state}(X,Y) \rightarrow \neg \text{next}(X,Y+1) \lor \neg \text{next}(Y,X+1) \lor \neg \text{next}(X,Y)))$$

The formula as property $P_1$ is actually a simple form of the formula

$$(P'_1) \quad \Box((\forall X)(\forall Y)(\text{state}(X,Y) \land \text{state}(X,Y) \rightarrow \neg \text{next}(X,Y+1) \lor \neg \text{next}(X,Y+1) \lor \neg \text{next}(Y,X)))$$

The property $P_1$ states that at any moment in time, if the state values of variables $x$ and $y$ are in $X$ and $Y$ respectively, then at the next moment the value of $x$ is $X+1$ and the value of $y$ is $Y$, or the value of $y$ is $Y+1$ and the value of $x$ is $X$ or the values of both $x$ and $y$ are not changed. That is, intuitively, for any proposed implementation of the system, an execution of each program's atomic operation does only one of three things: increasing $x$ only, increasing $y$ only and no change for both $x$ and $y$.

In Lamport's work [5], which uses TLA (Temporal Logic of Action) to specify concurrent systems, the specification of the system could be written in the following form:

$$\text{Init} = (x = 0) \land (y = 0)$$
$$M_1 = (x' = x + 1) \land (y' = y)$$
$$M_2 = (y' = y + 1) \land (x' = x)$$
$$M_3 = (x' = x) \land (y' = y)$$
$$M = M_1 \lor M_2 \lor M_3$$

$$\mathcal{F} = \text{Init} \land \Box[\text{M}_1 \land \text{W}_2 \land \text{M}_3]$$

The TLA formula $\mathcal{F}$ describes the system. That is, it gives the properties which any proposed implementation of the system should satisfy. Here $\Box[\text{M}_1 \land \text{W}_2 \land \text{M}_3]$ represents that every step is an $M_1$ step or an $M_2$ step or a stuttering step $M_3$. The formula $\text{W}_2 \land \text{M}_3$ asserts that infinitely many $M_1$ and $M_2$ steps occur. $\text{Init} \land \Box[\text{M}_1 \land \text{W}_2 \land \text{M}_3]$ is a safety property; while the formula $\text{W}_2 \land \text{M}_3$, which asserts that the system never terminates, is its liveness property.

We denote the initial state:

$$(I) \quad \text{first state}(0,0).$$

Thus, in our approach, $I \land P_1$ or $I \land P'_1$ expresses a safety property which corresponds to the one provided by Lamport. Whereas the liveness property can be represented through the following formulas:

$$(P_2) \quad \Box((\exists X)(\exists Y)(\text{state}(X,Y) \land \neg \text{next}(X,Y)))$$

We have that $P_2 \land P_3$ expresses the liveness property which corresponds that of the Lamport's. Therefore, the formula $I \land P_1 \land P_2 \land P_3$ actually describes the requirements to any proposed implementation of the system given in last section.

6 Verification Techniques

According to the above discussion, in order to prove any proposed implementation of the system satisfies the safety and liveness properties, we only need to show that the formula $I \land P_1 \land P_2 \land P_3$ can be derived from $RCP$. That is, we want to prove that
\[
RCP \vdash I \land P_1 \land P_2 \land P_3.
\]

The whole proof depends on proving the following four components: \( RCP \vdash I \), \( RCP \vdash P_1 \), \( RCP \vdash P_2 \), and \( RCP \vdash P_3 \). The proof for the first component is trivial. To prove other components, we need to use some verification techniques.

The verification techniques we propose in this paper include: the use of induction, the use of fixed-time rules and the use of local reasoning. The details of these techniques will be discussed in the full paper. As an example, we outline the proof procedure for the property \( P_1 \).

To do this, we need the following assertions:

\[
\begin{align*}
RCP &\vdash \text{first next}(n) \text{ active.p}(r), \\
&\text{for any natural number } n. \\
RCP &\vdash \text{first next}(n) \text{ active.q}(a), \\
&\text{for any natural number } n. \\
\end{align*}
\]

We now prove the first assertion by induction on \( n \). When \( n = 0 \), the assertion obviously holds. Assume that when \( n = k \) the assertion is true, i.e., we have that the formula

\[
\text{first next}(k) \text{ active.p}(r).
\]

can be derived from \( RCP \). Then we can obtain a list of formulas derived from \( RCP \) as follows:

\[
\begin{align*}
\text{next active.p} &\leftarrow \text{active.p}(r). \\
\text{first next}(k + 1) \text{ active.p}(r) &\leftarrow \\
&\text{first next}(k) \text{ active.p}(r). \\
\text{first next}(k + 1) &\text{ active.p}(r). \\
\end{align*}
\]

The last formula is exactly what we want to show. The proof for the second assertion is similar.

Now consider any moment \( t \in gck \). We assume that \( \text{state.x}(X) \land \text{state.y}(Y) \) is true at \( t \). That is, we have the formulas

\[
\begin{align*}
\text{first next}(t) &\text{ state.x}(X). \\
\text{first next}(t) &\text{ state.y}(Y). \\
\end{align*}
\]

Because \( ck_1 \land ck_2 \leftarrow \), there are only three cases we need to consider: (1) \( t \in ck_1 \) but \( t \notin ck_2 \), (2) \( t \in ck_2 \) but \( t \notin ck_1 \), (3) \( t \notin ck_1 \) and \( t \notin ck_2 \).

For further discussion, we need the the notation \( \text{rank}(t, ck_i) \), which is defined as follows: Given a local clock \( ck_i = \langle t_0, t_1, t_2, \ldots \rangle \), we call \( n \) the rank of \( t_n \) on \( ck_i \), written as \( \text{rank}(t_n, ck_i) = n \).

Inversely, we write \( t_n = ck_i^{(n)} \), which means that \( t_n \) is the moment in time on \( ck_i \) whose rank is \( n \).

We now consider case 1. Without loss the generality, assume that \( k = \text{rank}(t, ck_1) \). Thus, by the above assertion we have proved that we have the formula \( \text{first next}(k) \text{ active.r} \) derived from \( RCP \). And due to the fact that \( t \notin ck_2 \), there is no such number \( k \) so that \( k = \text{rank}(t, ck_2) \). Therefore, for any \( k \), \( \text{first next}(k) \text{ active.s} \) is not true at \( t \), so we have that \( \text{first next}(t) \neg \text{change.y} \) is true at \( t \). Thus, we have the following derivation:

\[
\begin{align*}
(1) &\text{first next}(k) \text{ active.p}(r). \\
(2) &\text{change.x} \leftarrow \text{active.p}(r). \\
(3) &\text{first next}(k) \text{ change.x} \leftarrow \\
&\text{first next}(k) \text{ active.p}(r). \\
(4) &\text{first next}(t) \text{ change.x}. \\
(5) &\text{first next}(t) \text{ equal}(X_1, X + 1). \\
(6) &\text{first next}(t) \text{ state.x}(X). \\
(7) &\text{next state.x}(X_1) \leftarrow \text{state.x}(X), \\
&\text{change.x, equal}(X_1, X + 1). \\
(8) &\text{first next}(t + 1) \text{ state.x}(X_1) \leftarrow \\
&\text{first next}(t) \text{ state.x}(X), \\
&\text{first next}(t) \text{ change.x,} \\
&\text{first next}(t) \text{ equal}(X_1, X + 1). \\
(9) &\text{first next}(t + 1) \text{ state.x}(X_1). \\
(10) &\text{first next}(t) \neg \text{change.y}. \\
(11) &\text{first next}(t) \text{ state.y}(Y). \\
(12) &\text{next state.y}(Y) \leftarrow \text{state.y}(Y), \neg \text{change.y}. \\
(13) &\text{first next}(t + 1) \text{ state.y}(Y) \leftarrow \\
&\text{first next}(t) \text{ state.y}(Y), \\
&\text{first next}(t) \neg \text{change.y}. \\
(14) &\text{first next}(t + 1) \text{ state.x}(X_1). \\
(15) &\text{first next}(t + 1) \text{ state.x}(X_1) \land \text{state.y}(Y)). \\
\end{align*}
\]

In this derivation, the formula (4) is derived from formulas (1) and (3) by using a fixed-time rule as follows:

\[
\begin{align*}
\text{first next}(k)(B \rightarrow A), \text{where } k = \text{rank}(t, ck_{B \rightarrow A}) \\
\text{first next}(m)B, \text{where } m = \text{rank}(t, ck_B) \\
\text{first next}(n)A, \text{where } n = \text{rank}(t, ck_A) \\
\end{align*}
\]

Here \( ck_F \) denotes the local clock associated with the formula \( F \).

The formula (15) denotes that at the next moment \( t + 1 \), the values of \( x \) and \( y \) are \( X + 1 \) and \( Y \) respectively. In the same manner, we can show that in case 2, at the next moment \( t + 1 \), the values of \( x \) and \( y \) are \( X \) and \( Y + 1 \) respectively; and in case 3, at the next moment \( t + 1 \), the values of \( x \) and \( y \) are still \( X \) and \( Y \) respectively. That is, we have proved that the property \( P_1 \) can be derived from \( RCP \).

In the same way, we can prove that \( RCP \) also satisfies the properties \( P_2 \) and \( P_3 \).
7 Concluding Remarks

We have presented an approach to specifying and verifying temporal properties of concurrent systems. In our approach, we use raw Chronolog(MC) programs to describe the behavior of concurrent systems. The description of the implementation environment of a system usually involves defining the clocks associated with concurrent processes of the system. Since the user is free to choose any clock for each predicate, Chronolog(MC) is more expressive in describing the behavior of concurrent systems.

Temporal properties as requirements for implementing such systems can be specified as TLC formulas. As an example, we have discussed the safety and liveness properties of the system given in section 4. For the system, we may have some other properties which are also easily expressed as TLC formulas. For example, the type-correctness property discussed in [5], which asserts that \(x\) and \(y\) are always natural numbers, can be represented as follows:

\[
P_4 \equiv (\forall x)(\forall y)(\text{state}(x, y) \rightarrow \text{nat}(x) \land \text{nat}(y))
\]

where the predicate \(\text{nat}(x)\) denotes that \(x\) is a natural number. The property says: it is always true that at any moment in time, for all \(x\) and for all \(y\), if the state values of variables \(x\) and \(y\) are \(x\) and \(y\) respectively, then \(x\) and \(y\) are natural numbers. This assertion may also be expressed in a stronger form as follows:

\[
P_5 \equiv (\exists x)(\exists y)(\text{state}(x, y) \land \text{nat}(x) \land \text{nat}(y))
\]

Both \(P_4\) and \(P_5\), which are invariance properties of the system, are obvious corollaries of \(P_1 \land P_2 \land P_3\). In fact, all such properties of a concurrent system can be directly derived from its raw Chronolog(MC) programs. The verification techniques we propose in this paper make the proof procedure simpler.

Future work includes the study of general formal specification of concurrent systems based on TLC and an automated verification methodology for such systems. We may also consider the applications of the approach to specify reactive systems.

Acknowledgements

The work presented in this article has been supported in part by an Australian Research Council (ARC) Grant. C. Liu has been supported by an Australian Postgraduate Award (APA) and an MPCE Postgraduate Research Scholarship at Macquarie University.

References

Appendix:
Temporal Logic TLC

We give a brief introduction to the temporal logic TLC, including the definition of local clocks, and the syntax and semantics of TLC. In particular, the axioms and inference rules are given.

Clocks, syntax

The global clock is the increasing sequence of natural numbers: \( < 0, 1, 2, \ldots > \), and a local clock is a subsequence of the global clock. In other words, a local clock is a strictly increasing sequence of natural numbers, either infinite or finite: \( < t_0, t_1, t_2, \ldots > \).

Let \( CK \) be the set of all clocks and \( \subseteq \) be an ordering relation on the elements of \( CK \). For any \( c_{k_1}, c_{k_2} \in CK \), we define

\[
\begin{align*}
ck_1 \cap ck_2 & \overset{def}{=} g.l.b.(ck_1, ck_2) \\
ck_1 \cup ck_2 & \overset{def}{=} l.u.b.(ck_1, ck_2)
\end{align*}
\]

where \( ck_1, ck_2 \in CK \), g.l.b. stands for “the greatest lower bound” and l.u.b. for “the least upper bound” under the relation \( \subseteq \).

TLC formulas are constructed by the following rules:
1. Any formula of first-order logic is a formula of TLC;
2. If \( A \) is a formula, so are \( \text{first} \ A \) and \( \text{next} \ A \).

We use a clock assignment to assign a local clock for each predicate symbol. A clock assignment \( ck \) is a map from the set of predicate symbols to the set of clocks. The clock associated with a predicate symbol \( p \) is denoted by \( ck_p \). For any formula \( A \), the clock \( ck_A \) is determined by the clocks of predicate symbols appearing in \( A \). In other words, let \( Pred \) be the set of predicate symbols which appear in \( A \). Then we define that \( ck_A = \bigcap_{p \in Pred} ck_{p} \). That is, every TLC formula can be clocked.

Given a local clock \( ck_i = < t_0, t_1, t_2, \ldots > \), we define the rank of \( t_n \) on \( ck_i \) to be \( n \), written as \( \text{rank}(t_n, ck_i) = n \).

Inversely, we write \( t_n = ck_i^{(n)} \), which means that \( t_n \) is the moment in time on \( ck_i \) whose rank is \( n \).

Semantics

In TLC, the semantics of formulas with logical connectives are defined in the usual way, but with respect to local clocks [7]. Here we only give the meaning of temporal operators:

- For any \( t \in ck_A \), \( \text{first}A \) is true at \( t \) if and only if \( A \) is true at \( ck_A^{(0)} \).
- For any \( t \in ck_A \), \( \text{next} A \) is true at \( t \) if and only if \( A \) is true at \( ck_A^{(i+1)} \), where \( i = \text{rank}(t, ck_A) \).

Proof system

The proof system for TLC consists of the following axioms and inference rules.

Axioms

\[
\begin{align*}
A1. & \vdash \text{first} A \leftrightarrow \text{first} A. \\
A2. & \vdash \text{next} A \leftrightarrow \text{first} A. \\
A3. & \vdash \text{first} \ (\neg A) \leftrightarrow \neg (\text{first} A). \\
A4. & \vdash \text{next} \ (\neg A) \leftrightarrow \neg (\text{next} A). \\
A5. & \vdash \text{first} \ (\forall x)(A) \leftrightarrow (\forall x)(\text{first} A). \\
A6. & \vdash \text{next} \ (\forall x)(A) \leftrightarrow (\forall x)(\text{next} A). \\
A7. & \vdash ck(A \land B) \ 	ext{first}(A \land B) \leftrightarrow (\text{first} A) \land (\text{first} B). \\
A8. & \vdash ck(A \land B) \ 	ext{next} \ (A \land B) \leftrightarrow (\text{next} A) \land (\text{next} B).
\end{align*}
\]

Inference rules

In addition to substitution and Modus Ponens, we have the following rules for temporal operators.

R1. If \( \vdash ck_A \), then \( \vdash \text{first} A \), when \( ck_A \) is non-empty.

R2. If \( \vdash ck_A \), then \( \vdash \text{next} A \), when \( ck_A \) is infinite.

R3. If \( \vdash ck \ B \rightarrow A \), \( \vdash ck \ t \ B \) and \( t \in ck_B \rightarrow A \), then \( \vdash ck \ t \ A \).

R1 and R2 are called the temporal operator introduction rules, and R2 holds only when there is always a next moment in time on the clock \( ck_A \). Using R3, we are allowed to consider the case when \( ck_A \neq ck_B \).

According to axiom A7, we have that \( \text{first}(A \land B) \) is true if and only if \( (\text{first} A) \land (\text{first} B) \) is true. However, we must note that from the assertion “\( (\text{first} A \land B) \) is true”, in general, we can not conclude that \( \text{first} A \) is true. The reason is that \( ck_A \) and \( ck_{A \land B} \) may be different. A similar note should also be made for axiom A8.

Therefore, we also have the following fixed-time rules, which are very useful in those reasoning processes involved in several formulas that are associated with different clocks. These rules are directly derived from the axioms and inference rules given above.

\[
\begin{align*}
(F \rightarrow) & \ 	ext{first} \ (\text{next}(k)(B \rightarrow A)), \text{where} \ k = \text{rank}(t, ck_B \rightarrow A) \\
& \ 	ext{first} \ (\text{next}(m)B), \text{where} \ m = \text{rank}(t, ck_B) \\
& \ 	ext{first} \ (\text{next}(n)A), \text{where} \ n = \text{rank}(t, ck_A)
\end{align*}
\]

\[
\begin{align*}
(F \land) & \ 	ext{first} \ (\text{next}(m)A), \text{where} \ m = \text{rank}(t, ck_A) \\
& \ 	ext{first} \ (\text{next}(n)B), \text{where} \ n = \text{rank}(t, ck_B) \\
& \ 	ext{first} \ (\text{next}(k)(A \land B)), \text{where} \ k = \text{rank}(t, ck_{A \land B})
\end{align*}
\]

\[
\begin{align*}
(F \lor) & \ 	ext{first} \ (\text{next}(k)(A \lor B)), \text{where} \ k = \text{rank}(t, ck_{A \lor B}) \\
& \ 	ext{first} \ (\text{next}(n)A), \text{where} \ n = \text{rank}(t, ck_A)
\end{align*}
\]