BCTL: A Branching Clock Temporal Logic

Chuchang Liu *Mehmet A. Orgun*

Department of Computing
Macquarie University
NSW 2109, Australia

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

Abstract

This paper presents a branching time temporal logic called BCTL. In this logic, branching time is represented by branching clocks, which can be specified as Chronolog programs. In BCTL, formulas are allowed to be defined on different branching clocks. Apart from the temporal operators `first` *and* `next`*, BCTL contains a next-bounded symbol ! and four modalities:* $\forall\Box$, $\forall\Diamond$, $\exists\Box$ *and* $\exists\Diamond$. *This logic can be used to describe nondeterministic programs and concurrent systems.*

1 Introduction

Chronolog [10] is an extension of logic programming based on a linear-time temporal logic [4]. In [7], intending to extend the Chronolog temporal logic, we proposed that in a temporal logic formulas could be allowed to be defined on different clocks. That is, every predicate can be assigned a local clock, so that each formula can also be clocked. Therefore, reasoning about the behavior of a system depends on the clock assignment given. This logic, called Temporal Logic with Clocks (TLC), can be used for modelling those systems where granularity of time is needed.

In TLC, the global clock is defined as the increasing sequence of all natural numbers, i.e., $< 0, 1, 2, \ldots >$, and local clocks are subsequences of the global clock. In other words, it is not required that every granularity is a refinement of another one. Furthermore, in Chronolog(MC) [7], a programming language based on TLC, 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.

Note that both Chronolog logic and its extension TLC are linear time temporal logics. Usually, linear time temporal logics assume that there is a unique future at any specific time; while branching time temporal logics [2, 3, 8] assume that there may be several possible futures at any moment in time.

In order to guarantee that the definition of a (local) clock in TLC really specifies an "actual" linear clock, we proposed three clock constraints [7] to ensure the linearity and monotonicity of given clocks. For any moment in time on a given linear clock, there is a unique "next" moment as its immediate successor. When the clock constraints are relaxed, branching time can be obtained. Thus, given a formula, executing the clock definition involves finding a branch of time in which we can prove that the formula is true.

In this paper, we extend TLC by allowing logical formulas to be defined on branching clocks, on which there may be some moments in time that are followed by more than one *next moment.* The extension is called branching clock temporal logic (BCTL). Apart from the temporal operators `first` and `next`, BCTL contains a *next-bounded* symbol ! and four modalities: $\forall\Box$ ("for all branches always"), $\forall\Diamond$ ("for all branches sometime"), $\exists\Box$ ("for some branches always")and $\exists\Diamond$ ("for some branches sometime"), so that it has a more powerful expressive ability. In particular, it is suitable for describing those systems which contain a number of processes running concurrently, such as concurrent systems and nondeterministic programs.

The structure of the rest of the paper is as follows. Section 2 discusses branching clocks, including the definition and specifications of clocks. In section 3, the syntax and semantics of BCTL are given. In section 4, the axioms and inference rules of BCTL are given. Section 5 discusses the properties of BCTL. Section 6 gives a simple example, which shows how to describe some properties of systems based on branching clocks. Last section concludes the paper with a brief discussion on related work.

0-8186-7937-9/97 $10.00

2 Branching Clocks

We use the set ω of natural numbers to model the collection of moments in time. In TLC [7], linear clocks can be viewed as sequences over ω. Given a linear clock $ck =< t_0, t_1, t_2, \ldots >$, we call n the rank of t_n on ck, written as $rank(t_n, ck) = n$. Inversely, we write $t_n = ck^{(n)}$, which means that t_n is the moment in time on ck whose rank is n.

Informally, any given clock ck can be represented as a meta-Chronolog program shown in Figure 1.

```
first ck(ck^(0)).

next ck(N) <- ck(M), K=rank(M,ck),
              N is ck^(K+1).
```

Figure 1: A Meta-specification of a clock

Here we have two temporal operators, `first` and `next`, which refer to the initial and the next moment in time respectively. The next moment is defined relative to a given moment in time, whereas the initial moment in time always refers to time 0. To guarantee that such a specification really represents an actual clock, the following clock constraints are proposed [7]: With respect to the specification of a clock, we may have

1. For any successful query `first next(`m`) ck(X)`, $m \leq$ `X`.
2. For any pair of successful queries `first next(`m`) ck(X)` and `first next(`n`) ck(Y)`, if $m < n$, then `X` $<$ `Y`.
3. For any pair of successful queries `first next(`m`) ck(X)` and `first next(`m`) ck(Y)`, `X` $=$ `Y`.

The notation `next(`n`)` denotes n applications of `next`. The intuitive meaning of these constraints are as follows. The first one says that the rank of a moment on the clock is not greater than the moment. The main motivation for this constraint is computational. The second one says that the clock can only tick forwards, that is, `ck` defined by the representation is monotonic. The third one says that the clock is single-valued at each moment. These constraints ensure that a clock specified by an appropriate Chronolog program is linear.

When the clock constraints are relaxed, we have branching time. That is, there may be a number of *next moments* as the immediate successors for a given moment in time on a clock. Such clocks are called branching clocks. For example, consider the specification of a clock ck shown in Figure 2.

```
first ck(0).
next ck(N) <- ck(M), X in {1,2,3,4},
              N is M+X.
```

Figure 2: Specification of a branching clock

It satisfies the first clock constraint, however, it does not satisfy the second and the third ones. For example, the next moment of time 0 may be 1, 2, 3 or 4. The specification actually defines a branching clock because each moment has four successors.

Formally, we have the definition:

Definition 1 *A branching clock ck is a tree which consists of a (finite or infinite) number of time nodes, or simply nodes, and satisfies the following conditions:*

- *ck has a special node called root which is not a successor of any node, and each node has 0 or more nodes as its immediate successors (called next nodes of the node).*
- *each node is assigned a natural number, called the current time of the node.*
- *The current time of any node is strictly less than the current time of any of its next nodes.*

When there is no confusion, we do not distinguish between a node and the time with which the node is associated. That is, when we talk about the node t, t may stand for the node itself or for the current time of the node.

Definition 2 *Let ck be a branching clock. We call* $< t, t_1, t_2, \ldots >$ *a branch of ck starting from t, or simply a branch from t, if* t_1 *is the current time of a next node of t and, in general, for any* t_k $(k = 1, 2, \ldots)$, t_{k+1} *is the current time of a next node of* t_k *when* t_{k+1} *exists.*

Given a branching clock, let t be a time node. We define:

$[t]$: the set of all next nodes of t,
$\mathbf{B}[t]$: the set of all branches from t.

According to the definition of branching clocks, linear clocks are actually a particular kind of branching clocks. If a clock has only one branch from the root, then it is linear.

Let ck_1 and ck_2 be any two branching clocks. We now present a constructive method to generate a new branching clock ck from ck_1 and ck_2, which we call the product of ck_1 and ck_2.

In the following, for convenience, we write $[t]_T$ to denote the set of all next nodes of t on a given clock T, i.e., a tree T, and $\mathbf{B}[t]_T$ the set of all branches from t on T.

1. If $min\{t|t \in ck_1 \text{ and } t \in ck_2\}$ exists, let $t_0 = min\{t|t \in ck_1 \text{and } t \in ck_2\}$ be the root of ck; otherwise, ck is an empty clock.

2. If there is only one node identified by t_0 on ck_1, then the tree consisting of all branches in $\mathbf{B}[t_0]_{ck_1}$ and with the root t_0, denoted by $\mathrm{T}_1(t_0)$, is called an accompanying tree of t_0 on ck_1. If there are more than one node identified by t_0 in ck_1, merge all $\mathbf{B}[t_0]_{ck_1}$ into a tree with the root t_0 as the accompanying tree $\mathrm{T}_1(t_0)$. In the same way, obtain an accompanying tree $\mathrm{T}_2(t_0)$ of t_0 on ck_2.

3. For any node t on ck which has been generated, based on its accompanying trees $\mathrm{T}_1(t)$ and $\mathrm{T}_2(t)$, first find $[t]_{ck}$, and then find the accompanying trees $\mathrm{T}_1(t_i)$ and $\mathrm{T}_2(t_i)$ for each $t_i \in [t]_{ck}$.

The general idea for doing 3 is: for all $t_i \in [t]_{T_1}$, check if $t_i \in [t]_{T_2}$, if so, then $t_i \in [t]_{ck}$; If there is only one node identified by t_i on T_1, then the tree consisting of all branches in $\mathbf{B}[t_i]_{T_1}$ and with the root t_i is $\mathrm{T}_1(t_i)$, otherwise merge all $\mathbf{B}[t_i]_{ck_1}$ into a tree with the root t_i as $\mathrm{T}_1(t_i)$; in the same way obtain $\mathrm{T}_2(t_i)$. If the fact that $t_i \in [t]_{T_2}$ is not true, we may check $[t']_{T_2}$ for some $t' \in [t]_{T_2}$, and so on.

In the above procedure, for any $t \in ck$, the root of its accompanying tree $\mathrm{T}_i(t)$ $(i = 1, 2)$ may be obtained by one node or by merging more nodes identified by t on ck_i. Such a node identified by t on ck_i is called a corresponding node to t on ck. Note that there may be other nodes which are also identified by t but do not make any contribution for the root of the accompanying tree, therefore, they are not regarded as nodes corresponding to t on ck.

In the procedure of generating the branching clock ck, t_0 is uniquely determined by ck_1 and ck_2 and, by induction, it can be shown that, for any t on ck generated by the method, $[t]$ is also uniquely determined by its accompanying trees $\mathrm{T}_1(t)$ and $\mathrm{T}_2(t)$, i.e., by ck_1 and ck_2, ck must be therefore uniquely determined by ck_1 and ck_2. That is, we have actually defined an operation over the set of branching clocks: for any two branching clocks, there exists a unique branching clock, which is constructed by the method. The resulting branching

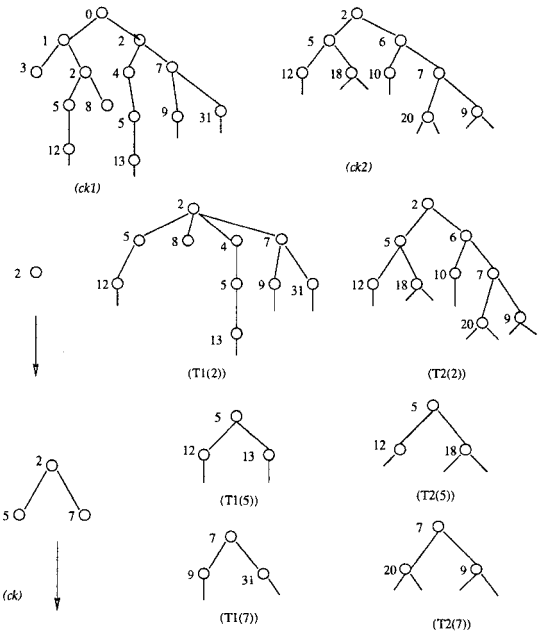

Figure 3: An Example of Generating Clocks

clock ck is the product of ck_1 and ck_2, denoted as $ck = ck_1 \sqcap ck_2$.

Because of space limitations, we do not give details for the method and the proof of the uniqueness of the generated clocks. An example for constructing a branching clock from given clocks is shown in figure 3.

3 Syntax and Semantics of BCTL

3.1 Formulas

In the vocabulary of BCTL, apart from variables, constants, function symbols, predicate symbols, we also have two propositional connectives: $\neg$ and $\wedge$, one quantifier: $\forall$, two temporal operators: `first` and `next`, one *next-bounded* symbol: `!` and punctuation symbols: (,). In BCTL, the definition of terms is as usual. We now give the definition of (well-formed) formulas.

Definition 3 *A formula is defined inductively as follows:*

- *If p is an n-ary predicate symbol and $e_0, e_1, \ldots, e_{n-1}$ are terms, then $p(e_0, e_1, \ldots, e_{n-1})$ is a formula (called an atomic formula, or an atom).*
- *If A and B are formulas, then so are $\neg A$, $A \wedge B$,* `first` A , `next` A *and* `!next` A .
- *If A is a formula and x is a variable, then $(\forall x)A$ is a formula.*

We may also use the notation `next(`n`)` to denote n applications of `next`. In particular, we may write `next(0)` and `next(1)`, which denote 0 and 1 application of `next` respectively. Similarly, `!next` (n) denotes n applications of `!next`.

The connectives $\vee$, $\rightarrow$, $\leftrightarrow$ and the quantifier $\exists$ can be derived from the prime connectives and quantifier as usual. The derived basic modalities are recursively defined as follows:

$$\forall\Box A \stackrel{\text{def}}{=} (\texttt{first}\ A) \wedge (\forall\Box \texttt{next} A)$$
$$\forall\Diamond A \stackrel{\text{def}}{=} (\texttt{first}\ A) \vee (\forall\Diamond \texttt{next} A)$$
$$\exists\Box A \stackrel{\text{def}}{=} (\texttt{first}\ A) \wedge (\exists\Box \texttt{!next} A)$$
$$\exists\Diamond A \stackrel{\text{def}}{=} (\texttt{first}\ A) \vee (\exists\Diamond \texttt{!next} A)$$

Note that each of the modalities $\forall\Box$, $\forall\Diamond$, $\exists\Box$ and $\exists\Diamond$ is an operator as a whole. Their formal meanings will be given in next section.

3.2 Clock assignment

We now give the definition of a clock assignment, which assigns local branching clocks for all predicate symbols.

Definition 4 *A clock assignment ck of BCTL is a map from the set $\mathcal{SP}$ of predicate symbols to the set $\mathcal{CK}$ of (branching) clocks, i.e. $ck \in [\mathcal{SP} \rightarrow \mathcal{CK}]$. The notation $ck(p)$ denotes the clock which is associated with a predicate symbol p on a given clock assignment ck.*

We now extend the notion of a clock assignment to formulas. Let A and B be formulas, $p(x_1, x_2, \ldots, x_n)$ an atomic formula, and ck a clock assignment. We define a clock assignment ck on formulas of $BCTL$ as follows:

$$ck(p(x_1, x_2, \ldots, x_n)) = ck(p)$$
$$ck(\neg A) = ck(A)$$
$$ck(A \wedge B) = ck(A) \sqcap ck(B)$$
$$ck(\texttt{first}\ A) = ck(A)$$
$$ck(\texttt{next}\ A) = ck(A)$$
$$ck(\texttt{!next}\ A) = ck(A)$$
$$ck((\forall x)A) = ck(A)$$

The only operation which requires the generation of new clocks is conjunction. We propose to use the product operation discussed above to generate the clock of $A \wedge B$. Thus, every formula of BCTL can be clocked. In general, let A be a formula of BCTL, $\mathcal{SP}_A$ the set of predicate symbols occurring in A, and ck a clock assignment, then we have that $ck(A) = \sqcap_{p \in \mathcal{SP}_A} ck(p)$.

In the following, we always use ck to represent a clock assignment and simply call it a clock, and we also use the notation $ck(A), ck(B), \ldots, ck_1, ck_2, \ldots$ to represent local clocks.

3.3 Semantics

In the following, we provide the semantics of the elements of BCTL in terms of clocks and interpretations. We assume that the meanings of variables and function symbols are "rigid", that is, independent of time nodes on a given branching clock.

We now define interpretations of $BCTL$ as follows:

Definition 5 *A temporal interpretation I on a given clock ck of $BCTL$ comprises a non-empty set $\mathbf{D}$, called the domain of the interpretation, over which the variables range, together with for each variable, an element of $\mathbf{D}$; for each n-ary function symbol, an element of $[\mathbf{D}^n \rightarrow \mathbf{D}]$; and for each n-ary predicate symbol p, an element of $[ck(p) \rightarrow P(\mathbf{D}^n)]$.*

We write $\models_{I,ck,t} A$ to denote the fact that a formula A is true at moment t (or, more strictly, the time node t, $t \in ck(A)$) on the clock ck under a given interpretation I. We define $\models_{I,ck,t}$ inductively as follows:

(1) If $f(e_0, \ldots, e_{n-1})$ is a term, then $I(f(e_0, \ldots, e_{n-1})) = I(f)(I(e_0), \ldots, I(e_{n-1}))$.

(2) For any n-ary predicate symbol p and terms $e_0, \ldots, e_{n-1}$ and any $t \in ck(p)$, $\models_{I,ck,t} p(e_0, \ldots, e_{n-1})$ if and only if $< I(e_0), \ldots, I(e_{n-1}) > \in I(p)(t)$.

(3) For any $t \in ck(A)$, $\models_{I,ck,t} \neg A$ if and only if it not the case that $\models_{I,ck,t} A$.

(4) For any $t \in ck(A) \sqcap ck(B)$, $\models_{I,ck,t} (A \wedge B)$ if and only if $\models_{I,ck,t} A$ for all $t \in ck(A)$ corresponding to the node t on $ck(A \wedge B)$ and $\models_{I,ck,t} B$ for all $t \in ck(B)$ corresponding to the node t on $ck(A \wedge B)$.

(5) For any $t \in ck(A)$, $\models_{I,ck,t} (\forall x)A$ if and only if $\models_{I[d/x],ck,t} A$ for all $d \in \mathbf{D}$ where the interpretation $I[d/x]$ is just like I except that the varible x is assigned the value d in $I[d/x]$.

(6) For any $t \in ck(A)$, $\models_{I,ck,t}$ `first`A if and only if $\models_{I,ck,t_0} A$, where t_0 is the root of $ck(A)$.

(7) For any $t \in ck(A)$, $\models_{I,ck,t}$ `next`A if and only if $\models_{I,ck,s} A$ for all $s \in [t]$.

(8) For any $t \in ck(A)$, $\models_{I,ck,t}$ `!next`A if and only if there exists at least one time node $s \in [t]$ such that $\models_{I,ck,s} A$.

We now give the following notations:

$\models_{I,ck_i} A$:	denotes that A is true on a branch ck_i of $ck(A)$ under a given interpretation I, which means that A is true at all nodes on ck_i under I.
$\models_{I,ck} A$:	denotes that A is true on the clock ck under a given interpretation I, which means that A is true at all time nodes on $ck(A)$ under I.
$\models_{ck} A$:	denotes that A is true on the clock ck under any interpretation.
$\models A$:	denotes that A is true on any clock under any interpretation.

4 The proof system for BCTL

We use the notation $\vdash A$ to denote that A is a theorem of BCTL and $\vdash_{ck} A$ to denote that A is a theorem of BCTL which holds on ck. Then the notion of deducibility can be characterized in terms of theoremhood: $\Gamma \vdash A$ means that the formula A is deducible from the set Γ of formulas using axioms and inference rules in BCTL. The explanation for $\Gamma \vdash_{ck} A$ can be made in a similar fashion.

The proof system for BCTL is as follows.

- **Axioms:**

(A1) All substitution rules of first-order calculus.

(A2) $\texttt{first first}\ A \leftrightarrow \texttt{first}\ A$.

(A3) $\texttt{next first}\ A \leftrightarrow \texttt{first}\ A$

(A4) $\texttt{!next first}\ A \leftrightarrow \texttt{first}\ A$

(A5) $\texttt{first}(\neg A) \leftrightarrow \neg(\texttt{first}\ A)$.

(A6) $\texttt{next}(\neg A) \leftrightarrow \neg(\texttt{!next}\ A)$.

(A7) $\texttt{!next}(\neg A) \leftrightarrow \neg\ \texttt{next}\ A$.

(A8) $\texttt{first}\ (\forall x)(A) \leftrightarrow (\forall x)(\texttt{first}\ A)$.

(A9) $\texttt{next}\ (\forall x)(A) \leftrightarrow (\forall x)(\texttt{next}\ A)$.

(A10) $\texttt{!next}\ (\forall x)(A) \leftrightarrow (\forall x)(\texttt{!next}\ A)$.

(A11) $\texttt{first}(A \wedge B) \leftrightarrow (\texttt{first}\ A) \wedge (\texttt{first}\ B)$.

(A12) $\texttt{next}(A \wedge B) \leftrightarrow (\texttt{next}\ A) \wedge (\texttt{next}\ B)$.

(A13) $\texttt{!next}(A \wedge B) \rightarrow (\texttt{!next}\ A) \wedge (\texttt{!next}\ B)$.

(A14) $\texttt{next}\ A \rightarrow \texttt{!next}\ A$.

Note that temporal operators `!next` and `next` do not satisfy the commutativity rule. In general, the formula $\texttt{!next next}\ A \leftrightarrow \texttt{next !next}\ A$ is not true.

- **Inference rules:**

(R1) $B \rightarrow A$, $B \vdash A$. (Modus Ponens)

(R2) $A \vdash_{ck} \texttt{first}\ A$, when $ck(A)$ is non-empty.

(R3) $A \vdash_{ck} \texttt{next}\ A$, when there is always a next node on $ck(A)$, i.e., when there are no finite branches.

As can be seen from axioms A6 and A7, `next` and `!next` are dual.

The soundness of the system says that the proof system for BCTL is valid with respect to the given semantics scheme. Its proof is straightforward. We omit the details. Note that we do not discuss the issue of completeness of BCTL in this paper.

5 Properties of BCTL

In this section, we discuss the properties of temporal logic BCTL. We first prove the following lemma:

Lemma 1 *Let A be a formula and t any moment (strictly, any time node) on the branching clock $ck(A)$. Then we have that $\forall\Box A$ is true at t if and only if $\models_{ck_i} A$ for all branches $ck_i \in \mathrm{B}[t_0]$, where t_0 is the root of $ck(A)$.*

Proof. Let I be any interpretation. Without loss of generality, we assume that $ck_i =< t_0, t_1, t_2, \ldots >$. We now want to show that, if $\forall\Box A$ is true at t, then $\models_{I,ck_i,t_n} A$, for any t_n on ck_i. Actually, by induction on n and the definition of $\forall\Box$, from the fact that $\forall\Box A$ is true at t, it is easy to show that for any natural number n $\texttt{first next}(n)A$ is true at t. Thus, according to the semantics definition, $\texttt{first next}(n)A$ is true at t iff $\texttt{next}(n)A$ is true at t_0 iff $\texttt{next}(n-1)A$ is true at r, for all $r \in [t_0]$. Therefore, in particular, $\texttt{next}(n-1)A$ is true at t_1. Continuing the procedure, from the fact that $\texttt{first next}(n)A$ is true at t, we must obtain that A is true at t_n. Therefore, considering the arbitrariness of t_n, we have that $\models_{I,ck_i} A$. Furthermore, since I can be any given interpretation, we have that $\models_{ck_i} A$.

The proof of the sufficiency is trivial, therefore it is omitted. ∎

This lemma says that $\forall\Box A$ is true at any moment (time node) in time on a given branching clock if and only if A is always true at all branches on the clock. Therefore, we naturally read $\forall\Box$ as "for all branches always". In other words, from the view of an investigator, the fact "$\forall\Box A$ is true" means that he can find: A is true everywhere, which does not depend on when he investigates it or where he stands.

Similarly, we have following lemmas regarding about the modalities $\forall\Diamond$, $\exists\Box$ and $\exists\Diamond$. Because of space limitations, we omit their proofs.

Lemma 2 *Let A be a formula and t be any moment on the branching clock $ck(A)$. Then we have that $\models_{ck,t} \forall\Diamond A$ if and only if for all branch $ck_i \in \mathbf{B}[t_0]$, there exists $r \in ck_i$ so that $\models_{ck_i,r} A$, where t_0 is the root of $ck(A)$.*

Lemma 3 *Let A be a formula and t be any moment on the branching clock $ck(A)$. Then we have that $\models_{ck,t} \exists\Box A$ if and only if there exists a branch $ck_i \in \mathbf{B}[t_0]$ so that $\models_{ck_i} A$, where t_0 is the root of $ck(A)$.*

Lemma 4 *Let A be a formula and t be any moment on the branching clock $ck(A)$. Then we have that $\models_{ck,t} \exists\Diamond A$ if and only if there exists a branch $ck_i \in \mathbf{B}[t_0]$ so that $\models_{ck_i,r} A$ for some $r \in ck_i$, where t_0 is the root of $ck(A)$.*

Lemma 2 says that $\forall\Diamond A$ is true at any moment (time node) in time on a given branching clock if and only if A is true at sometimes on all branches of the clock. Therefore, we read $\forall\Diamond$ as "for all branches sometime". Lemma 3 says that $\exists\Box A$ is true at any moment (time node) in time on a given branching clock if and only if A is true always on at least a branch of the clock. Therefore, we read $\exists\Box$ as "for some branch always". Lemma 4 says that $\exists\Diamond A$ is true at any moment (time node) in time on a given branching clock if and only if A is true at sometime on at least a branch of the clock. Therefore, we read $\exists\Diamond$ as "for some branch sometime".

The properties of computational systems can be expressed by using BCTL formulas. For example, a safety property is expressible as an invariance assertion of the form $\forall\Box A$.

6 An Example

BCTL can be used to specify properties of concurrent systems and nondeterministic programs. BCTL allows formulas to be defined on different clocks. Although different processes may have different local branching clocks, the predicates which describe a local behavior of the system may be defined on the same local branching clock. Thus, a property describing the local behavior may be expressed as a formula on the local branching clock, and it may be proved by involving the clock only.

We now give a simple example. Consider RS232 software repeater problem, which is first proposed by

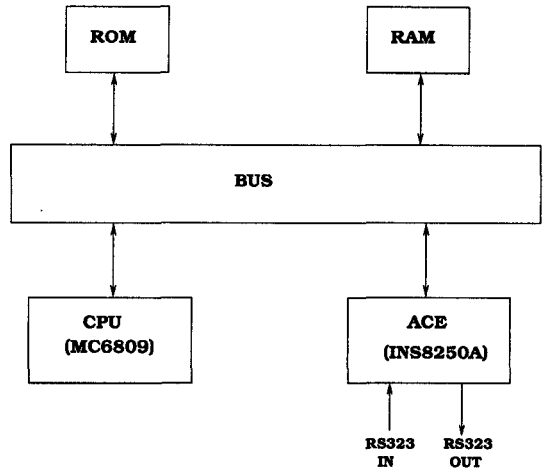

Figure 4: Software Repeater Environment

Landwehr [6]. The problem may be described as follows. An RS232 software repeater runs on a CPU interfaced to RS232 communications chip which comprises a receiver and a transmitter. The receiver processes an incoming bit stream and assembles the data into bytes for handling by the software repeater. The transmitter accepts bytes from the repeater and generates a serial bit stream. The software repeater is required to copy incoming data from the receiver to the transmitter, with buffering as required, so that over an appropriate range of receiver and transmitter baud rates, all incoming data are correctly transmitted. The problem specification requires the system to halt at the first occurrence of an error. Figure 4 is a block diagram of the repeater environment.

To simplify RS232 repeater problem, we omit system initialization and buffer management, as proposed in [5]. Also, we assume that no errors occur and the problem will be simplified to access to status information. We rationalize that by assuming an idealized ACE, inspired by INS8250A. It has three separately addressable locations from which the CPU can access data:

- *inport* is the location where received bytes appear;
- *dataready* initialized by 0 (by the system, not the software) is set to 1 by ACE whenever a received byte appears at *inport* and is cleared to 0 as a side effect of reading *inport*;
- *outport* is the location at which the CPU writes data in order to transmit it.

The following is a proposed solution code, which is written in an assembly language:

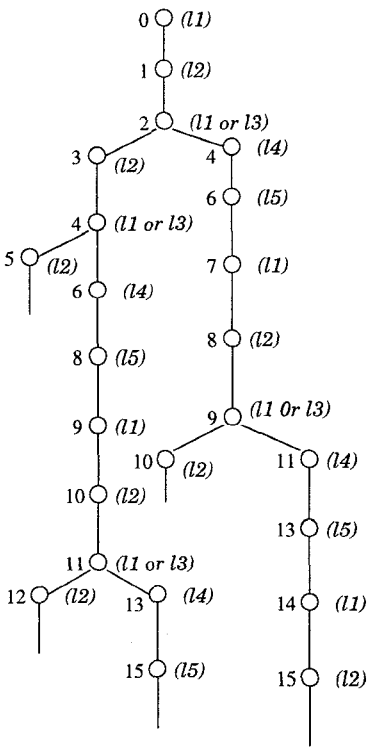

Figure 5: A Branching Clock

l_1:	check	dataready
l_2:	beq	l_1
l_3:	lda	inport
l_4:	sta	outport
l_5:	jump	l_1

We assume that the time for executing each instruction, i.e. the time from fetching an instruction to completing the execution of the instruction is as follows:

l_1:	1 unit time	l_2:	1 unit time
l_3:	2 unit time	l_4:	2 unit time
l_5:	1 unit time		

Therefore, we may construct a branching clock as shown in figure 5. In the figure, the label (or labels) made in each node denote the instruction which is currently fetched to be executed at the current time of the node.

We define the following predicates:

`inseq(X)`:	`X` is inport sequence so far.
`outseq(X)`:	`X` is outport sequence so far
`X` $\ll$ `Y`:	Sequence `X` is a prefix of sequence `Y`.
`copy(X,Y)`:	`Y` is a copy of `X`.

All the predicates, according to our problem, can be defined on the same branching clock shown in figure 5.

Note that the predicate `copy` can be directly defined by using `inseq` and `outseq` as follows:

$$\texttt{copy(X,Y)} \stackrel{\text{def}}{=} \texttt{X} \ll \texttt{Y and Y} \ll \texttt{X}.$$

Then the program should satisfy the following properties:

$\forall\Box((\forall X)(\forall Y)(\texttt{inseq(X)} \wedge \texttt{outseq(Y)} \rightarrow \texttt{Y} \ll \texttt{X}))$.
$\forall\Box((\forall X)(\forall Y)(\texttt{inseq(X)} \wedge \texttt{outseq(Y)} \wedge \neg\texttt{copy(X,Y)} \rightarrow$
$\exists\Diamond((\exists Z)(\texttt{next outseq(Z)} \wedge \texttt{copy(X,Z)})))$.

The first property says that for all branches (at any moment in time) we always have the fact that if `X` is the inport sequence and `Y` outport sequence up to now then `Y` is a prefix of `X`. The second one says: it is always true for all moments on all branches that, if at the current time the outsequence `Y` is not a copy of the inport sequence `X`, then there exists a branch, at some moment in time on this branch, so that the outsequence `Z` at that moment is a copy of `X`.

The proofs of the properties of a system depend on the formal specification of the system, which can be written as a set of BCTL formulas. Thus, those formulas representing properties can be directly derived from the set by using axioms and inference rules of BCTL. For some cases we may need to use induction.

7 Conclusions

In this paper, we have presented the temporal logic BCTL, which is an extension of first order logic with branching clocks. In this logic, branching time is represented by branching clocks, which can be specified as Chronolog programs. Due to introduction of the next-bounded symbol ! and the modalities, the logic is more expressive.

When a given clock is linear, we actually obtain a linear time temporal logic. At this time, `!next` is not needed. We also do not need the modalities $\exists\Box$ and $\exists\Diamond$, and $\forall\Box$ and $\forall\Diamond$ can be directly replaced by $\Box$ and $\Diamond$, respectively. In the case when the clock is linear, we can describe the properties of programs or systems in the same way. For example, a safety property of a program is expressible as an invariant assertion $\Box A$, which expresses A is true at all moments in time on the local linear clock of the program.

Let us consider a Consumer-Producer program [9], which consists of two processes: p_1 – Producer and p_2 – Consumer. We assume that L and M are the subsets of the instructions of p_1 and p_2, respectively, and a critical condition for the program is that executing instructions in L by p_1 and executing instructions in M by p_2 can

not happen simultaneously, i.e., it is never the case that p_2 simultaneously executes the instructions in M when p_1 is executing an instruction in L or vice versa. Assume that the predicate $\texttt{execute}(P, X)$ denotes that process P is currently executing the instruction X, then the mutual exclusion can be specified by the formula

$$\Box(\texttt{execute}(p_1, X) \wedge \texttt{execute}(p_2, Y) \rightarrow \neg((X \in L) \wedge (Y \in M)))$$

A liveness property can be specified as an assertion $\Diamond A$, which says that A is true at some moments in time on the clock. For example, to describe the fact that a data item D is currently sent the buffer by process p_1, then we assert that at some moment from next moment in time the data must be received by process p_2. We may use the formula

$$\texttt{send}(p_1, D) \rightarrow \Diamond \texttt{ next receive}(p_2, D).$$

As it is expected, linear time logic with or without multiple clocks can be treated as a special case of the branching clock temporal logic BCTL.

Now let us mention a number of related works. The early ideas about branching time logics can be found from Abrahamson [1]. Later, Ben-Ari et al [8] proposed the unified branching time system (UB). Then Clarke and Emerson defined computational tree logic (CTL) [2]. And, in 1983, Emerson and Halpern gave the definition of CTL* [3]. These languages are all suitable for describing concurrent systems. In particular, CTL* is a very powerful temporal logic which can be used for specifying a variety of program properties due to its modal operators. For example, in CTL*, the formula $F = (\forall\Box A) \vee (\forall\Box\neg A)$ can be true when A is true at all time nodes on a given branching clock or A is false at all nodes on the given branching clock. Also, the formula $G = (\forall\Diamond A) \wedge (\forall\Diamond\neg A)$ is false when A is true at all nodes on the given branching clock or A is false at all nodes on the given branching clock. All these formulas can also be expressed in BCTL. However, these formulas are not in the linear temporal logic LTL [11] and no LTL formulas are equivalent to them. Obviously, all the above formulas can be expressed in BCTL. We can show that all the properties expressible in CTL* are expressible in BCTL. Therefore BCTL also has a powerful expressive ability.

Future work includes completing theoretical study of BCTL, and extending Chronolog(MC) with multiple branching clocks.

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

[1] K. Abrahamson. Modal logics for concurrent program. *Lecture Notes in Computer Science*, 70, 1979.

[2] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, 1981.

[3] E. A. Emerson and J. Y. Halpern. "somtimes" and "not never" revisited: on brabching versus linear time temporal logic. *Journal of the Association for Computing Machinery*, 1(33):151–178, 1986.

[4] R. Goldblatt. *Logics of Time and Computation.* CSLI – Center for the Study of Language and Information, Stanford University, 1987. Lecture Notes no:7.

[5] P. Kearney, J. Staples, A. Abbas, and C. Liu. Functional verification of real-time code: A simplified RS232 softwere repeater problem. *High Integrity Systems*, 1(4):359–373, 1995.

[6] Carl Landwehr. The RS-232 software repeater problem. In *CIPHER, the Newsletter of IEEE Technical Committee on Secrity and Privacy, Summer*, pages 34–35, 1989.

[7] C. Liu and M. A. Orgun. Dealing with multiple granularity of time in temporal logic programming. Technical Report TR–95–04, Accepted to *Journal of Symbolic Computation*, Department of Computing, Macquarie University, Sydney, NSW 2109, Australia, October 1995.

[8] Z. Manna M. Ben-Ari and A. Pnueli. The temporla logic of branching time. In *Proceedings of the 8th Annual ACM Symposium on Principles of Programming languages*, pages 164–176, 1981.

[9] Z. Manna and A. Pnueli. Verification of concurrent programs: The temporal framework. In R. S. Boyer and J. S. Moore, editors, *The Correctness Problem in Computer Science.* Academic Press, 1981.

[10] M. A. Orgun and W. W. Wadge. Theory and practice of temporal logic programming. In L. Fariñas del Cerro and M. Penttonen, editors, *Intensional Logics for Programming*, pages 23–50. Oxford University Press, 1992.

[11] A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.