K. M. Passino and P. J. Antsaklis, "Branching Time Temporal Logic for Discrete Event System Analysis,"Proc. of the 26th Allerton Conference on Communication, Control and Computing, pp. 1160-1169, Monticelo, IL Sept. 28-30, 1988.

Presented at the Twenty-Sixth Annual
Allerton Conference on Communication,
Control, and Computing
University of Illinois at Urbana-Champaign
28-30 September 1988

pp 1160 – 1169

BRANCHING TIME TEMPORAL LOGIC FOR DISCRETE EVENT SYSTEM ANALYSIS

K.M. PASSINO and P.J. ANTSAKLIS
Dept. of Electrical and Computer Engineering
University of Notre Dame, Notre Dame IN 46556

Abstract

Given a linear-time temporal logic description of discrete event system behavior, it has been shown that a proof system can be used to prove that certain properties stated in the linear-time temporal language are satisfied. Although such proofs offer an analysis method for discrete event systems, they are often quite difficult to construct, requiring special insights and clever tricks. As an alternative, design specifications can be stated in a branching-time temporal language and a verification algorithm, which replaces the hand-constructed proofs, can be used to test whether the specifications are true or not. In this paper, a certain Petri net is used to model finite state discrete event systems. It is shown that the Petri net model can be used to generate the structure over which the verification algorithm operates. Hence, the algorithm can be used to verify discrete event system properties that can be stated in a branching-time temporal language. The approach to analysis presented in this paper is illustrated by giving three new examples where the Petri nets are specified and the algorithm is used to verify a variety of design specifications.

1.0 Introduction

This paper contains some initial results on the use of a branching-time temporal logic called Computation Tree Logic (CTL) [4,3] for discrete event system analysis in the field of control theory. CTL formulas are used here to state the desired properties or the design specifications that we would like the system to satisfy. A certain Petri net is used to model the underlying discrete event system and to generate a CTL structure which is used by a design specification verification algorithm. Three new examples are given to illustrate the approach to discrete event system analysis. The first example involves a surge tank where the design specifications describe how a liquid level should be regulated. A manufacturing system is used for the second example and a mutual exclusion and priority specification are verified. The third example involves the study of a "self stabilizing distributed system". It is shown that a certain system which was originally thought to be self stabilizing will actually deadlock.

An discrete event system from chemical process control is the "surge tank" [9]. It will be used below as an illustrative example in the explanation of the analysis process for discrete event system studied here. The surge tank has a sensor which can distinguish between five liquid levels (empty, low, normal, high, and full) and a controller which simply opens and closes a fill valve. Unpredictable users operate an empty valve to take liquid from the tank. One design specification for the surge tank system is that the tank never become empty or full if it is initially at a normal liquid level.

First, the discrete event system is modelled with a certain Petri net. The Petri net model defined here allows for control inputs, outputs, and disturbances. There are both control inputs that can cause an immediate change in the plant, and inputs that act to enable and disable certain events. Although the modelling methodology used here is quite flexible, it is limited by a finite state restriction. The Petri net model for the surge tank has one input, the fill valve, one output, the liquid level, and one disturbance, the empty valve. Once the discrete event system has been modelled one must provide the design specifications, i.e., how one would like the system to behave. In this paper the design specifications are stated with CTL formulas. CTL formulas are comprised of (i) atomic propositions such as "the liquid level is low" or "the fill valve is open", (ii) the Boolean connectives (e.g., $\wedge$ and, $\vee$ or, $\neg$ not), and the temporal operators (iii) $\forall s \odot (f_1)$ $(\exists s \odot (f_1))$, intuitively meaning for all (for some) *next* structure state(s) the formula f_1 is true, and (iv) $\forall p[f_1 U f_2]$ $(\exists p[f_1 U f_2])$, intuitively meaning for all (for some) path(s) f_1 is true *until* f_2 becomes true. Suppose that the initial liquid level of the surge tank is normal and the fill valve is open. An example design specification for the surge tank is $\neg\exists s[\text{True}U(\text{empty}\vee\text{full})]$. This intuitively means "it is not the case that there exists a path such that the liquid level will become empty or full". The Petri net model of the discrete event system is used to generate a CTL structure. The design specification verification algorithm from [4] is used here to determine whether a CTL formula is true or not for the CTL structure.

Hence, discrete event system analysis here involves (i) modelling the discrete event system with a certain Petri net, (ii) using the Petri net to generate a CTL structure, (iii) stating the design specifications in the branching time temporal language, and (iv) using a verification algorithm to mechanically test whether the design specifications are true or not. Next, the relationships to relevant research in temporal logic and the theory of Petri nets are given.

Note that, given a linear-time temporal logic description of discrete event system behavior, it has been shown that a proof system can be used to prove that certain properties stated in the linear-time temporal language are satisfied [12,13,18,14]. Although such proofs offer an analysis method for discrete event

systems, they are often quite difficult to construct, requiring special insights and clever tricks. In this paper we study the use of a branching-time temporal logic called Computation Tree Logic (CTL) [4,3]. This logic system is similar to the ones found in [1,6]. Design specifications are stated in the CTL language and an algorithm which replaces the hand-constructed proofs is used to test whether the specifications are true or not. A branching-time temporal language is used here since there is a design specification verification algorithm that has complexity linear in both the size of the structure and the length of the specification. In [14] similar efficient decision procedures for simple (low length) formulas in his linear-time temporal language are developed. The verification algorithm for linear-temporal logic in general, however, must have high complexity [4]. The two types of logic must also be compared based on their expressiveness, i.e., what sort of properties can be stated in the temporal language. It is interesting to note that neither linear nor branching-time temporal logics are more expressive than the other; each can express things that the other cannot [11]. For example, linear-time temporal logic cannot express the existence of paths and branching-time cannot express certain "fairness properties". For a complete, up to date discussion of the advantages and disadvantages of linear and branching-time temporal logics see [14,6].

Petri nets appropriate for a discrete event control theoretic setting were first defined in [10]. Analysis results for discrete event systems represented with a sort of Petri net were also developed by Ichikawa and Hiraishi in [7]. The Petri net definition here follows and extends those given in [17,15,16]. It is more expressive than those in [7] and [10] in that it allows for the use of an "inhibitor arc". The expressiveness of the Petri net is, however, limited here by a finite state restriction. The "state equations" described in [17] for Petri nets are used here. These equations are analogous to the state equations used in conventional control theory. Such recursive state equations for discrete event system description have also been given in [8]. The Petri net is used here to generate the CTL structure for a discrete event system. It can be used as an alternative to the "State Machine Language" (SML) described in [2,4] for producing the structure.

The branching-time temporal logic CTL is defined in the next section. The design specification language and verification algorithms are described. In Section 3, the definition of the Petri net used here is given and it is explained how to produce the CTL structure. In Section 4, three examples are described in detail and certain results of this paper are applied to these examples.

2.0 Branching Time Temporal Logic

In this section the design specification language, i.e., the language used to state discrete event system properties is given. The semantics of the language are defined relative to a structure which is generated here with the Petri net defined in the next section. The verification algorithm which operates over the structure is described. The results of this section follow those of [4] with some notational differences.

2.1 The Design Specification Language

The specification language is a propositional branching-time temporal logic called "Computation Tree Logic" (CTL) reported in [4,3]. The language syntax is given first. Let N denote the set of natural numbers. Let A_p denote the underlying finite set of *atomic propositions*. The *formulas* of the CTL are defined inductively.

(i) Every atomic proposition $p \in A_p$ is a CTL formula.

(ii) If f_1 and f_2 are CTL formulas, then so are $\neg f_1$, $f_1 \wedge f_2$, $\forall s \odot (f_1)$, $\exists s \odot (f_1)$, $\forall p[f_1 U f_2]$, and $\exists p[f_1 U f_2]$.

(iii) Nothing else is a CTL formula unless it is obtained by finitely many applications of (i) and (ii) above.

The symbols $\wedge$ and $\neg$ mean "and" and "not" respectively. The parentheses "(" and ")" are used as needed for forming formulas. The *next time* operator is $\odot$. The formula $\forall s \odot (f_1)$ ($\exists s \odot (f_1)$) intuitively means that f_1 holds in every (in some) immediate successor structure state of the current structure state. The *until* operator is U. The formula $\forall p[f_1 U f_2]$ ($\exists p[f_1 U f_2]$) intuitively means that for every path of structure states (for some path of structure states), there exists an initial prefix of the path such that f_2 holds at the last structure state of the prefix and f_1 holds at all other structure states along the prefix.

The semantics of CTL formulas are defined relative to a labeled state-transition graph (or Kripke structure). A CTL structure is a triple $M=(S,R,P_c)$ where

(i) S is a nonempty finite set of *structure states*.

(ii) R is a binary relation on S ($S \times S \supseteq R$) which gives the possible transitions between states and must be total, i.e. $\forall s \in S\ \exists s' \in S$ such that $(s,s') \in R$.

(iii) $P_c : S \rightarrow 2^{A_p}$ assigns to each structure state the set of atomic propositions true in that structure state.

A *path* is an infinite sequence of structure states denoted by $<s_0, s_1, s_2, \ldots>$ rooted at s_0 such that $\forall i \in N$, $(s_i, s_{i+1}) \in R$. The standard notation for indicating truth in a structure is used: $M,s_0 \models f$ means that formula f holds at structure state s_0 in structure M. When the structure M is understood $s_0 \models f$ is used. Assume that f, f_1, and f_2 are CTL formulas. The relation $\models$ is defined inductively as follows.

(i) $s_0 \models a$ iff $a \in P_c(s_0)$.
(ii) $s_0 \models \neg f$ iff it is not the case that $s_0 \models f$.
(iii) $s_0 \models f_1 \wedge f_2$ iff $s_0 \models f_1$ and $s_0 \models f_2$.
(iv) $s_0 \models \forall s \odot (f)$ iff for all states s such that $(s_0,s) \in \mathbf{R}$, $s \models f$.
(v) $s_0 \models \exists s \odot (f)$ iff for some state s such that $(s_0,s) \in \mathbf{R}$, $s \models f$.
(vi) $s_0 \models \forall p[f_1 \mathbf{U} f_2]$ iff for all paths $p = <s_0, s_1, s_2, \ldots>$, $\exists i \in \mathbf{N}$ where $i \geq 0$ and $s_i \models f_2$, and $\forall j \in \mathbf{N}$ where $0 \leq j < i$, $s_j \models f_1$.
(vii) $s_0 \models \exists p[f_1 \mathbf{U} f_2]$ iff for some path $p = <s_0, s_1, s_2, \ldots>$, $\exists i \in \mathbf{N}$ where $i \geq 0$ and $s_i \models f_2$, and $\forall j \in \mathbf{N}$ where $0 \leq j < i$, $s_j \models f_1$.

The atomic proposition "True" is always true, i.e. for all $s \in \mathbf{S}$, True$\in P_c(s)$. The following abbreviations are used in writing CTL formulas.

(i) $f_1 \vee f_2 \equiv \neg(\neg f_1 \wedge \neg f_2)$ for logical disjunction.
(ii) $f_1 \rightarrow f_2 \equiv \neg f_1 \vee f_2$ for logical implication.
(iii) $f_1 \leftrightarrow f_2 \equiv (f_1 \rightarrow f_2) \wedge (f_2 \rightarrow f_1)$ for logical equivalence.
(iv) $\forall p \diamond (f) \equiv \forall p[\text{True} \mathbf{U} f]$ intuitively means that f holds in the future along every path p from s_0, i.e. f is *inevitable*.
(v) $\exists p \diamond (f) \equiv \exists p[\text{True} \mathbf{U} f]$ intuitively means that there is some path p from s_0 that leads to a structure state at which f holds, i.e., f *potentially* holds.
(vi) $\forall p \boxdot (f) \equiv \neg \exists p \diamond (\neg f)$ intuitively means that f holds at every structure state on every path p from s_0, i.e., f holds *globally*.
(vii) $\exists p \boxdot (f) \equiv \neg \forall p \diamond (\neg f)$ intuitively means that there is some path p on which f holds at every structure state.

This completes the definition of the design specification language. Once a design specification is stated in the branching-time temporal language it is tested whether it is true or not with a verification algorithm that is described next.

2.2 The Design Specification Verification Algorithm

In this section a brief outline of the design specification verification algorithm is presented. More details are given in [4]. The verification algorithm is an effective procedure for determining whether a CTL formula is true or not. It is a "model checker" since it determines if the structure **M** is a model for the CTL formula being tested. Assume that we wish to determine whether formula f_0 is true or not in the finite structure **M**. Roughly speaking, the verification algorithm iteratively labels the states of the structure **M** with successively longer subformulas of f_0 (and negations of subformulas of f_0) until structure states are labelled with the formula f_0 or $\neg f_0$. With this, the algorithm can determine whether the formula f_0 is true or not. A more detailed description follows.

Let $\mathbf{sub}(f_0)$ denote the nonempty finite set of *subformulas* of CTL formula f_0. Note that all formulas $f \in \mathbf{sub}(f_0)$ are also CTL formulas. The definition of subformulas proceeds by induction on f_0.

(i) If $f = f_0$, then $f \in \mathbf{sub}(f_0)$.
(ii) If $f \in \mathbf{sub}(f_0)$ and $f = \neg f_1$, $f = \forall s \odot (f_1)$, or $f = \exists s \odot (f_1)$, then $f_1 \in \mathbf{sub}(f_0)$.
(iii) If $f \in \mathbf{sub}(f_0)$ and $f = f_1 \wedge f_2$, $f = \forall p[f_1 \mathbf{U} f_2]$, or $f = \exists p[f_1 \mathbf{U} f_2]$, then $f_1, f_2 \in \mathbf{sub}(f_0)$.
(iv) Nothing else is a subformula of f_0.

The nonempty finite set of *positive subformulas* of f_0 is given by $\mathbf{sub}^+(f_0) = \{f' \mid$ for all $f \in \mathbf{sub}(f_0)$, if $f = \neg f_1$, then $f' = f_1$ else $f' = f\}$. Next, we define the *length* of a formula f, denoted by $\rho(f)$. The definition proceeds by induction on f.

(i) $\rho(f) = 1$ if $f = p$ where $p \in A_p$, the set of atomic propositions,
(ii) $\rho(f) = \rho(\neg f)$,
(iii) $\rho(\forall s \odot (f_1)) = \rho(\exists s \odot (f_1)) = \rho(f_1) + 1$,
(iv) $\rho(f_1 \wedge f_2) = \rho(\forall p[f_1 \mathbf{U} f_2]) = \rho(\exists p[f_1 \mathbf{U} f_2]) = \max\{\rho(f_1), \rho(f_2)\} + 1$.

The notion of length of a formula is similar to "rank" [9] in that it quantifies the complexity of the formula.

Let $\mathbf{label}(s)$ denote the set of formulas that the algorithm has labeled structure state $s \in \mathbf{S}$ with. Beginning with i=1, on pass k the verification algorithm labels each structure state $s \in \mathbf{S}$ with the set of formulas f in $\mathbf{sub}^+(f_0)$ (or their negations) of length $\rho(f) = k$ so that $f \in \mathbf{label}(s)$ iff $\mathbf{M}, s \models f$ and $\neg f \in \mathbf{label}(s)$ iff $\mathbf{M}, s \models \neg f$. The algorithm makes n passes where $n = \text{length}(f_0)$. Hence, at termination $f_0 \in \mathbf{label}(s)$ iff $\mathbf{M}, s \models f_0$ and $\neg f_0 \in \mathbf{label}(s)$ iff $\mathbf{M}, s \models \neg f_0$. The actual algorithm that was used here is given in [4]. It runs in time $O(\rho(f_0)(|S| + |R|)^2)$. There is another algorithm for solving this problem that runs in time $O(\rho(f_0)(|S| + |R|))$ [3].

3.0 A Petri Net Representation for Discrete Event Systems

In this section a Petri net modelling methodology is developed that is used to produce the structure M. A *multiset* (bag) is a collection of objects over some domain X, but unlike standard definitions of a set, multisets allow multiple occurrances of elements [17]. Let B be a multiset, then $\#(x,B)$ represents the number of occurrances of element x in multiset B. The set X^{∞} is the set of all multisets over a domain X. If $x,y \in N^n$, $x=[x_1,x_2, \ldots ,x_n]^t$, and $y=[y_1,y_2, \ldots ,y_n]^t$ (t indicates transpose) then the statement $x \geq y$ is true iff $x_i \geq y_i$ $1 \leq i \leq n$. Similarly for $>,<$, and $\leq$. Let Ω_n denote the set containing the column vetors of an n-dimensional identity matrix and an n-dimensional vector of zeros. Let Ø denote the null set.

The Petri Net structure is described by $P_S=(P,U,Y,T,I_D,I_N,O_D,\Gamma,\Psi)$ where:

(i) $P=\{p_1,p_2, \ldots ,p_n\}$ is a non-empty finite set of $n=|P|$ (*state*) *places* represented graphically with circles (◯).

(ii) $U=\{u_1,u_2, \ldots , u_{n_c}\}$ is a finite set of $n_c=|U|$ *control places* represented graphically with circles as in (i).

(iii) $Y=\{y_1,y_2, \ldots ,y_{n_y}\}$ is a finite set of $n_y=|Y|$ *measurement places*, represented graphically with circles as in (i).

(iv) $T=\{t_1,t_2, \ldots ,t_m\}$ is a non-empty finite set of $m=|T|$ *transitions* represented graphically by line segments (|).

Note that $P \cap U \cap Y \cap T = \emptyset$. Let $\Pi=P \cup U$ and $n_\pi=|\Pi|$.

(v) $I_D:T \rightarrow \Pi^{\infty}$ is a mapping from transitions to the set of all multisets over Π. It is represented graphically by *directed arcs* (⟶) pointing from each *input place of* t_j, $\pi \in I_D(t_j)$, to t_j. If for some $\pi \in \Pi$, $\#(\pi,I_D(t_j))=k>1$, then the mapping can be represented graphically by a directed arc with multiplicity k ($\xrightarrow{k}$). Note that k is finite.

(vi) $I_N:T \rightarrow \Pi^{\infty}$ is a mapping from transitions to the set of all multisets over Π. It is represented graphically by *not arcs* (inhibitor arcs) (——o) pointing from each input place of t_j, $\pi \in I_N(t_j)$, to t_j. If for some $\pi \in \Pi$, $\#(\pi,I_N(t_j))=k>1$, then the mapping can be represented by a not arc with multiplicity k (—k—o). Note that k is finite.

(vii) $O_D:T \rightarrow P^{\infty}$ is a mapping from transitions to the set of all multisets over P. It is represented graphically by directed arcs pointing from the transition $t_j \in T$ to each *output place of* t_j, $p_i \in O_D(t_j)$. A directed arc with multiplicity k can be used in a manner similar to (v), except $k=\#(p_i,O_D(t_j))$.

(viii) $\Gamma:T \rightarrow U$ is a mapping from transitions to control places. It is represented graphically by *control arcs* (——■) pointing from each $u_i \in \Gamma(t_j)$ to t_j. It is required that $\#(u_i,\Gamma(t_j)) \leq 1$ for all $t_j \in T$ and $u_i \in U$.

(ix) $\Psi:P \rightarrow Y$ is a mapping from places to measurement places. It is represented graphically with a *connection arc* (▷——◁) from $p_i \in P$ to $y_j \in \Psi(p_i)$. In this paper we require that for all $p_i \in P$ there exist a unique $y_j \in \Psi(p_i)$.

The set $T_c=\{t_j |$ there exists some u_i, $u_i \in \Gamma(t_j)\}$ and $T_{uc}=T-T_c$. For convenience, a *two-way directed arc* (◀—▶) is used to indicate a *self loop*, i.e. for $p_i \in P$, $p_i \in O_D(t_j)$ and $p_i \in I_D(t_j)$. A *directed not arc* (◀—o) is used to indicate the connections $p_i \in O_D(t_j)$ and $p_i \in I_N(t_j)$, where $p_i \in P$. Also, every arc (except the connection arc) defined above has a transition on one end and a place on the other, and no transition or place exists without being connected to an arc.

A complete description, which also includes the *execution* characteristics of the Petri Net, is given by $P_N=(P_S,T_s,X_p,X_{po},U_p,U_a,Y_p,Y_a,E_r,\Phi)$ where:

(i) P_S is described above.

(ii) T_s is the *time index set* which is specified according to the modelling problem at hand. Often $T_s=N$ and the initial time is 0, and each successive natural number represents an arbitrary length time step. Alternatively, the natural numbers represent fixed length time units such as seconds.

(iii) $X_p:P \times T_s \rightarrow N$ is the *marking function*, a mapping from (state) places and time steps into nonnegative integers representing the *marking* of the place. The n-dimensional column vector $x_p(k)=[X_p(p_1,k),X_p(p_2,k), \ldots ,X_p(p_n,k)]^t$ is used to denote the *state* of the Petri net. The state is represented graphically by *tokens* (●) that are put inside places (e.g. $X_p(p_i,k)=2$ is represented as p_i (⦁⦁)). The complete set of "reachable states" [17] will be denoted by X_p. It is required that $X_p(p_i,k)$ be finite for all $p_i \in P$ and $k \in T_s$.

(iv) X_{po} is a non-empty finite set of *initial conditions* $x_p(0)$ for the state of the Petri Net; $N^n \supseteq X_{po}$.

(v) $U_p:U \times T_s \rightarrow N$ is a mapping from control places and time steps into nonnegative integers representing the marking of the control place. It is represented graphically with tokens as in (iii). The n_c-dimensional

column vector $u_p(k)=[U_p(u_1,k), U_p(u_2,k), \ldots, U_p(u_{n_c},k)]^t$ is used to denote the *control input* to the Petri net. It is required that $u_p(k) \in \Omega_{n_c}$ for all $k \in \mathbf{T_s}$.

(vi) $\mathbf{U_a}$ is a non-empty finite set of admissible control inputs $u_p(k)$ for the Petri Net.

(vii) Y_p:$\mathbf{Y} \times \mathbf{T_s} \rightarrow \mathbf{N}$ is a mapping from measurement places and time steps to a nonnegative integer representing the marking of the measurement place. Let $Y_p(y_j,k)=X_p(p_i,k)$ for all $k \in \mathbf{T_s}$ where $y_j \in \Psi(p_i)$. The $n_y(n)$-dimensional column vector $y_p(k)=[Y_p(y_1,k), Y_p(y_2,k), \ldots, Y_p(y_{n_y},k)]^t$ is used to denote the *output* of the Petri net.

(viii) $\mathbf{Y_a}$ is a finite set of admissible outputs $y_p(k)$ for the Petri Net, which, for this paper, is $\mathbf{X_p}$.

(ix) E_r:$\mathbf{N}^{n_\pi} \times \mathbf{T_s} \rightarrow 2^{\mathbf{T}}$ is the Petri net *enable rule*, a mapping from an n_π-dimensional column vector of nonnegative integers representing the marking of the state and control places and time steps into subsets of transitions that are said to be *enabled* at step k. The notation $t_j \in E_r(k)$ is used to indicate that t_j is enabled at step k. A transition t_j can *fire* whenever it is enabled. Tokens are redistributed in the Petri net when a transition fires according to the next state function described below. Following the definition in [17] it is assumed that no two transitions will fire at exactly the same time. The set of enabled transitions is formed at time k, then one is chosen to be fired. Also, the transitions fire only at times $k \in \mathbf{T_s}$.

(x) Φ:$\mathbf{T} \times \mathbf{N}^n \times \mathbf{T_s} \rightarrow \mathbf{N}^n$ is the Petri net *next state function (firing rule)*, a mapping from transitions, an n-dimensional column vector of nonnegative integers representing $x_p(k)$, and time steps into an n-dimensional column vector of nonnegative integers representing the next state $x_p(k+1)$. The next state function is defined at step k iff $t_j \in E_r(k)$. Suppose that a transition fires at time k (or if k=0 then none has fired), then the transition which fires next is chosen in the following manner. If $t_j \in E_r(k)$ and $t_j \in \mathbf{T_c}$ then the transition t_j is fired at time $k+\Delta t$ where $\Delta t \rightarrow 0$, i.e., immediately after the last transition firing. All enabled transitions $t_j \in \mathbf{T_c}$ are fired, then the enabled transitions $t_j \in \mathbf{T_{uc}}$ are allowed to fire asynchronously. This will allow the plant to respond immediately to an input.

A candidate for the enable rule is given by $E_r'=\{t_j | X_p(p_i,k) \geq \#(p_i,I_D(t_j)), X_p(p_i,k) < \#(p_i,I_N(t_j)), U_p(u_i,k) \geq \#(u_i,\Gamma(t_j)), U_p(u_i,k) \geq \#(u_i,I_D(t_j)), U_p(u_i,k) < \#(u_i,I_N(t_j))$, for all $p_i \in \mathbf{P}, u_i \in \mathbf{U}\}$. The enable rule is chosen based on the specific modelling task. This enable rule is used for all the examples in this paper. Let $A^-=[a_{ij}^-]$, where $a_{ij}^-= \#(p_i,I_D(t_j))$ and $A^+=[a_{ij}^+]$, where $a_{ij}^+=\#(p_i,O_D(t_j))$. Let $A=A^+-A^-$. Let $e_j(k)=[0\ 0 \ldots 1 \ldots 0\ 0]^t$ where the 1 is in the jth position $1 \leq j \leq m$. Let $C=[c_{ij}]$ with $c_{ij}=1$ if $y_i \in \Psi(p_j)$ and $c_{ij}=0$ if $y_i \notin \Psi(p_j)$. Let $\Phi=[\phi_1,\phi_2, \ldots, \phi_n]^t$. Using E_r' as the enable rule, an example next state function is $\phi_i(t_j,X_p(p_i,k),k) = X_p(p_i,k+1)=X_p(p_i,k) - \#(p_i,I_D(t_j)) + \#(p_i,O_D(t_j))$ for all i, $1 \leq i \leq n$. Hence, for $t_j \in E_r'(k)$ firing we have,

$$x_p(k+1)=x_p(k) + Ae_j(k)$$
$$y_p(k)=Cx_p(k)$$

which are similar to the *state equations* described in [17]. For this next state function tokens are not removed from any control or disturbance place if t_j fires, but they must be present to enable t_j. Likewise, tokens are not removed from places connected to transitions via a not arc. The connection arcs between any place p_i and a measurement place y_j indicate that any tokens added or subtracted from p_i are also added and subtracted from y_j, i.e. with the connection arc they are essentially duplicate places.

It must be emphasized that the Petri net defined above is a flexible modelling tool since we do not confine ourselves to the standard definitions of the arcs, enable rules, and firing rules. They can be chosen according to the modelling task at hand. For instance, in the third example on self stabilizing distributed systems it is convenient to define an arc that connects transitions $t_j \in \mathbf{T}$ to places $p_i \in \mathbf{P}$ which if t_j is enabled at step k and is fired then $X_p(p_i,k+1)=1$. Special symbols can be used for the graphical representation of such arcs. For the arc just defined we use the symbol —●▶ .

This completes the definition of the Petri net. Although it is not necessary, a controller can be used for the Petri net. One possible controller for the Petri net is given by G_c:$\mathbf{X_p} \rightarrow \mathbf{U_a}$, a mapping of states of the Petri net into admissible inputs to the Petri net. The initial controller output is the appropriate element of $\mathbf{U_a}$ depending on the initial state of the plant. Note that if a transition in the plant fires, changing $x_{pi} \in \mathbf{X_p}$ to $x_{pi'} \in \mathbf{X_p}$ the control input to the plant is updated instantaneously from $G_c(x_{pi})$ to $G_c(x_{pi'})$.

We only briefly outline how the structure **M** can be formed using the Petri net. Clearly, regardless of whether a controller is used or not, the state equations above provide for the specification of the structure **M**. The structure states **S** are the states of the Petri net and the firing of various transitions specifies the relation **R**. The function $\mathbf{P_c}$ depends on what is to be verified. For instance, if the input-output properties of the plant are of interest then the atomic propositions may be elements of the sets $\mathbf{U_a}$ and $\mathbf{Y_a}$. The CTL structure also depends on the initial state. If there is more than one initial state then the CTL structure may have to be produced for each initial state (depending on if the initial condition is in the CTL structure produced for another initial condition). With **M**, and the design specifications stated in the branching-time temporal language, the verification algorithm is used to prove whether the specifications are true or not.

Several examples of the modelling methodology, design specifications, and their verification are given in the next section.

4.0 Examples

In this section we briefly describe how the above results are applied to three examples. For the first example, consider the surge tank taken from [9] shown in Figure 4.1 with a sensor distinguishing among five liquid levels (empty, low, normal, high, and full) and with the controller simply opening and closing a fill valve. Unpredictable users operate an empty valve to drain the tank. The objective is to verify that a given controller will properly regulate the liquid level in the tank.

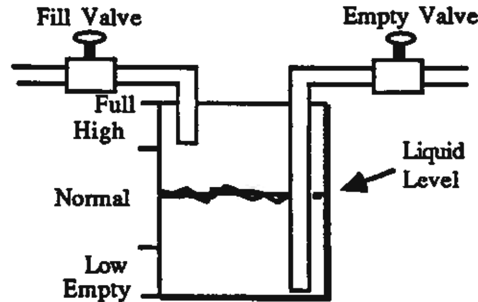

Figure 4.1 Surge Tank

The liquid level "empty" indicates that the tank is absolutely empty, hence if the fill valve is opened the tank will immediately become "low". The liquid level of "low" means that the liquid height is between empty and "normal". "High" is between normal and "full", which indicates that the tank (which is sealed at the top) is completely full. The Petri net model of the surge tank is given in Figure 4.2 below.

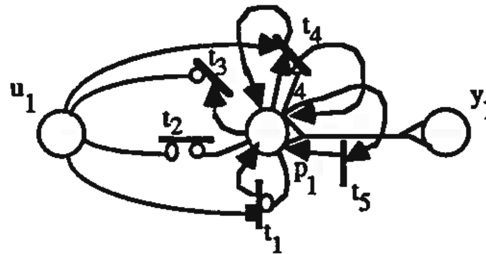

Figure 4.2 Petri Net Model of the Surge Tank

The place p_1 is used to represent the state of the system which is the liquid height. The interpretation of time is that each successive $k \in T_s$ represents one second. The cases where $x_p(k)=X_p(p_1,k)=i$, with $i=0,1,2,3,4$ represent the five liquid levels in the tank (empty, low, normal, high, and full respectively). Denote these states with atomic propositions x_{pi}, $i=0,1,2,3,4$ respectively. The connection arc joins the measurement place y_1 to the state place p_1, hence the measured output is $y_p(k)=x_p(k)$, the state of the system. The input place u_1 represents the fill valve, and $u_p(k)=1$ indicates that the fill valve is on while $u_p(k)=0$ indicates that it is off. The set of transitions $T=\{t_1,t_2,t_3,t_4,t_5\}$. Each of these transitions specifies certain rules about the behavior of the plant. Transition (i) t_1 indicates that if the liquid level is empty ($x_p(k)=0$) and the fill valve is on, then the liquid level will immediately become low ($x_p(k)=1$), (ii) t_2 indicates that if the liquid level is empty and the fill valve is closed then the liquid level will stay empty, (iii) t_3 models the case where if there is a liquid level above empty and the fill valve is off then the liquid level may drop one level depending on the empty valve, (iv) t_4 indicates that if the liquid level is less than full ($x_p(k)<4$) and it is greater than or equal to the empty level and the fill valve is on, then the liquid level may raise depending on the empty valve, and (v) t_5 models the case where if the liquid level is above empty it may be that the next state is exactly the same as the previous one.

Next, the controller for the surge tank is specified. For the surge tank the set of reachable states is $X_p=\{0,1,2,3,4\}$ and the set of admissible inputs is $U_a=\{0,1\}$. The controller for the surge tank, G_c, is given by: (i) $G_c(0)=G_c(1)=G_c(2)=1$ and (ii) $G_c(3)=G_c(4)=0$. Using the controller and the Petri net model of the surge tank, the CTL structures appropriate for each of the following design specifications were generated for the verification algorithm.

The first two design specifications involve verifying that if the liquid level is initially normal ($x_p(0)=2$) and the fill valve is open then (i) it will not be the case that the tank will become full or empty, and (ii) it is inevitable that the level will always be either normal or full. Stated formally these two design specifications are:

(i) $\forall p \boxdot (\neg(x_{p0} \vee x_{p4}))$, and

(ii) $\forall p \Diamond (\forall p \boxdot (x_{p2} \vee x_{p3}))$

respectively. These two specifications were easily verified using the algorithm described in Section 2.2. The third and fourth design specifications consider the case when the tank is initially empty ($x_p(0)=0$) and the fill valve is initially off. The two specifications are:

(iii) $\forall p \Diamond (\exists p \boxdot (x_{p2} \vee x_{p3}))$, and

(iv) $\exists p \Diamond (\exists p \boxdot (x_{p2} \vee x_{p3}))$.

The verification algorithm was used to show that the third design specification is false and the fourth one is true. The third specification is not true because it is possible that the liquid level end up being forever at the low level and the fourth one is true because it is possible for the controller to raise the level if the empty valve is off sufficiently often.

We use the <u>manufacturing system</u> originally studied in [9] for the second example. We have one machine which must process different parts from two buffers (B1 and B2) each supplied by two different producers (P1 and P2). Upon completion of processing, each type of part is put in its respective output bin. A controller must ensure mutual exclusion in the machine, i.e., only one part of one type is processed at a time. Also, the controller must also ensure that producer P1 gets priority in the use of the machine. To perform its task, the controller can sense whether there is a part in a given buffer and can also determine the type of part in the machine. The controller acts to allow the different parts to enter the machine. The Petri net model of the machine is given in Figure 4.3 below.

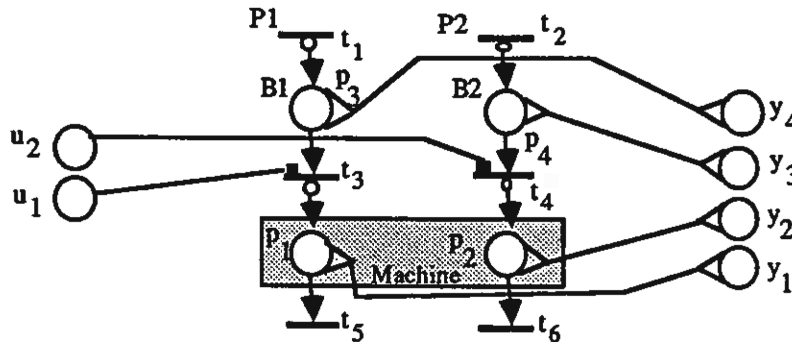

Figure 4.3 Petri Net Model of a Manufacturing System

The producer P1 (P2) is represented with the transition t_1 (t_2). The buffer B1 (B2) is represented with the place p_3 (p_4) and measurement place y_4 (y_3) is used to indicate if a part is in buffer B1 (B2). The place p_1 (p_2) and measurement place y_1 (y_2) are used to represent whether or not a P1 (P2) type part is in the machine. The firing of t_3 (t_4) indicates that parts from B1 (B2) are put in the machine and the firing of t_5 (t_6) indicate that parts are removed from the machines and put into their respective output bins.

The interpretation of time is that each $k \in T_s$ represents an arbitrary length time step. The state $x_p(k)=[X_p(p_1,k)\ X_p(p_2,k)\ X_p(p_3,k)\ X_p(p_4,k)]^t$. Hence the state $x_p(k)=[1\ 0\ 0\ 1]^t$ means that there is a P1 part in the machine, there is not a P2 part in the machine, there is not a P1 part in buffer B1, and there is a P2 part in buffer B2. We shall use the notation "1001" to represent this state and a similar notation for other states. The input $u_p(k)=[U_p(u_1,k)\ U_p(u_2,k)]^t=[1\ 0]^t$ indicates that a P1 part is to be allowed to be put into the machine. The notation "10" will be used to denote this input and others will be denoted similarly. The set of admissible inputs is contains 10,01, and 00. The controller for the manufacturing system G_c is given by: (i) $G_c(x)=10$ if $x=0010$ or 0011, (ii) $G_c(x)=01$ if $x=0001$, and (iii) $G_c(x)=00$ for all other reachable states. The initial conditions are that the state of the plant is 0000 and the controller output is 00. The CTL structure was generated using the Petri net model of the manufacturing system and the following specifications were considered:

(i) $\forall p \boxdot (\neg(1100 \vee 1101 \vee 1110 \vee 1111))$

(ii) $\forall p \boxdot (0011 \rightarrow \forall s \odot (1001 \vee 1011))$

The first specification is a statement of mutual exclusion. It was verified first. The second specification says that producer P1 should have priority over producer P2. In particular, it says that if it is ever the case

where there are parts in both of the buffers then the P1 part should be allowed to enter the machine. This CTL formula was also verified using the verification algorithm. This manufacturing example is similar to the "Two Class Parts Processing" example in [18]. There, however, Thistle and Wonham allow an arbitrary finite number of parts of one type to enter the machine. This forces their controller to have an infinite number of states, so the verification algorithm cannot be applied to their example. There are, however, many practical problems that are finite state.

The third example is taken from Dijkstra's paper "Self Stabilizing Systems in Spite of Distributed Control" where he considers the existence of self stabilizing distributed systems and provides three systems that are supposed to possess this property [5]. Thistle has used a certain linear temporal logic proof system to prove that one of Dijkstra's systems is self stabilizing (although in a different manner than originally thought by Dijkstra) [18]. In this example we shall prove, using the verification algorithm, that another of Dijkstra's machines that is supposed to be self stabilizing is in fact not. The problem formulation follows the one in [18].

The sort of distributed systems considered are those that can be modelled as a sparsely connected finite graph having at each node a finite state machine. Each of these machines can communicate only with its "neighbors", i.e., those machines located at nodes directly connected to its own. If a machine makes a state transition, or "move", its new state is a function only of its previous state and the states of its neighbors. Also, the moves of a machine can be enabled or disabled only on the basis of the states of the machine and its neighbors: in order for a move to occur, a corresponding Boolean condition on the states of the machine and its neighbors must be satisfied. These conditions are called "privileges", and when satisfied, a privilege is said to be "present". A "demon" selects at random one of the privileges that is present and the corresponding move occurs; consequently asynchronous operation of the machine is modelled.

Dijkstra considers a distributed system to be "self stabilizing" if for all initial states it eventually reaches a "legitimate" state. The set of legitimate states can be defined to be any set having the following properties: (i) in every legitimate state, at least one privilege is present; (ii) from any legitimate state, all enabled moves bring the system into another legitimate state; (iii) each privilege is present in at least one legitimate state; and (iv) from any legitimate state it is possible to reach any other legitimate state (as long as the demon happens to select privileges appropriately). The concern is with whether self stabilizing distributed systems exist that are "non-trivial", meaning that not all system states are legitimate states. Dijkstra proposes three solutions to this problem, but he leaves the proof that the systems are self stabilizing to the reader. We shall study his second solution, the "Solution with Four-State Machines".

Dijkstra confines the his study to N+1 machines placed in a "ring", and considers legitimate states to be the states in which exactly one privilege is present. In his solution using four-state machines each machine in the ring has four possible states. We consider the simple case where there are only four machines in a ring shown in Figure 4.4.

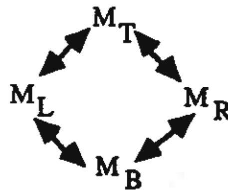

Figure 4.4 Ring of Four Machines

The arcs represent communications between the various machines. The state of each of the four machines is represented with two Boolean variables:

Machine	M_T	M_R	M_B	M_L
Position	Top	Right	Bottom	Left
Boolean State Variables	x_T, x_{Tu}	x_R, x_{Ru}	x_B, x_{Bu}	x_L, x_{Lu}

Although, in general the Boolean state variables can take on values of 1 (True) or 0 (False), Dijkstra restricts the values of the top and bottom machine's states. For the bottom machine $x_{Bu}=1$, and for the top machine $x_{Tu}=0$ by definition. These machines are therefore only two-state machines. If x is a Boolean variable let $x:=0$ ($x:=1$) denote the assignment of false (true) to x. Also, let $x:=\neg x$ denote the switching of the truth value of x. To complete the definition of the distributed system the privileges are defined with a set of rules as follows:

(i) **If** $x_B=x_R$ **and** $x_{Ru}=0$ **Then** $x_B:=\neg x_B$
(ii) **If** $x_T \neq x_L$ **Then** $x_T:=\neg x_T$

(iii) **If** $x_R \neq x_B$ **Then** $x_R := \neg x_R$ **and** $x_{Ru} := 1$
(iv) **If** $x_R = x_T$ **and** $x_{Ru} = 1$ **Then** $x_{Ru} := 0$
(v) **If** $x_L \neq x_T$ **Then** $x_L := \neg x_L$ **and** $x_{Lu} := 1$
(vi) **If** $x_L = x_B$ **and** $x_{Lu} = 1$ **and** $x_{Bu} = 0$ **Then** $x_{Lu} := 0$

Before the Petri net model is given a few observations are in order. First, notice that since $x_{Bu}=1$ by definition the sixth rule will never be used. Hence, there is no communication between the bottom machine and the machine to its left even in the general case where there are N+1 machines in the ring. The result is that the "ring" is not a ring at all but a string of communicating finite state machines. This does not hinder our ability to determine if the system is self stabilizing. The second observation is that the left hand side of all the rules (i)-(v) does not depend on x_{Lu}; consequently no privileges depend on it and it will not affect which moves are made in the system. It is therefore not necessary to model it, so M_L can also be considered a two-state machine. The Petri net model of the string of finite state machines is shown in Figure 4.5.

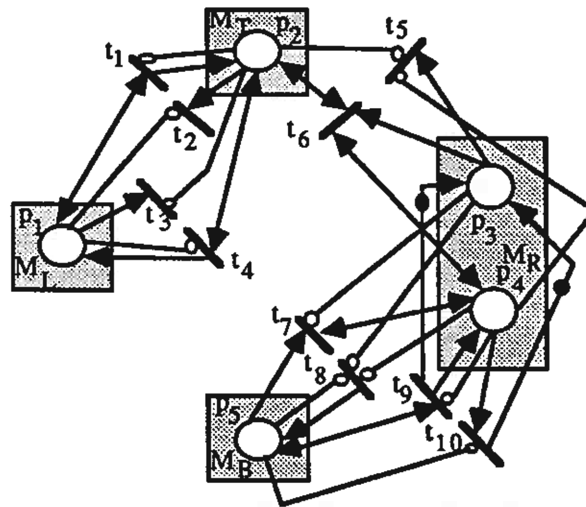

Figure 4.5 Petri Net Model of a Distributed System

The interpretation of time is that each $k \in T_s$ represents an arbitrary length time step. Notice that places were used to represent each of the machines and that $X_p(p_1,k)=x_L$, $X_p(p_2,k)=x_T$, $X_p(p_3,k)=x_{Ru}$, $X_p(p_4,k)=x_R$, and $X_p(p_5,k)=x_B$. The transitions t_j, $j=1,2, \ldots ,10$ represent the rules that describe the privileges. For instance, privilege (i) is represented with transitions t_1 and t_2. The function of the special arc —●—▶ was explained at the end of Section 3.0; it is used to make the assignment $x_{Ru}:=1$. The state $x_p(k)=[0\ 0\ 0\ 1\ 1]^t$ indicates that $x_B=1$ and $x_R=1$ and that the other state variables are equal to zero. As in the manufacturing system example we shall use 00011 to denote this state and other states will be denoted in a similar manner. The set of initial states $X_{po}=X_p$, the set of reachable states. Notice that there is no explicit representation of a controller. The CTL structure is formed by considering in turn each initial condition and generating the structure for that particular initial condition by firing various transitions.

The verification algorithm was used to prove that the specification $\exists p \Diamond (00111 \vee 11100)$ is true, i.e., that there exists a path from any state to the state 00111 or 11100. But notice that in these states the distributed system is deadlocked. This is easily seen by examining the above Petri net. Clearly, Dijkstra's solution is invalid since it does not satisfy the criteria necessary for the set of legitimate states. Note that the proof here is only for the case when N=3, i.e., four machines. It may be the case that the system is self stabilizing for N>3.

Acknowledgement:

The authors gratefully acknowledge the partial support of the Jet Propulsion Laboratory under contract No. 957856.

5.0 References

[1] Ben-Ari M., Pnueli A., Manna Z., "The Temporal Logic of Branching Time", Acta Inf., Vol. 20, pp. 207-226, 1983.

[2] Browne M.C., Clarke E.M., "SML - A High Level Language for the Design and Verification of Finite State Machines", IFIP WG 10.2 International Working Conference from HDL Descriptions to Guarantee Correct Circuit Designs, Grenoble, France, Sept. 1986.

[3] Clarke E.M., Emerson E.A., Sistla A.P., "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach", Tenth ACM Symp. on Principles of Programming Languages, Austin, Texas, 1983.

[4] Clarke E.M., Browne M.C., Emerson E.A., Sistla A.P., "Using Temporal Logic for Automatic Verification of Finite State Systems", Apt K.R., ed., Logics and Models of Concurrent Systems, Springer Verlag, NY, 1985.

[5] Dijkstra E.W., "Self-stabilizing Systems in Spite of Distributed Control", Comm. of the ACM, Vol. 17, No. 11, pp. 643-644, Nov. 1974.

[6] Emerson E.A., Halpern J.Y., ""Sometimes" and "Not Never" Revisited: On Branching versus Linear Time Temporal Logic", Journal of the Association for Computing Machinery, Vol. 33, No. 1, pp. 151-178, January 1986.

[7] Ichikawa A., Hiraishi K., "Analysis and Control of Discrete Event Systems Represented by Petri Nets", in Varaiya P., Kurzhanski A.B., eds., Discrete Event Systems: Models and Applications, Lecture Notes in Control and Information Sciences, Springer Verlag, NY, 1987.

[8] Inan K., Varaiya P., "Finitely Recursive Process Models for Discrete Event Systems", IEEE Trans. on Automatic Control, Vol. 33, No. 7, pp. 626-639, July 1988.

[9] Knight J.F., Passino K.M., "Decidability for Temporal Logic in Control Theory", Proc. of the Allerton Conf. on Communication, Control, and Computing, Univ. of Illinois, Urbana-Champaign, pp. 335-344, Oct. 1987.

[10] Krogh B., "Controlled Petri Nets and Maximally Permissive Feedback Logic", Proc. of the Allerton Conf. on Communication, Control, and Computing, Univ. of Illinois, Urbana-Champaign, pp. 317-326, Oct. 1987.

[11] Lamport L., ""Sometime" is Sometimes "Not Never": On the Temporal Logic of Programs", ACM Symp. on Principles of Programming Languages, pp. 174-185, 1980.

[12] Manna Z., Wolper P., "Synthesis of Communicating Processes from Temporal Logic Specifications", Dept. of Computer Science, Stanford Univ., Report No. STAN-CS-81-872, 1981.

[13] Manna Z., Pnueli A., "Verification of Concurrent Programs: a Temporal Proof System", Dept. of Computer Science, Stanford Univ., Report No. STAN-CS-83-967, 1983.

[14] Ostroff J. S., Real-Time Computer Control of Discrete Systems Modelled by Extended State Machines: A Temporal Logic Approach, PhD Dissertation, Report No. 8618, Dept. of Elect. Eng., Univ. of Toronto, Jan. 1987.

[15] Passino K.M., Antsaklis P.J., "Artificial Intelligence Planning Problems in a Petri Net Framework", Proc. of the American Control Conf., pp. 626-631, Atlanta GA, June 1988.

[16] Passino K.M., Antsaklis P.J., "Planning Via Heuristic Search in a Petri Net Framework", Proc. of the Third IEEE Int. Symposium on Intelligent Control, Arlington, VA, Aug. 1988.

[17] Peterson J.L., Petri Net Theory and the Modelling of Systems, Prentice-Hall, NJ 1981.

[18] Thistle J. G., Wonham W. M., "Control Problems in a Temporal Logic Framework", International Journal Control, Vol. 44, No. 4, pp. 943-976, 1986.