The StateClock tool and clockcharts

Jonathan S. Ostroff
Department Of Computer Science, York University,
4700 Keele Street, North York Ontario, Canada, M3J 1P3.
Email: jonathan@yorku.ca    Tel: 416-736-2100 X77882   Fax: 416-736-5872.

Abstract: The purpose of this paper is to give a theoretical description of clockcharts, and their implementation in the StateClock tool for COSC4352 students.

1.0 Temporal logic and fair transition systems

In the sequel, we use relative quantification \((Qx:T|R:P)\) where \(Q\) is a quantifier (e.g. \(\forall\) or \(\exists\), \(T\) is the type of the dummy variable \(x\), \(R\) is the range of the dummy variable and \(P\) a predicate [Gries, 1993 #369]. For example, \((\forall i:INTEGER|3 \leq i:P)\) means “for all values of an integer variable \(i\), if \(i\) is at least as large as 3 then \(i\) has property \(P\)”. If no range is supplied then it is true.

We distinguish between 3 kinds of logical descriptions:SDs, CDs and TLDs.

- An \(SD\) is a state-description (also called a state-formula). A state-description is a predicate that can be evaluated to true or false in a single state. For example, a description \(d\) defined by \(d \equiv b \land (\exists i:INTEGER|y = 2i)\) is satisfied in the state \(s = \langle b: true, y: 6 \rangle\) (written \(s \vDash f\)). The free variables of \(d\) are called its alphabet \(\alpha\), i.e. \(d.\alpha \equiv \{b, y\}\). The variable \(i\) is a dummy variable, and hence is not part of the alphabet. Each variable \(v\) in the alphabet has a type denoted by \(v.type\).

- A \(CD\) is a command-descriptions, i.e. a predicate whose alphabet is a set of primed and unprimed variables. The purpose of such a description \(d\) is to describe a change that occurs between a prestate \(s\) and a possible poststate \(s'\) under the execution of a command (e.g. a program statement, transition or routine). We will need two names for a variable \(v \in d.\alpha\) — the value of the variable in the poststate will be recorded under the primed name \(v'\), whereas the unprimed name \(v\) will stand for the value of the variable in the prestate. For example, the command-description \(d\) where

\[
d \equiv (x = 4) \land (x' = x + 1) \land (y' = z + 2) \land (z' = z)
\]

(Eq. 1)

represents the effects of a command that increments \(x\) by one, changes \(y\) to two more than the prestate value of \(z\), and keeps \(z\) the same. If a command-description \(d\) is satisfied in the pair of states \(\langle s, s' \rangle\) where \(s\) is the prestate and \(s'\) is the poststate of the command, then we write \(\langle s, s' \rangle \vDash d\). The set of variables \(d.\alpha.in = \{x, y, z\}\) is called the

1. This research was supported with the help of NSERC (National Science and Engineering Research Council of Canada).
input alphabet of the description, and \( d.\alpha_{out} = \{x', y', z'\} \) the output alphabet. The alphabet \( \alpha \) is defined as \( \alpha \equiv \alpha_{in} \cup \alpha_{out} \). Given a command-description \( d \), we define its enabling condition \( d.\alpha_{enb} \) by

\[
d.\alpha_{enb} \equiv (\exists d.\alpha_{out}|d).
\]

Using the above definition (Eq. 2) for the description (Eq. 1), we obtain:

\[
d.\alpha_{enb} \equiv (\exists x'y'z')(x' = 4) \land (x' = x + 1) \land (y' = z + 2)
\]

\[
= (x = 4)
\]

- A TLD is a temporal-logic-description. The atoms of temporal-logic-descriptions are SDs and PUDs. In addition, temporal connectives may be applied to the atoms. The satisfaction of a temporal logic description is an infinite sequence of states. The next subsection describes temporal logic in more detail.

### 1.1 Temporal Logic Descriptions

Linear time temporal logic [Manna, 1992 #235] uses temporal connectives such as □ (henceforth), ○ (next), ◊ (eventually), ∀ \( u \) (until) and past operators such as ⊕ (previous state) to represent qualitative temporal properties. The temporal connectives are applied to SDs or PUDs to obtain temporal logic formulas.

A temporal logic formula such as \( ◊f \) (“eventually \( f \) is true”) cannot be interpreted in a single state (such as a SD) or even in a state pair (such as a CD); rather it is evaluated in an infinite sequence of states \( \sigma \) given by \( \sigma = s_0s_1s_2...s_i... \) where \( \sigma \models ◊f \) (“\( \sigma \) satisfies \( ◊f \)”) will mean that there is at least one state subsequent to the initial state that satisfies \( f \). An inductive definition of the satisfaction relation \( \models \) can then be given. Let \( (\sigma, i) \models f \) denote the satisfaction of temporal formula \( f \) at a position \( i \geq 0 \) of the sequence \( \sigma \). If \( f \) is an SD, then \( [(\sigma, i) \models f] \equiv [s_i \models f] \), and if \( f \) is a CD, then \( [(\sigma, i) \models f] \equiv [(s_p, s_{i+1}) \models f] \).

We can then give the appropriate inductive definitions for the propositional connectives (e.g. negation, conjunction, implication) followed by the usual definition of the temporal operators. For example, for temporal logic formulas \( g \) and \( h \), the until operator is defined by \( [(\sigma, i) \models g \looparrowright h] \equiv (\exists j \exists i:(\sigma, j) \models h \land (\forall k)i \leq k < j:(\sigma, k) \models g)) \). For an arbitrary temporal logic formula \( f \), \( \sigma \models f \) is an abbreviation for \( (\sigma, 0) \models f \). A formula \( f \) is generally-valid iff \( (\forall \sigma)(\sigma \models f) \), and we write \( \models f \).

The implication \( (f \rightarrow ◊g) \) states only that “\( f \) implies eventually \( g \)” at the initial position of the computation, i.e. if \( f \) holds at the initial position then there is a subsequent position where \( g \) holds. As a notational convenience, we will write \( f \Rightarrow ◊g \) for \( □(f \rightarrow ◊g) \) which states that the implication holds at all positions of the sequence. In general, the double arrow entails operator is defined by \( [p \Rightarrow q] \equiv □[p \rightarrow q] \) for any temporal logic formulas \( p \) and \( q \).

### 1.2 Fair transition systems

Manna and Pnueli introduced the notion of a fair transition system as a mathematical description of a reactive program. We refer the reader to [Manna, 1992 #235] for methods of transforming a reactive program, with many concurrent processes, into a fair transition system.
The StateClock tool and clockcharts March 2, 1998

We provide below a brief account of the notion of a Manna-Pnueli fair transition system, and then use it to describe the set of all the computations of a reactive program. A fair transition system $FTS$ is defined as a 5-tuple $FTS = (V, I, T, J, C)$ where:

- $V$ is a finite set of system variables, some of which are control variables (representing the locations of execution of a concurrent process) and others are data variables explicitly manipulated by the program text. Each variable $v \in V$ has a corresponding type $v.type$ which is the set of values that the variable may have.

- $I$ is the initial condition, which is a satisfiable boolean valued expression in the system variables that characterizes the states at which the execution of the system can begin. A state $s$ satisfying $I$ is called an initial state.

- $T$ is a finite set of transitions. Each transition $\tau \in T$ is a function $\tau : \Sigma \rightarrow \text{powerset}(\Sigma)$ that maps a prestate $s$ in $\Sigma$ to a set of $\tau$-successor states $\tau(s) \subseteq \Sigma$. A successor state $s'$ is also called a poststate of $\tau$ from $s$. The transition set always has a special transition called idling, which can always occur from any state without changing any variables.

- $J$: a justice set where $J \subseteq T$. Informally, the justice constraint for each transition $\tau \in J$ disallows computations in which $\tau$ is continually enabled but is taken only finitely many times.

- $C$: a compassion set where $C \subseteq T$. Informally, the compassion constraint for each transition $\tau \in C$ disallows computations in which $\tau$ is enabled infinitely often but is taken only finitely many times.

The effect of taking a transition $\tau \in T$ is often represented by a command description $\tau.p$, called a transition relation, which is a predicate in the primed and unprimed system variables. The corresponding enabling condition (Eq. 2) is denoted $\tau.enb$. To emphasize that a transition $\tau$ is taken at a position $i \geq 0$ of a computation $\sigma$, we use the abbreviation $\text{taken}(\tau) \triangleq \tau.p$, i.e. $(\sigma, i) \models \text{taken}(\tau)$ iff $\langle s_i, s_{i+1} \rangle \models \tau.p$.

A transition can also be described by providing its enabling condition $\tau.enb$, and a simultaneous assignment function $\tau.asg$. Let the system variables be $V = \{w, x, y, z\}$. An assignment function such as $\tau.asg = \{x := e_1, y := e_2\}$, where $e_1$ and $e_2$ are expressions in the system variables of the appropriate type, indicates that $x$ and $y$ are assigned the values of the expressions $e_1$ and $e_2$ respectively. No other system variables are changed.

Given an enabling condition and assignment function, the corresponding transition relation $\tau.p$ is $\tau.p \triangleq (\tau.enb) \land (x' = e_1) \land (y' = e_2) \land \text{pre}(\{w, z\})$, where if $U \subseteq V$ then $\text{pre}(U) \triangleq \forall v|v \in V: (v' = v)$. We often abbreviate $\text{pre}(\{w, z\})$ to $\text{pre}(w, z)$.

A computation $\sigma$ of a fair transition system $(V, I, T, F)$ is an infinite sequence of states $s_0s_1s_2\ldots$ satisfying the following three constraints:

1. Initialization constraint: The first state of the computation satisfies the initial condition, i.e. $s_0 \models I$.

2. Succession constraint: $(\forall i|i \geq 0: s_{i+1} \in \tau_{i+1}(s_i))$, i.e. every prestate at position $i$ must be followed at position $i + 1$ by a $\tau$-successor where $\tau \in T$. This can be described in temporal logic by: $\Box(\exists \tau:T|\text{taken}(\tau))$. 


3. **Justice constraint**: Informally, the justice constraint for each transition \( \tau \in J \) disallows computations in which \( \tau \) is continually enabled but is taken only finitely many times. This can be described in temporal logic by: \((\Box \Box \tau.enb) \rightarrow (\Box \Box \text{taken}(\tau))\).

4. **Compassion constraint**: Informally, the compassion constraint for each transition \( \tau \in C \) disallows computations in which \( \tau \) is enabled infinitely often but is taken only finitely many times. This can be described in temporal logic by: \((\Box \Box \tau.enb) \rightarrow (\Box \Box \text{taken}(\tau))\).

### 1.3 Visual descriptions based on statecharts

The StateTime tool [???] introduced a statechart based language, called TTMcharts, for the graphical description of transition systems. In the sequel, we will continue to use the graphical constructs of TTMcharts, but with additional features such as the ability to start and stop clocks; these new charts will be called **clockcharts**.

**FIGURE 1. A TTMchart “a”**

Parallel object \( a \) is composed of sub-objects \( b \) and \( c \). Serial object \( c \) is composed of sub-objects \( c1 \) and \( c2 \). Serial object \( c2 \) is composed of primitive objects (states) \( c21 \), \( c22 \) and \( c23 \). Control variables are: \( cv1, cv2, cv3 \) where \( cv1.type = \{b1,b2\} \), \( cv2.type = \{c1,c2\} \), and \( cv3.type = \{c21,c22,c23\} \). The **default** objects are denoted by an arrow without a source.

There is one data variable: \( g \): BOOLEAN.

In general, an **event** is denoted \( E[l,u]: \text{guard} / \text{action} \), where \( \text{guard} \) is an SD in the system variables, \( \text{action} \) is a (possibly simultaneous) assignment of data variables, and \( l \) and \( u \) lower and upper time bounds on the transition.

Initially: \((cv1 = b1) \land (cv2 = c1) \land (cv3 = c21) \land g\).

When converting a TTMchart into a fair transition system, each event becomes a transition. For example, the transition relation for event \( E1 \) is:

\((cv2 = c1) \land g \land (cv2' = c2) \land (cv3' = c21) \land (cv1' = cv1) \land (g' = \neg g)\).

An example of a chart is given in Fig. 1. A chart is a hierarchy of **objects**. Objects describe control information and impose structure on the operation of the system. An object is either **primitive**, **parallel** (called AND in statecharts) or **serial** (XOR in statecharts). A primitive object has no internal structure. A parallel object is constructed from a collection of child objects (or sub-objects) by parallel composition. The parallel composition of child objects operates in all of these child objects simultaneously. The entry into a parallel object via an event causes the simultaneous entry into each of the child objects. The exit from the object causes the simultaneous exit from all its children. A serial object is constructed from a collection of child objects such that only one of the children operates at a time. The entry and exit from a serial object via an event causes the simultaneous entry and exit of the currently operating child object.

Charts may have **data variables** which are tested and set by events. Each non-primitive serial (XOR) object has a **control variable** which is used to indicate which of its children is
currently operating. In the figure we given an example of how charts may be converted into fair transition systems.

1.4 Object oriented description of fair transition systems.

In the next section we will introduce the notion of a clock transition system. It will be convenient to give an abstract description of the relevant data structures of fair and clock transition systems using the object oriented descriptive notation of classcharts in the spirit of [Meyer, 1997 #373].

The basic graphical element in a classchart is the class interface shown in Fig. 2. The purpose of a class interface is to describe the visible features of an abstract data type (i.e. its queries, commands and class invariants). A query is either an attribute (i.e. a state variable of a well-defined type) or a typed function. Since the internal state of an object is not part of its interface, we do not need to distinguish between attributes and functions. Conceptually, an exported attribute can be viewed as a function returning the value of some hidden state information.

A query returns information about the state of the object without changing the object. A command, by contrast, does not return a value; instead, it changes the state of the objects (i.e. the object attributes). A command is usually described abstractly by its pre/postconditions and the class invariant. We replace the pre/postconditions with an equivalent command-description whose input alphabet is the set of class attributes. For example, consider a class with attributes value and mode and a command start. Then, instead of specifying start by

\begin{verbatim}
start
  require mode = stopped
  ensure (value = 0) \& (mode = up)
\end{verbatim}

we instead write: \(start \equiv (mode' = up) \& (value' = 0) \& (mode = stopped).\)

Fig. 2 also shows restricted features, i.e. features that are available only to certain sets of classes (e.g. classes A and B).

The classchart of a fair transition systems is given in Fig. 3. The description below the chart explains how it captures the essential data structures of fair transition systems. Classes are either in a client/supplier relationship with each other (denoted by the bold arrows in Fig. 3), or inherit from other classes (denoted by ordinary arrows). For example CLOCK inherits from VARIABLE. Thus CLOCK will have the same features as VARI-
ABLE, except for those features that are redefined or added. In general, suppose a class B has feature \( f \). If class C inherits from B, then the notation \( f^+ \) in B means that the feature is
redefined in B. In redefining a feature, inherited preconditions may only be weakened, and inherited postconditions may only be strengthened.

We use the notion of command-descriptions in two different but related ways. (a) In Fig. 3, the class CD is the template for all command-descriptions in a set of instances of VARIABLES (intended to represent the system variables of a fair transition system). The query *otr* (ordinary transition relation) of a transition is declared to be of class CD, and it returns a predicate in the primed and unprimed instances of VARIABLES. (b) A class having a command feature that changes the class attributes, is specified by a command description. Hence an instance of CLOCK may have a command *start* that is specified by a command-description in the clock attributes (*mode* and *value*).

### 2.0 Clocked Transition Systems

A clocked transition system (CTS) is a fair transition system that is enhanced as follows:

- One of the transitions is a *tick* transition that must be taken infinitely often. The *tick* transition represents a conceptual global clock that provides a uniform notion of time for distributed processes. A special transition variable *ε* is added to the system variables, which is needed for describing event occurrences.

- The user may declare and use count-up or count-down local *clocks*, that are updated in lockstep with the *tick* transition.

- Each transition has a *timer* attribute that is used to describe the lower and upper time bounds of the transition. The transition cannot be taken until the timer has reached the lower bound *low*, but must be taken by the upper time bound *hi* provided it is continuously enabled. The *timer* must also increment in lockstep with the *tick*, and be reset to zero if the transition is taken or becomes disabled.

The resulting clock transition system is still a fair transition system and hence can be used with the STEP tool [Manna, 1994 #277] for reactive system verification. We now describe each of the enhancements.

#### 2.1 The transition variable and tick transition

Since we envisage that a transition *τ_i* causes a transfer from state *s_i* to state *s_{i+1}*, we may rewrite the infinite sequence of states *σ* as:

$$
σ = (s_0, τ_0)(s_1, τ_1)(s_2, τ_2)…
$$

(Eq. 3)

The system starts in an initial state *s_0*, takes the transition *τ_0* which puts the system in state *s_1*. The transition *τ_1* takes the system from state *s_1* to *s_2* and so on.

The distinguished variable *ε* (the *transition variable*) is always part of the state. The transition variable is used to record the last event taken, i.e. for the sequence *(s_0, τ_0)(s_1, τ_1)(s_2, τ_2)…* we have that *s_0*(ε) = *void* and (∀ i ≥ 0: *s_i*(ε) = *τ_{i-1})*. If *T* is the set of all transitions (including *tick* and *idling*), then ε.type = *T*.

The transition variable can be used to refer directly to event occurrences in temporal logic formulas. For example, the formula (*ε = turn_red*) ⇒ ◯(*ε = turn_green*) asserts that anytime the light turns red, it must eventually turn green.
The special transition *tick* represents the progress of a conceptual global clock, which is used to provide a uniform notion of time. A *timed computation* $\sigma$ must satisfy the temporal logic formula $\Box\Diamond (\varepsilon = \text{tick})$ (i.e. $\sigma \models \Box\Diamond (\varepsilon = \text{tick})$), which asserts that there are an infinite number of ticks occurring in the computation. Thus, time must progress irrespective of what happens in the system or its environment.

Of course, it is possible for any finite number of transitions to occur between two ticks. For example, in the computation

$$(s_0, \text{reboot})(s_1, \text{go\_red1})(s_2, \text{go\_red2})(s_3, \text{tick})(s_4, \text{tick})(s_5, \text{go\_green1})\ldots$$

after a computer reboot, two traffic lights are first set to red. After two ticks, the first light is set to green. Various other transitions may occur before the next tick.

A CTS $M$ is a 5-tuple $(M.V, M.I, M.T, M.J, M.C)$. We will use the classes $\text{CLOCK}$, $\text{TICK}$ and $\text{CLOCK\_TRANSITION}$ (Fig. 4) in the sequel to describe the elements of the 5-tuple. In particular, $\text{TICK}$ and $\text{CLOCK\_TRANSITION}$ inherit from class $\text{TRANSITION}$ (Fig. 3). An important function of our description is to show how the transition relations of these inherited class are redefined (see (Eq. 7) and (Eq. 11)) for the case of clock transition systems.

### 2.2 Clocks

A system may be described using any number of user designated clocks. Clocks can be started and stopped by appropriate transition update actions. The clocks have to be updated in lockstep with the *tick* transition. A clock is an instance of the $\text{CLOCK}$ class.

#### The $\text{CLOCK}$ class

The class $\text{CLOCKS}$ inherits all the features of $\text{VARIABLE}$, i.e. it has a *value*, a *type* (a non-negative integer as indicated by the invariant), and $\text{group} = \text{clock}$. The *value* attribute of the clock contains the current time of the clock (initially set to zero). The added features are:

- the private attribute *mode* represents whether the clock is stopped, counting up or counting down.
- the public query *ticking* returns information about the clock mode.
- commands *start*, *stop*, *count\_up* and *count\_down* can be invoked by transitions to control clocks. The command *lockstep* is used by the *tick* transition to keep all the clocks synchronized with it.

We now provide definitions for features whose definition was not given in Fig. 4. The query *ticking* is defined by

$$ticking \equiv (\text{mode} = \text{up}) \lor (\text{mode} = \text{down}). \quad \text{(Eq. 4)}$$

If we declare $c: \text{CLOCKS}$, then the state-formula $(c.\text{value} = 4) \land c.\text{ticking}$ asserts that four ticks have elapsed since the clock $c$ was started, and the clock is still ticking (i.e. the *stop* command has not been issued). The *start* and *stop* commands are defined by:

- $\text{start} \equiv \neg \text{ticking} \land (\text{value}' = 0) \land (\text{mode}' = \text{up}),$
- $\text{stop} \equiv \text{ticking} \land (\text{value}' = \text{value}) \land (\text{mode}' = \text{stopped}).$
The commands `reset_check` and `lockstep` of `CLOCK_TRANSITION`, and the commands `start`, `stop`, `count_up` and `count_down` of `CLOCK` will be defined in the sequel.

An instance of `CLOCK_TRANSITION` may issue a `reset_check` to other instances of `CLOCK_TRANSITION`, as will be explained in the sequel.
When a clock is stopped, the clock time is not reset to zero; it stays at whatever the current value is. A clock satisfies the TLD \( \text{start} \Rightarrow (\text{ticking}\forall \text{stop}) \), which asserts that once the clock is started, it ticks until it is stopped.

The \textit{count\_up} and \textit{count\_down} commands allow the clock to increment or decrement from the last stopped value, or to count from a user defined non-negative integer \( udv \):

\[
\begin{align*}
\text{count\_up} & \equiv (\text{mode}' = \text{up}) \land (\text{value}' = \text{value}) \\
\text{count\_up(udv)} & \equiv (\text{mode}' = \text{up}) \land (\text{value}' = \text{udv}) \\
\text{count\_down} & \equiv (\text{mode}' = \text{down}) \land (\text{value}' = \text{value}) \\
\text{count\_down(udv)} & \equiv (\text{mode}' = \text{down}) \land (\text{value}' = \text{udv}).
\end{align*}
\]

The \textit{lockstep} command for a clock is defined by:

\[
\text{lockstep} \equiv \begin{cases} 
    (\text{mode} = \text{stopped}) \rightarrow (\text{value}' = \text{value}) \\
    (\text{mode} = \text{up}) \rightarrow (\text{value}' = \text{value} + 1) \\
    (\text{mode} = \text{down}) \land (\text{value} > 0) \rightarrow (\text{value}' = \text{value} - 1) \\
    (\text{mode} = \text{down}) \land (\text{value} = 0) \rightarrow \text{true}
\end{cases} \quad \text{(Eq. 5)}
\]

The \textit{lockstep} command is invoked by the \textit{tick} transition to keep all clocks to keep synchronized with the global time \textit{tick} transition. If the clock is counting down, it decreases until zero; it remains at zero until it is re-started or told to count up. Using the relationship between enabling conditions and command-descriptions (Eq. 2), we see that the enabling condition \( \text{lockstep}.\text{enb} \equiv \text{true} \). This means that \textit{tick} will not become disabled by invoking lockstep.

### 2.3 Clock transitions

A clock transition is an instance of the class \text{CLOCK\_TRANSITION}. Thus, a clock transitions inherit all the features of standard fair transitions (e.g. \textit{name}, \textit{otr}, \textit{fair} etc.). In addition, clock transitions may also:

- manipulate clocks via the additional feature \( \textit{ctr} \) (clock transition relation) which is a command-description in clock values, and

- have a \textit{timer} attribute that is used to ensure that the transition is taken between its lower and upper time bounds \( \text{low} \) and \( \text{hi} \). The \textit{reset} command can be used to reset the timer to zero.

If we have a set of clock variables \( c1, c2, c3 : \text{CLOCK} \) available to a transition \( \tau : \text{CLOCK\_TRANSITION} \), then the command-description \( \tau.\text{ctr} \)

\[
\tau.\text{ctr} \equiv c1.\text{start} \land c2.\text{count\_up}(25) \land (c3.\text{value} = 10) \land (c3.\text{value}' = c3.\text{value}) \quad \text{(Eq. 6)}
\]

specifies the simultaneous pair of commands to start clock \( c1 \) and to let \( c2 \) count up, starting at 25 time units, provided \( c3 \) has reached 10 time units in the prestate. In the classchart of Fig. 4, we constrain \( \textit{ctr} \) to be a command-description in clock variables as shown in the formula above.

Fig. 4 indicates that the transition relation \( \rho \) of a clock transition is redefined. The redefined transition relation is:
The conjuncts of the above clock transition relation describe updates to the various system variables (i.e. control, data and clock variables) and transition timers:

- **otr** describes changes to control and data variables,
- **ctr** describes clock starts and stops,
- **(timer ≥ low)** ensures that the transition is taken only when it is eligible, i.e. when the lower bound has been reached (i.e. after the transition has been delayed for low time units),
- **(ε’ = name)** indicates that the transition variable ε: VARIABLE is suitably updated to reflect that the current transition is taken, and
- the last conjunct ensures that for each clock transition τ, the transition timer τ.timer is reset to zero in the next state, provided the transition is disabled in the next state (i.e. its sentry is false in the next state). Thus, if the occurrence of the current transition (which may change system variables) thereby causes other transitions to become disabled, then the timers of the other transitions must be reset to zero. This is because the timer remains non-zero and increasing in lockstep with the tick transition, only if the transition continues to be enabled. Since the quantification is universal, the current transition is included in the reset_check.

A transition has a lockstep command, that is called by the tick transition, as follows

\[
\text{lockstep} \equiv \begin{cases} 
\text{pending} & \rightarrow (\text{timer'} = \text{timer} + 1) \\
\text{eligible} \land (hi \neq \infty) & \rightarrow (\text{timer'} = \text{timer} + 1) \\
\text{urgent} & \rightarrow false \\
\text{otherwise} & \rightarrow (\text{timer'} = \text{timer}) 
\end{cases}
\]

(Eq. 8)

where a transition with its sentry true is either pending, eligible or urgent as defined by:

\[
\text{pending} \equiv \text{sentry} \land \text{timer} < \text{low} \\
\text{eligible} \equiv \text{sentry} \land \text{timer} \geq \text{low} \\
\text{urgent} \equiv \text{sentry} \land (\text{timer} = hi)
\]

(Eq. 9)

The functions pending, eligible and urgent are private features of transitions. If the transition is pending (i.e. the timer has not yet reached its lower time bound), then lockstep keeps incrementing the timer. Once the transition become eligible (i.e. the lower time bound is reached), lockstep increments the timer, provided the upper time bound is finite. Otherwise the timer is kept as is, i.e. it stops counting up when it reaches the lower bound (this keeps the range of the timer finite). Finally, when the transition becomes urgent, the enabling condition of lockstep is false. Since tick.ρ is defined in terms of lockstep, this means that tick will be disabled until urgent becomes true. Thus, for tick to be re-enabled, the transition sentry must become false (i.e. the transition must be disabled by the occurrence of some other transition).
To ensure that an urgent transition is not delayed indefinitely from being taken, we set the fair attribute of transitions to just whenever the hi attribute is finite. This is imposed by the class invariant

\[(hi \neq \infty) \rightarrow (fair = just).\]  
(Eq. 10)

In the above development, the timer bounds hi and low may be any integer valued functions consistent with the class invariant. If we choose to make them constants, then we must impose a further class invariant \((hi' = hi) \land (low' = low)\).

### 2.4 The tick transition

While a clock transition system may consist of many instances of the class `CLOCK_TRANSITION`, the class `TICK` (Fig. 4) will have precisely one instance called `tick`. The tick transition relation is redefined to be:

\[
tick.\rho \triangleq \left[ (\forall c: CLOCK\, c.lockstep) \land (\forall \tau: CLOCK\_TRANSITIONS, \tau.lockstep) \right] 
\]  
(Eq. 11)

At every tick of the global clock, the tick issues the lockstep command which updates all the clock values and transition timers.

The class invariant ensures that tick is compassionate. Thus, tick is taken infinitely often provided it is enabled infinitely often. Because tick models global time, it is important that tick actually be taken infinitely often in any computation. The progress of time cannot be delayed forever!

There are certain clock transition systems where the progress of time can become impeded forever. We would now like to characterise what these systems are. A transition \(\tau\) is called immediate if \(\tau.low = \tau.hi = 0\).

As explained at (Eq. 5), the first conjunct in (Eq. 11) which involves clocks, will not disable tick. However, the second conjunct, involving transition timers, is a source of potential disablement. This conjunct becomes false if even one transition becomes urgent (Eq. 8).

Consider a case where there no immediate transitions and \(\tau\) becomes urgent (i.e. \((\tau.timer = hi) \land \tau.sentry\)), thus disabling tick. We can then argue that tick will eventually become re-enabled, by considering the following two possibilities:

1. The state-description or guard \(\tau.sentry\) may eventually become false because some other transition (that changes the data, control or clock variables in sentry) is taken. In such a case, the reset_check conjunct of (Eq. 7) ensures that the other transition resets \(\tau.timer\) to zero. Thus, \(\tau\) thus ceases to be urgent, and the tick transition becomes re-enabled.

2. Since \(\tau\) is urgent, and any urgent transition is also just, \(\tau\) cannot remain continually urgent (and hence enabled) and not be taken. So, if some other transition as described in (1) above does not occur, then \(\tau\) itself will eventually be taken. Since \(\tau\) is not immediate \((\tau.hi > 0)\), it becomes disabled once taken, thus resetting \(\tau.timer\) to zero (Eq. 7). The transition thus ceases to be urgent \((\tau.timer \neq hi)\), and the tick transition becomes re-enabled.

We have thus shown, for non-immediate transitions, that although an urgent transition disables tick, the state of urgency will not continue forever, and tick will eventually be re-
enabled. Even if the urgent transition is immediate, problems will occur only when there are selfloops or cycles of immediate transitions as shown in (Fig. 5). A transition system is called zeno if either (a) it has an immediate self-loop transition, or (b) it has a circuit of immediate transitions (Fig. 5). An immediate transition that is not in a cycle must be self-

![FIGURE 5. Zeno transition systems](image-url)

disabling, i.e. its sentry will be false in the poststate. Hence it also looses its urgent status, and allows tick to be re-enabled. We thus have the following lemma.

**Lemma 1:** All the computations of a clocked transition system satisfy $\Box \Diamond (e = \text{tick})$, provided it is not a zeno transition system.

A zeno system cannot be implemented, as it allows an infinite number of immediate transitions be taken before a tick of the global clock. It is therefore important to avoid system descriptions that are zeno. It is possible to have a circuit of immediate transitions and yet be non-zeno provided the transitions in the cycle have guards that are not tautologies. In such cases, the system may not be zeno, as the guards disable an immediate occurrence of the relevant transitions. It is then neccessary to check that the non-zeno property $\Box \Diamond (e = \text{tick})$ holds in all computations. Fortunately, if the transition system can be reduced to a finite representation, model-checking can then be used to do this check automatically.

### 2.5 Clock transition systems are fair transition systems

A clock transition system (CTS) can be formed by declaring (a) a set of transitions each of which are instances of the class \textit{CLOCK\_TRANSITION}, and (b) a single instance of the class \textit{TICK} (Fig. 4). Each such transition has a well-defined transition relation (see (Eq. 7) and (Eq. 11)). The resulting system is a standard fair transition system that must respect the four constraints of initiality, succession, justice and compassion (Sect. 1.2).

We now provide the precise definition of a clock transition system $M$ defined by $M \equiv (M.V, M.I, M.T, M.J, M.C)$. The system variables $M.V$ is a collection of control, data and clock variables, as well as clock modes and transition timers given by

$$M.V \equiv \{v: \text{VARIABLE} | v.\text{value}\} \cup \{v: \text{CLOCK} | v.\text{mode}\} \cup \{\tau: \text{CLOCK\_TRANSITION} | \tau.\text{timer}\} \cup \{e\}$$

where $e$ is also an instance of \textit{VARIABLE} with $e.\text{type} \equiv M.T$. The initial condition is the conjunction of all the standard initilizations of control and data variables. In addition, all clock values and transition timers are initially zero, and all clock modes are initially set to stop.

The transition set is $M.T \equiv \{\tau: \text{CLOCK\_TRANSITION} | \tau\} \cup \{\text{tick}\} \cup \{\text{idle}\}$, where \textit{tick} is the sole instance of class \textit{TICK}, and \textit{idle} is an instance of a clock transition. The transition relation $\rho$ for these transitions is described in (Eq. 7) for clock transitions and in


---

The StateClock tool and clockcharts  
March 2, 1998  
13
(Eq. 11) for \( \text{tick} \). In addition, we need to add a constraint that asserts that any variable not in the output alphabet of the transition relation is preserved, i.e. the final transition relation for each of these transitions in the CTS is \( \rho \land \text{pre}(M.V - p.\text{aout}) \).

The transition relation of \( \text{idle} \) is \( \text{idle}.p \equiv (\epsilon' = \text{idle}.\text{name}) \land \text{pre}(M.V - \{\epsilon\}) \), and \((\text{idle}.\text{low} = 0) \land (\text{idle}.hi = \infty) \land (\text{idle}.\text{fair} = \text{no}) \). The transition \( \text{idle} \) is a spontaneous transition (i.e. its upper time bound \( hi \) is infinity). It is therefore never forced to occur, neither by virtue of a fairness constraint nor by virtue of a timing requirement.

The justice set is \( M.J = \{\tau.\text{CLOCK}_\text{TRANSITION}|\tau.\text{just}\} \) and the compassion set is \( M.C = \{\tau.\text{CLOCK}_\text{TRANSITION}|\tau.\text{fair}\} \). This completes the definition of the CTS \( M \).

2.6 An example of a CTS

We use \emph{clockcharts} to describe clock transition systems. A clockchart uses the statechart-like notation described in Sect. 1.3. In addition, a clockchart allows the user to specify event time bounds, and to start and stop clocks. We can use the same techniques mentioned in Sect. 1.3, to convert a clockchart into a clock transition system.

A typical \emph{event} in a clockchart looks like

\[
E[\text{low},\text{hi}]: \quad \text{guard } / \text{assign}
\]

where \( \text{low} \) and \( \text{hi} \) are the lower and upper time bounds. With each event \( E[\text{low},\text{hi}] \), we can associate a clock transition by the same name. The corresponding transition enabling condition is \((v = a) \land \text{guard} \land (\text{timer} \geq \text{low}) \) (see (Eq. 7)), and the transition \text{assign} function can change data variables as well as start and stop clocks, and get them to count up or down.

As described in Sect. 2.3, each transition has an associated \emph{timer}. When the transition \emph{sentry} defined by \emph{sentry} \( \equiv (v = a) \land \text{guard} \) (Eq. 7) becomes true, the \emph{timer} starts to count up in lock-step with \( \text{tick} \). The transition becomes eligible to be taken when \( \text{timer} \geq \text{low} \). Provided the transition \emph{sentry} continues to hold, the transition becomes urgent when \( \text{timer} = \text{hi} \). At that point, it is either disabled (by other transitions falsifying the \emph{sentry}), or its justice constraint causes it to be taken before the next occurrence of \emph{tick}.

An event \( E[\text{low},\infty] \) is called a \emph{spontaneous} event, and its corresponding transition is never forced to occur. An immediate event \( E[0,0] \) means that it must be taken immediately on becoming enabled, i.e. before the next \emph{tick}. Immediate events are useful for describing interrupts, or other quick responses within the current cycle, but care must be taken that the resulting description is non-Zeno. An event can also be declared \( E[\text{just}] \) or \( E[\text{compassion}] \), i.e. without any time bounds. Such events are converted into transitions with \((\text{low} = 0) \land (hi = \infty) \), i.e. there no transition \emph{timer} constraints, and the \emph{fair} feature is set to \emph{just} or \emph{compassion} (respectively).

Fig. 6 presents an example of a clockchart called \( \text{apm} \) that describes an arbitrarily varying pressure of some chemical reaction, as sensed by a pressure sensor, together with a description of an alarm that is generated if the pressure stays high for 3 seconds. Once the alarm is generated, it must remain high for 2 seconds. The pressure sensor updates every one tenth of a second.

The example illustrates the use of transition \emph{timers} as well as a count-up and count-down \emph{clock}. Each \emph{tick} stands for one tenth of a second. Two clocks \( c1 \) and \( c2 \) are used.
The first clock counts down starting from 30, once it is detected that the pressure is high. The second clock is a standard count-up clock. It is started when the alarm is

The body of the module consists of two parallel objects: pressure_sensor (with control variable pressure) and alarm_generator (with control variable cv). If the pressure goes high and (cv = ready), then clock c1 is set to count-down from 30, and (cv = sensed_hi) becomes true. This happens immediately because the upper time bound of the event sense_pressure_hi is zero. If at any moment during the next 30 ticks, the pressure returns to normal, the object generate_alarm returns to the cycle starting point ready. Otherwise, the pressure has been high for 30 ticks of the clock, and the alarm signal is set. At the same time, the count-up clock c2 is started, so that the alarm signal can be held high for 20 ticks of the clock. After 20 ticks, the clock is stopped, and the alarm is reset. The generate_alarm object returns to the cycle starting point ready.

The events of the pressure sensor have lower time bounds of 1 and upper time bounds of infinity. This describes a situation where the pressure cannot change any faster than one tick of the global clock, but thereafter the change can occur at any moment (or never). All the other events are immediate, i.e. their lower and upper time bounds are zero, meaning they must occur before the next tick. For brevity in the formula below, we write c instead of c.value. The module specification apm.spec consists of apm.s1 and apm.s2:

\[
\begin{align*}
\text{apm.s1} & \equiv (\text{start}_c1 \Rightarrow (c.\text{ticking}) \land c1 = 0 \land \text{alarm} \land \text{start}_c2) \\
\text{where} \quad (\text{start}_c1 & \equiv (c1 = 30 \land c1.\text{ticking}) \\
\text{start}_c2 & \equiv (c2 = 0 \land c2.\text{ticking}) \\
\text{i.e. sound the alarm provided the pressure is continuously high for 30 ticks, and} \\
\text{apm.s2} & \equiv (\text{alarm} \land \text{start}_c2) \Rightarrow (\text{alarm} \land c2.\text{ticking}) \land (c2 = 20) \quad (\text{Eq. 13}) \\
\text{i.e. sound the alarm continuously for 20 ticks.} \\
\text{The above specification treats the clock c1 as an indicator of high pressure. This is because the clock works in synchronization with the pressure as follows:} \\
\text{sanity check1: start}_c1 \Rightarrow (\neg \text{tick} \land \text{pressure} = \text{hi}) \quad \text{Since} \quad (\varepsilon = \text{go}_\text{hi}) \\
\text{sanity check2: c1.\text{ticking} \Rightarrow ((\text{pressure} = \text{hi}) \lor (\neg \text{tick}) \quad \text{Since} \quad (\varepsilon = \text{go}_\text{low}))} \\
\end{align*}
\]

The first clock c1 counts down starting from 30, once it is detected that the pressure is high. The second clock c2 is a standard count-up clock. It is started when the alarm is
generated, and stopped after 20 ticks at which point the alarm is turned off. Since the sensor
depiction of the pressure cannot vary any quicker than one tick of the clock, events
$go_{hi}[1, \infty]$ and $go_{low}[1, \infty]$ specify a lower time bound of 1 on the rate of change. This
means that they delay for 1 tick before they become eligible to be taken. Since the upper
time bound is infinity, they are never forced to be taken once they become eligible (e.g. the
pressure may stay high forever).

The informal English language specification expressed in the above paragraph can be
written more precisely by the module specification $apm.spec$, as shown by the temporal
logic description of (Eq. 12) in and (Eq. 13) in Fig. 6.

The clockchart $apm$ of Fig. 6 is a clock transition system, and hence also a fair transition
system. The module specification is a formula of standard temporal logic. We can
therefore use standard techniques to check the correctness of $apm \models apm.spec$. For the
$apm$ module of Fig. 6, the clocks stop after a finite number of increments. Hence a model-
checker can be used to check the validity of $apm.spec$.

### 3.0 The StateClock Tool

The StateClock tool is used for describing systems as clockcharts. The clockchart can
be executed using a simulation tool. The behaviour of a clockchart can also be specified
using a temporal logic description. The clockchart can be converted into a fair transition
system, which can be exported together with its specification into files that are compatible
with STeP. The STeP tool can then be used to prove that the clockchart (fair transition sys-
tem) satisfies its specification (temporal logic description).

Fig. 7 shows the StateClock view of the pressure and alarm clockchart called $apm$ that
was described in the previous section and in Fig. 6. Except for differences in notation,
StateClock implements clocks and transition timers as described in Sect. 2.0.

The features of StateClock and its relationship to STeP are shown in Fig. 8. The specifi-
cation file $apm.spec$, containing the temporal logic descriptions (Eq. 12) and (Eq. 13) of
Fig. 6 for the $apm$ clockchart, is shown in Fig. 9. The model-checker shows that all the
required properties hold for all computations of the clockchart.
The basic unit of description is a module, which has an interface, specification, body and environment (not shown). The variables are either data variables (e.g. alarm), control variables (e.g. \texttt{apm\_pressure}), and clock variables (e.g. \texttt{c1} and \texttt{c2}). The "*" after a variable name indicates it is a clock variable.

The body of the module is the same pressure and alarm clockchart \texttt{apm} that is shown in Fig. 6. To start clock \texttt{c1} counting down from 30 units of time, the action \texttt{cd(c1(30))} is used, as depicted in the \texttt{assign} part of the event \texttt{pressureIsHi\[0,0\]} of the parallel object \texttt{generate\_alarm}. Similarly, a count up action \texttt{cu(c1(30))} would mean start counting up from 30 ticks. Actions such as \texttt{start(c2)} and \texttt{stop(c2)} start clock \texttt{c2} counting up or stop it. The values of the clocks can be referred to in the event guards, and the boolean predicate \texttt{ct(c)} is an abbreviation for \texttt{c.ticking}. A count-down clock that has reached zero is still considered to be ticking, unless it is explicitly stopped.

Private variables always have the module name as a prefix, e.g. the control variable \texttt{cv} of the \texttt{generate\_alarm} object is denoted \texttt{apm\_cv}. A control-variable or clock variable may only occur with mode \texttt{out} in the interface, i.e. the environment can read the variable but not write to it. In addition, a control variable in the interface always has the module name as a prefix (e.g. \texttt{apm\_pressure}).

<table>
<thead>
<tr>
<th>Module Interface</th>
<th>Module Specification</th>
<th>Module Body</th>
<th>Private Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>i.e. variables shared with the environment with modes \texttt{in}, \texttt{out} or \texttt{shared}.</td>
<td>(clicking on this button brings up a temporal logic description)</td>
<td>(a clockchart)</td>
<td>(data, clock or control variables)</td>
</tr>
</tbody>
</table>
The StateClock tool and clockcharts

**FIGURE 8. StateClock and STeP**

- **m.stc** (StateClock image)
- **m.run** (simulation output)

**STATECLOCK**

- System as a hierarchy of modules
- View of a single module (interface, specification, body)
  - The body is a clockchart (see Fig. 7)
- **m.fts** (fair transition system)
- **m.spec** (temporal logic specification)

**STeP**

- Model-checker
- Theorem prover

Convert clockcharts to FTS

**execute system (or individual module)**
FIGURE 9. Temporal logic specification file apm.spec

% apm module --- alarm on pressure
SPEC

% Each clock c has an associated clock increment variable C_c.
% When the clock is counting up: C_c1 = 1.
% When the clock is counting down: C_c1 = -1.
% The predicate ct(C_c) asserts that clock "c" is ticking.
macro ct(i: int): bool where ct(i) = (i != 0)

% Some other macros
macro ticks: bool where tick = (Event = tick)
macro pressureIsHi: bool where pressureHi = (apm_pressure = hi)
macro pressureGoHi: bool where pressureGoHi = (Event = apm_go_hi)
macro pressureGoLow: bool where pressureGoLow = (Event = apm_go_low)
macro start_c1: bool where start_c1 = (c1 = 30 \&\& ct(C_c1))
macro start_c2: bool where start_c2 = (c2 = 0 \&\& ct(C_c2))

% Main specifications
PROPERTY sound alarm if pressure high for 30 ticks:
  start_c1 ==> ct(C_c1) Until (c1 = 0 \&\& alarm \&\& start_c2)

PROPERTY sound alarm for 20 ticks:
  alarm \&\& start_c2 ==> (alarm \&\& ct(C_c2) Until (c2 = 20)

% Some sanity checks
PROPERTY clock c1 started means pressure just went high:
  start_c1 ==> (~ticks \&\& pressureIsHi) Since pressureGoHi

PROPERTY clock c1 ticking means pressure is hi or just gone low:
  ct(C_c1) ==> pressureHi \&\& (~ticks Since pressureGoLow)

PROPERTY non-Zeno:
  []<>ticks