A Visual Toolset for the Design of Real-Time Discrete Event Systems.

Jonathan S. Ostroff

Department Of Computer Science, York University[1],

4700 Keele Street, North York Ontario, Canada, M3J 1P3.

Email: jonathan@cs.yorku.ca Tel: 416-736-2100 X77882 Fax: 416-736-5872

Abstract: StateTime is a prototype toolset that supports the design of verified real-time discrete event systems using executable visual state descriptions (the Build tool). Visual state descriptions allow the designer to browse and understand the structure of the system. A timing hierarchy of spontaneous, just and forced timed events, and a variety of computational notions such as concurrency, hierarchy, nondeterminism, process interaction and communication can be represented. The combination of model-checking (the Verify tool) and theorem proving allows for the treatment of finite and infinite state systems. The toolset is illustrated with a shutdown controller of a reactor, as taken from an actual industrial requirements document in which the system is described informally using a mixture of English descriptions, timing diagrams and pseudocode. Using StateTime, a unified precise description of the shutdown reactor is obtained, which can then be checked automatically for conformance with its requirements.

Keywords: real-time discrete event systems, verification tools, real-time temporal logic, timed transition models, visual descriptions.

1. This work is supported in part by the National Science and Engineering Council of Canada. This work is scheduled to appear in *IEEE Trans. on Control Systems Technology.*, May 1997.

Table of Contents

Figures

1.0 Introduction

Computers are increasingly used to monitor and control *reactive* safety critical systems. Real-time software in such computers controls aircraft, shuts down nuclear power reactors in emergencies, keeps telephone networks running, and monitors hospital patients. The use of computers in reactive systems offers considerable benefits, but also poses serious risks to life and the environment [35].

Reactive software is more realistically modelled by asynchronous discrete event theories than by continuous differential or synchronous difference equations. The complexity of reactive systems necessitates viewing them at different levels of abstraction. A high-level view of these systems will often involve discrete control actions, tasks, switching between modes and logical decision making. Many problems in reactive systems may be inherently discrete in nature (e.g. scheduling of control tasks).

The need for rigorous means of ensuring logical correctness of complex reactive systems, has given rise to diverse approaches by control theorists[2] on the one hand (e.g. the Ramadge-Wonham theory of discrete event systems [43,45]), and computer scientists on the other (e.g. process algebras [19], extended transition systems [9], and temporal logic [31]).

Control theorists developed various synthesis techniques for discrete event systems [5,12,13,20,28]. Most of the early work in synthesis was set in the simple framework of finite automata, with more recent extensions to real-time and infinite-string formal languages leading to the solution of some problems of realistic complexity [45]. However, actual industrial reactive systems need the representational advantages of real-time programming languages such as Ada and Occam (e.g. assignments to typed data variables and scheduling features).

In the more complex domain of typed variables and programming constructs, synthesis results are harder to come by. For example, the overall problem of program verification, including that of the synthesis of invariant assertions is undecidable [29].

In this paper we consider the TTM / RTTL framework for the design and analysis of discrete real-time reactive systems. The TTM / RTTL framework [32,33,40] consists of the following components:

- A *constructive description language* called timed transition models (TTMs) for describing reactive systems. A TTM is a guarded transition system with lower and upper bounds on the transitions that relate to the occurrence of a special transition *tick*. Concurrent real-time programs, nondeterministic timed Petri nets and diverse mechanisms for timing, synchronization and communication constructs can be converted into TTMs in a straightforward manner.

2. We focus on those theories that adapt the ideas of classical control theory to the synthesis of supervisory controllers for discrete event systems. There is another important stream associated with the performance analysis, simulation and optimization of discrete event systems (e.g. see [7,8]); this stream is not discussed in the sequel.

- A *declarative specification language* called real-time temporal logic (RTTL) for describing the requirements that a TTM should satisfy without discussing how the TTM is constructed. RTTL is a timed extension of linear temporal logic, augmented with clock and event variables.
- *Analysis techniques* for demonstrating that a TTM conforms to its specification. A proof system for theorem proving and model-checking are the main analysis techniques. Model-checking is a method of automatically verifying concurrent systems in which a finite-state model of the system (TTM) is compared with a correctness requirement (RTTL). Since time is a monotonically increasing variable, the state-space of naive timed systems is automatically infinite state. Hence, model-checking algorithms need special care to ensure that the system remains finite state.

Purpose of this paper

The purpose of this paper is to describe the *StateTime* toolset for design and analysis of real-time reactive systems within the TTM / RTTL framework. To convey how the various parts of StateTime are used as a unified tool for analysis and design, we provide a complete development example taken from the shutdown system for a nuclear plant, as taken from an actual industrial requirements document in which the system is described using a mixture of informal English language descriptions, timing diagrams and pseudocode (Section 2.0). Using StateTime, a precise visual description of the shutdown reactor is obtained, which can then be checked automatically for conformance with its requirements (Section 5.0).

StateTime has a visual description language with richer automated verification techniques for real-time systems than a commercially available visual tool. We compare StateTime to other tools in Section 6.0. The visual state descriptions allow the designer to browse and understand system structures.

Organization of the rest of this paper

Section 2 presents the informal descriptions and requirements of the reactor shutdown system in order to illustrate the type of problem StateTime is intended to be used on. Section 3 describes the TTM / RTTL framework. Section 4 presents the various features and tools of StateTime. Section 5 applies the toolset to the design and analysis of the shutdown system. In Section 6, StateTime is compared to other tools (especially Statemate). Conclusions and future work are discussed in Section 7.

Readers unfamiliar with formal logic and verification may want to skip Sections 3 and 4 on a first reading to obtain a working understanding of StateTime before looking at the underlying details.

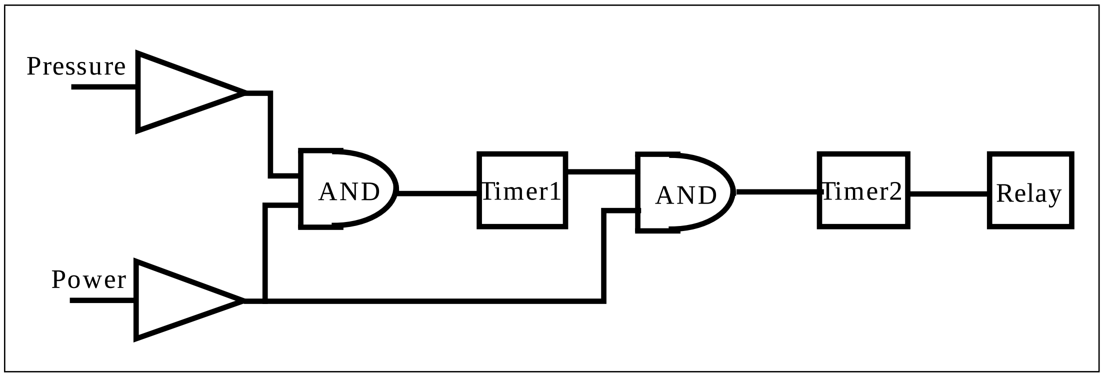
2.0 Example — nuclear reactor shutdown system

In early nuclear reactors, the shutdown systems were constructed of analog devices. The analog control had the virtue of being simple to understand but inflexible, unable to perform system checks and not always reliable. It was felt that the situation could be improved by installing computerized control with at least two independent shutdown systems, designed by different teams, each shutdown system itself having 3-version control and majority voting logic [41].

The delayed reactor trip (DRT) problem was first described by Lawford *et. al.* [25]. Lawford developed behaviour preserving transformations for timed transition models (TTMs) with which he was able to discover a flaw in the proposed design [24]. However, his theory cannot be automated as no set of transformations is complete for proving observation equivalence between the actual implementation and its abstract specification. In the sequel we will use the StateTime tool to describe the DRT and automatically check that it conforms to its requirements.

The DRT for nuclear reactors used to be implemented in hardware using timers, comparators and logic gates similar to the diagram shown in Figure 1. The

FIGURE 1. Analog implementation of the delay relay trip timing.



new DRT system is implemented on microprocessors. Digital control systems provide cost savings and flexibility over the hardware implementation. However, the question now is whether the new microprocessor based software controller satisfies the same specifications as the old hardware implementation.

The hardware version of the controller implements the following informal requirements:

> **[R1]** When the power and pressure of the reactor exceed acceptable safety limits, the comparators which feed in to the first AND gate cause Timer1 to start, which times out after 3 seconds and sends a message to one of the inputs of the second AND gate indicating that the time-out has occurred. If after this first time-out the power is still greater than its safety limit, then the relay is tripped (opened), and Timer2 starts. The relay must remain open until Timer2 times out which happens after 2 seconds.

Requirement [R1] ensures that the relay is opened and remains open for two seconds thus shutting down the nuclear reactor in a timely fashion. If the controller fails to shut down the reactor properly, then catastrophic results might follow including danger to life. Conversely, each time the reactor is improperly shut down, the utility operating the reactor loses money because it must bring additional fossil fuel generating stations on line to meet demand. The next informal requirement states:

> **[R2]** If the power reduces to an acceptable level then the relay should be closed as soon as possible (thus allowing the reactor to operate once more).

A final requirement that is *implicit* in the hardware specification, but must be *explicitly* stated for the software version is:

> **[R3]** The controller should never deadlock.

For example, if after the power and pressure have exceeded their critical values, and the computer has waited 3 seconds to check the power level again, if the power is below its critical limit, then the computer should reset and go back to monitoring its inputs (failure to do so would result in a deadlock).

In the actual DRT, there are three identical systems running in parallel with the final decision on when to shut down the reactor implemented on a majority rule basis. In this section we analyze a closed system consisting of the plant (relay, power and pressure) and a single microprocessor controller. The 3-version system can also be verified using the techniques discussed in this paper and compositional reasoning [38].

The original requirements document, taken from an actual industrial example, provided the pseudocode in Figure 2 as the specification for the controller. This

FIGURE 2. Faulty pseudocode for the computer to control the DRT

```
Every one tick of the clock Do:
If P=1 {pressure is high}
then
    If W=1 {power is high} then
        If counter Ta is reset then
            If Tb is reset then
                increment Ta
            EndIf
        Else
            If counter Tb has timed out then
                reset Tb
            Else
                increment Tb
                open Relay
            Endif
        Endif
    Else
        If counter Ta has timed out then
            open Relay
            reset Ta
            increment Tb
        Else
            increment Ta
        Endif
else
    If counter Ta is reset then
        If counter Tb is reset then
            close Relay
        Else
            If counter Tb has timed out then
                close Relay ; reset Tb
            Else
                increment Tb
                open Relay
            Endif
        Endif
    Else
        If counter Ta has timed out then
            reset Ta
        Else
            increment Ta
        Endif
```

code was to be implemented on a microprocessor with a cycle time of 100ms. The microprocessor samples the inputs (pressure and power) and passes through a block of code every 0.1 seconds. It is assumed that the input signals have been properly filtered and that the sampling rate is sufficient to ensure adequate control. In the formal model, one tick of the clock will represent 100ms. The pseudocode makes use of two integer counters *Ta* and *Tb* for the two time-outs of 30 and 20 clock ticks respectively.

3.0 The TTM/RTTL framework

3.1 Timed Transition Models (TTMs)

TTMs are timed extensions of fair transition systems of Manna and Pnueli [31]. Transitions have lower and upper time bounds that refer to the number of occurrences of a special transition *tick* in a computation of a system. A TTM *M* is defined as a 4-tuple $M = (V, I, T, J)$ as follows:

- *V*: a finite set of typed *system variables*. There are two distinguished variables that are always elements of *V*: the discrete clock time *t* and the event variable ε. The *clock time* $t \in V$ has $type(t) = \{0, 1, 2, \ldots\infty\}$. The event variable indicates what transition has just been taken; its type is the transition set (see below). There may also be *data variables* which range over data domains such as integers, rationals, lists or sets. *Object variables* (also called control variables) are used to indicate progress in the execution of the various concurrent threads or processes of the TTM. A state *s* of the TTM is a mapping that assigns to each variable $v \in V$ a value in *type(v)*. For example, if the system variables are $V = \{t, \varepsilon, b, r\}$ where *b* is a boolean and *r* is rational variable, then the state $s = \langle t{:}10, \varepsilon{:}\mathsf{tick}, b{:}\mathsf{true}, r{:}6.8\rangle$ denotes a system state after taking a clock tick at time $t = 9$. The set of all states is denoted by Σ.
- *I*: the *initial condition*. This is a satisfiable boolean valued expression in the system variables that characterizes the states at which the execution of the TTM can begin. A state *s* satisfying *I* (written $s \Vdash I$) is called an *initial state*.
- *T*: a finite *set of transitions* which includes the distinguished transitions *initial* and *tick*. Each transition $\tau \in T$ is a function $\tau \colon \Sigma \to powerset(\Sigma)$ that maps a *prestate s* in Σ to a (possibly empty) set of τ-*successor* states $\tau(s) \subseteq \Sigma$. The successor states are also called the *poststates* of τ and *s*.
- *J*: a *justice set* where $J \subset T$. Informally, the justice constraint for each transition $\tau \in J$ disallows computations in which τ is continually enabled but not taken beyond a certain point[3]. The *tick* transition is always in the justice set.

It is convenient to represent the effect of taking a transition by a *transition relation* which is a first order formula $\rho_\tau(V, V')$ that relates a *prestate s* to any of its poststates $s' \in \tau(s)$. The transition relation refers to both unprimed and primed system variables. An unprimed variable *v* is evaluated in the prestate *s* while the corre-

3. Justice (weak fairness) is defined more formally later. We may also allow a set of *compassionate* transitions (strong fairness). Compassionate transitions are not used in the examples of the sequel. The StateTime tool supports both just and compassionate transitions. See [31] for a precise definition of compassion.

sponding primed variable *v'* is evaluated in the poststate *s'*. Assuming $V = \{t, \varepsilon, b, r, y\}$, the tick transition relation is:

$$\rho_{tick}(V, V'): (t' = t + 1) \wedge (\varepsilon' = tick) \wedge same(b, r, y)$$

where $same(b, r, y) \equiv [(b = b') \wedge (r = r') \wedge (y = y')]$, i.e. the variables *b, r* and *y* remain unchanged when the tick transition is taken. The only variables that change are the global time *t* (which is incremented by one), and the event variable ε which is updated to indicate which transition has just been taken. All other variables remain the same. Any non-tick transition leaves the clock unchanged. An example of a non-tick transition relation (e.g. for transition α) is:

$$\rho_{\alpha}(V, V') \ : (r' = r + y) \wedge (r \geq y) \wedge (\varepsilon' = \alpha) \wedge same(t, y) \qquad \text{(Eq 1)}$$

The clock variable *t* remains the same when α is taken. The conjunct $(r \geq y)$ in the unprimed variables in (Eq. 1) is the *enabling condition* $enb(\alpha)$ of transition α, i.e. α can only be taken if *r* is at least as large as *y* in the prestate. The simultaneous *update function* $upd(\alpha) = \{e: \alpha, r: r + y\}$ is used to denote those variables that change when the transition is taken[4].

The enabling condition $enb(\tau)$ of transition τ can be formally defined from its transition relation by $enb(\tau) \equiv (\exists V' | \rho_{\tau}(V, V'))$. We say that transition τ is *enabled* in a state s (written: $s \Vdash enb(\tau)$) if $\tau(s) \neq \varnothing$ — otherwise τ is said to be *disabled*.

In the sequel, we often describe a transition τ by its enabling condition $enb(\tau)$ and update function $upd(\tau) = \{v_1: e_1, v_2: e_2\}$, where e_1 and e_2 are expressions in the system variables. The corresponding transition relation is given by $\rho_{\tau}(V, V')$: $enb(\tau) \wedge (v_1' = e_1) \wedge (v_2' = e_2)$.

In general, formulas such as the enabling condition and initial condition are called *state-formulas*, i.e. predicates in the unprimed system variables. State-formulas can be evaluated to true or false given a single state; hence the notation $s \Vdash f$ for a state-formula *f*. Transition relations require two states (the prestate and poststate) for their evaluation. Temporal logic formulas (see next section) require sequences of states (called computations) for their evaluation.

In addition to the enabling condition and update function associated with each transition, each non-tick transition τ also has an associated lower time bound $low(\tau)$ and upper time bound $up(\tau)$, where $0 \leq low(\tau) \leq up(\tau) \leq \infty$.

A timed transition $\tau[l, u]$ with lower time bound l ticks and upper time bound u ticks, must delay *l* ticks before being taken, but must be taken by *u* ticks of the clock, unless it is pre-empted by some other transition. In increasing order of timing stringency we have:

- A *spontaneous transition* $\tau[0, \infty]$ may occur at any point in time after becoming enabled, or it may never occur. For example, a device failure is spontaneous. In the sequel, a spontaneous transition is indicated by the fact that its upper time bound is infinity (∞).

4. The variables that are unchanged are by convention not included in the update function.

- *A just transition* must eventually occur if it is continually enabled. For example, a process that is continuously enabled to enter its critical region should eventually be allowed in. Justice is qualitative in the sense that although a just transition must occur, no finite bound on the time of occurrence is given.
- *A timed transition* such as $\tau[3, 7]$ must occur within an interval specified by a lower and an upper time bound. For example, sending a message may take between 3 and 7 ticks of the clock.

It is possible to have examples of all three types of transitions in a single TTM. When modelling a system, we can initially make all transitions spontaneous. As more timing information becomes available we can add justice constraints or tighten the bounds to provide a more precise description of the system behaviour.

In the next subsection, we provide the formal operational semantics of a TTM by describing its computations. Informally, a computation starts in an initial state. From any state of the computation, any enabled transitions is taken *in one atomic step*. The resulting interleaving of enabled transitions allows us to model concurrent processes[5]. When the transitions are taken, they update the variables according to the transition relation. The clock must tick infinitely often in any computation, and an arbitrary but finite number of (non-tick) transitions can be taken between any two ticks of the clock. The lower and upper time bounds of transitions must be respected, e.g. a lower bound of 3 requires that the transition not be taken for 3 clock ticks (even though the transition is enabled).

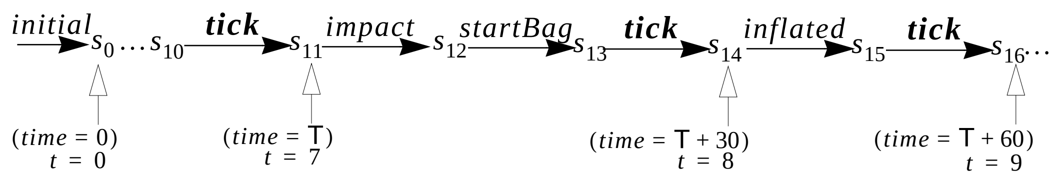
There are three sources of non-determinism in TTMs. (a) The transition relation itself may be non-deterministic. (b) Even if all transition relations are deterministic, many transitions may be enabled in the same state, and hence at any moment only one is nondeterministically taken. (c) Timing intervals introduce nondeterminism. Even a transition with an exact discrete delay (e.g. $\tau[3, 3]$) allows τ to be taken in any state after the 3rd but before the 4th tick of the clock from the moment the transition became enabled. Case (c) corresponds to asynchronous systems in which many actual events can occur sequentially between two ticks of the computer clock. In synchronous systems, all enabled transitions are taken simultaneously, and are thus based on the assumption that the environment does not interfere with the program during reactions [4].

As an asynchronous discrete example, consider a computer that checks all sensors in an automobile every 30 milliseconds (one tick = 30ms.). The air bag is one of the safety devices monitored by the computer. Impact is registered by a sensing device in the front of the car. The electronic circuit monitoring the sensor signals a solid propellant inflator to begin the chemical reaction that generates nitrogen gas to start filling the air bags. The air bag is fully inflated at 55 ms. The observed computation is provided in Figure 3.

As far as the discrete computer is concerned, the impact and start of the air bag expansion are in the same time interval [T,T+30) between the 7th and 8th clock

5. Actual systems may have *overlapped* rather than interleaved execution. However, provided an appropriate just set of transitions with the right level of atomicity is chosen, the interleaving model can accurately describe overlapped execution (see [31, p103] for further discussion).

FIGURE 3. A sample computation of the air bag system



t = discrete clock variable counting the number of ticks.
Assume there are 7 ticks between s_0 and s_{11}. The clock time t is a *flexible* variable.
time = passage of continuous time as measured on the real line.
Since there are 7 ticks of the clock between s_0 and s_{10}, with each tick = 30ms, the value of the *rigid* variable T = 210ms. The value of *time* in state s_{15} is T+55 milliseconds.

tick, but causally ordered within that interval (*startBag* is after *impact*). The computer cannot record that full inflation of the bag occurs at precisely 55ms., only that it is in the second interval [T+30,T+60]. The decision to keep time discrete is often a reasonable assumption as any computer runs to some digital clock cycle, and hence cannot discriminate to a finer timestamp than its basic cycle. The computation in Figure 3 is written more compactly as:

$$\langle initial, s_0\rangle \dots \langle tick, s_{11}\rangle \langle impact, s_{12}\rangle \langle startBag, s_{13}\rangle \langle tick, s_{14}\rangle \langle inflated, s_{15}\rangle \dots$$

If the transition information is unimportant we write the computation as $s_0 s_1 s_2 s_3 \dots$.

The event variable may be used to describe the occurrence of transitions. For example, the state-formula $(\varepsilon = impact \wedge t \geq 7)$ is true in state s_{12} but false in s_{13}. We refer to state s_i as *position i* in the computation. The initial state is always position zero.

3.2 Real-Time Temporal Logic (RTTL).

Linear time temporal logic [31] uses temporal connectives such as $\Box$ (henceforth), $\Diamond$ (eventually) and $\mathcal{U}$ (until) to represent quantitative temporal properties. The standard connectives are applied to state-formulas to obtain temporal logic formulas. Temporal logic formulas may be interpreted with respect to a computation, e.g.

- $\Diamond((\varepsilon = inflated) \wedge t = 8)$: there is some position, 8 ticks after the initial position of the computation, in which the air bag has been inflated. The computation in Figure 3 satisfies this temporal property, since the state s_{15} is a reachable state satisfying $((\varepsilon = inflated) \wedge t = 8)$.
- $(\varepsilon = impact \wedge t = 7) \rightarrow \Diamond(\varepsilon = inflated \wedge t = 7)$: if in the initial position of the computation there is an impact at time $t = 7$, then prior to any further ticks, there is a subsequent position in which the air bag is fully inflated. This property is paradoxically true in the computation of Figure 3, because the antecedent $(\varepsilon = impact \wedge t = 7)$ is false in the initial position.

- $\Box[(\varepsilon = impact \wedge t = 7) \rightarrow \Diamond(\varepsilon = inflated \wedge t = 8)]$: if in *any* position of the computation there is an impact at clock time $t = 7$, then there must be a subsequent state, one tick later, in which the air bag is fully inflated.

  We use the *entails* operator ($\Rightarrow$) for expressing response properties. The entails operator is defined as: $[p \Rightarrow \Diamond q] \equiv \Box[p \rightarrow \Diamond q]$. The property $p \Rightarrow \Diamond q$ is pronounced *p entails* eventually *q*. Implication ($p \rightarrow \Diamond q$) states only that *p* implies eventually *q* at the initial position of the computation. Entailment ($p \Rightarrow \Diamond q$) states that the implication holds at *all* positions of the computation.

- $\Box\Diamond(\varepsilon = tick)$: every position in the computation is eventually followed by a position at which the clock has just ticked, i.e. the clock ticks infinitely often. This property holds on computations that are *infinite* sequences of states.

- $p \mathcal{U} q$: *p until q*, i.e. *p* is true in all positions up to but not including *q*. The property $p \mathcal{W} q$ (*p* waiting-for *q*) can be defined as: $(p \mathcal{U} q) \vee \Box p$.

We refer the reader to [31] for a complete exposition of linear temporal logic. The logic allows for the use of flexible and rigid variables.The system variables in *V* are called *flexible variables* as they change from state to state. *Rigid variables* remain the same throughout a computation (e.g. T in Figure 3).

We will call the standard formulas of temporal logic (that have no occurrences of the clock variable *t*) *unclocked* properties. Real-time temporal logic (RTTL) includes standard temporal logic, but also allows references to the clock time *t* and additional rigid timing variables. We will call these additional timing properties *clocked* formulas.

Most proposals for *real-time* temporal logics extend the standard operators with new connectives such as $\Diamond_{\le 5} p$ (before the 6th ticks of the clock *p* holds) or use special clock variables [1]. Thus the bounded response property $p \Rightarrow \Diamond_{\le 5} q$ asserts that every occurrence of a *p*-state is followed within 5 clock ticks by a *q*-state (a state satisfying q).

In our framework there is no need to introduce new temporal operators. By using standard unextended temporal logic, we can re-use many of the tools and methods developed for untimed systems. For example, the definition of the bounded response property is [32]:

$$\Diamond_{[4,6]} q : \ [\forall t_0 | ((t = t_0) \rightarrow \Diamond(q \wedge (t_0 + 4 \le t < t_0 + 6)))] \qquad \text{(Eq 2)}$$

The response property is universally quantified over the rigid variable t_0 which has the same type as the clock variable t. The above response property states that if the clock variable has the value t_0 in the initial position of the computation, then there must be a subsequent position in which *q* is true, which occurs after the 4th but before the 7th tick of the clock from the initial position.

By (Eq. 2), $p \Rightarrow \Diamond_{\le 5} q$ is an abbreviation for $p \Rightarrow \Diamond_{[0,5]} q$, and $p \Rightarrow \Diamond_8 q$ is an abbreviation for $p \Rightarrow \Diamond_{[8,8]} q$ (i.e. every *p*-state is followed precisely 8 ticks later by a *q*-state). Additional timed operators can be defined. For example, $\Box_{<5} p$ means that *p* is true from the initial position until the position just before the 5th tick of the clock.

Given a computation $\sigma = s_0 s_1 s_2 s_3 \ldots$ and a temporal logic formula p we write $\sigma \models p$ if σ satisfies p. We write $M \models p$ when p is satisfied in all computations of the TTM M. If p is a state-formula, then $[\sigma \models p] \equiv [s_0 \Vdash p]$.

3.3 TTM semantics (computations and trajectories)

A *computation* $\langle initial, s_0\rangle, \langle \tau_1, s_1\rangle, \langle \tau_2, s_2\rangle, \ldots$ of a TTM $M = (V, I, T, J)$, where $\tau_i \in T$ for $i \geq 1$, is an infinite sequence of states satisfying the following three constraints:

1. *Initialization*: The first state of the computation satisfies the initial condition[6], i.e. $s_0 \Vdash (I \wedge (\varepsilon = initial))$.
2. *Succession*: $(\forall i | i \geq 1: s_{i+1} \in \tau_{i+1}(s_i))$, i.e. every prestate at position i must have as its successor a poststate according to the transition relation of τ_{i+1} (the transition taken at position i).

   It follows from succession that $s_i \Vdash (enb(\tau_{i+1}) \wedge (\varepsilon = \tau_i))$.
3. *Justice*: For each transition τ in the justice set, it is not the case that τ is continually enabled beyond some position in the trajectory, but taken at only finitely many positions in the trajectory.

The above three constraints are the standard description of fair transition systems [31], to which we have added the event variable ε, for descriptions that involve both state and transition information.

A *trajectory* is a computation that is further constrained by the lower and upper time bounds of transitions, defined as follows:

4. *Ticking*: The clock ticks infinitely often in the computation, i.e. $\Box\Diamond(\varepsilon = tick)$.
5. *Lower bound*: for every transition τ with lower bound $l > 0$, if τ is taken at position j of the computation, then there must exist a prior position $i \leq j$ so that $s_i(t) + l \leq s_j(t)$ and $(\forall k | i \leq k \leq j: s_k \Vdash enb(\tau) \wedge \varepsilon \neq \tau)$, i.e. τ is enabled but not taken in the states $s_i \ldots s_j$.
6. *Upper bound*: for every transition τ with upper bound $u \neq \infty$, if τ is enabled at position j of the computation, then there must exists a subsequent position $k \geq j$ with $s_j(t) + u \geq s_k(t)$, such that either τ is taken or disabled at position k.

Once a transition τ becomes enabled at position i, it begins to "mature" but cannot be taken until its lower time bound number of ticks has been taken, at which point the transition becomes "ripe" for execution. If the transition is continuously enabled during maturation, then it can be taken any time after it becomes ripe, but it must be taken or become disabled before the upper time bound number of ticks has expired. Thus, transitions "mature" together as time advances but execute separately in an interleaving manner.

6. The transition *initial* occurs once at the beginning of the computation and never again.

Unfortunately, not every TTM is guaranteed to satisfy both the ticking and the time bound requirements. If there is any *immediate* transition $\tau[0,0]$ that is a self-loop, then τ is taken an infinite number of times before a tick transition. This is called a *Zeno computation*. Any cycle whose elements are all immediate may also create Zeno computations.

The problem of Zeno computations can be avoided by disallowing self-looping immediate transitions. However, immediate transitions are useful for modelling "instantaneous" (i.e. before the clock ticks) reactions. If immediate transitions are used in a TTM *M*, then the we must check for the validity of $M \models \Box\Diamond(\varepsilon = \tau)$. Fortunately, for those systems where model-checking can be used (e.g. finite state systems), the ticking property can be verified automatically. Alternatively, suppose it is suspected that the TTM *M* may have Zeno behaviour. Then, to check that *M* satisfies the temporal logic requirement *r*, verify that $M \models (\Box\Diamond(\varepsilon = \tau) \to r)$, i.e. *r* is satisfied in every non-Zeno path.

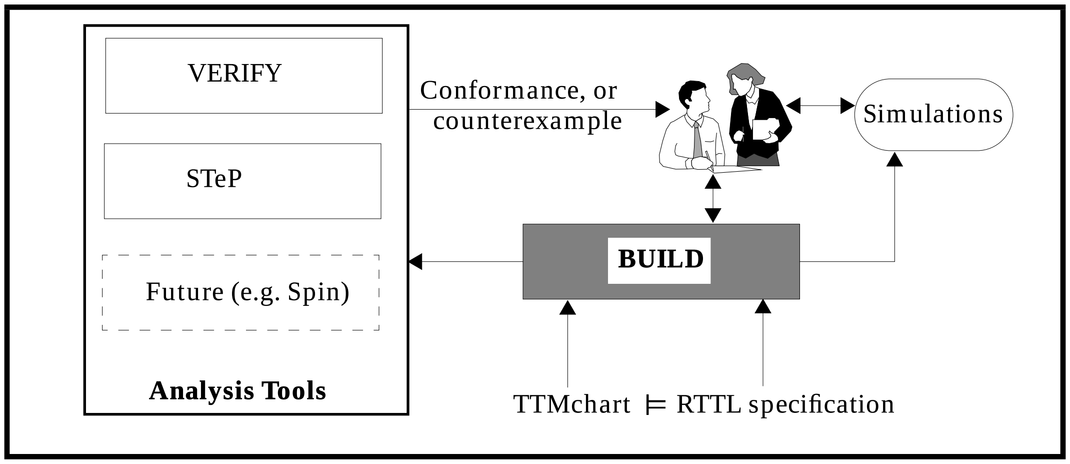
The model of TTMs presented above is expressive enough to capture most of the features specific to real time programs including delays, time-outs, preemptions and interrupts. See [32] for the use of TTMs to model the constructs of the real-time Ada-like language Conic.

4.0 Overview of the StateTime toolset

Engineers seek ways to describe and analyze their designs, whether these designs involve circuits, fluid flow in pipes, or the deformation of a beam. The purpose of the Statetime toolset is to support visual descriptions of discrete real-time reactive systems, including program components.

The toolset assists the user to (a) describe devices and systems diagrammatically, (b) execute the description so as to validate that the description is a reasonable model of the actual system, and (c) check that the description conforms to its requirements (e.g. the absence of deadlock or bounded response) using model-checking and theorem proving. The TTM/RTTL framework provides the underlying mathematical basis for describing and analyzing systems. Figure 4 shows the various parts of StateTime. The *Build* tool is a window-based front end for

FIGURE 4. StateTime checks that a TTM conforms to its RTTL specification



constructing visual models called TTMcharts. TTMcharts resemble statecharts, but with a simpler semantics and with the additional feature that transitions may have time bounds. We often use the terms TTMcharts, charts and TTMs interchangeably.

A chart is a hierarchy of objects[7], which in turn are composed of sub-objects, down to basic objects (which are control states called *activities*). The objects can be related by one of two relations:

1. "AND" which means that they act (and interact) in parallel, or
2. "XOR" which means that only one of them is active at a time (i.e. they form a transition system, so that one may have multi-level transitions from one XOR object to another).

Objects have variables which are tested and set by transitions.

An engineer can describe systems incrementally by composing objects together to form more complex objects (bottom up), or by decomposing an object which is an abstraction of a component into further sub-objects (top down). A chart can be executed (or simulated) at any point in the development cycle even before it is finally fixed. The simulation tool displays various computations or trajectories of the chart.

The Build tool automatically translates charts into a TTM. The main advantage of this translation is that the chart can be automatically analyzed for conformance with its requirements. If the chart is translated into a timed transition model M that is required to satisfy RTTL properties r_1, r_2, r_3, then *conformance* means that $M \models (r_1 \wedge r_2 \wedge r_3)$ holds. The main analysis techniques are model-checking and theorem proving. Build is written in Smalltalk, which allows it to run on most platforms (Unix, Windows and Apple Macintosh).

Model-checking supports routine automated analysis of systems but is subject to the problem of combinatorial explosion of states, i.e. the system must be reducible to a few million states (or nodes if symbolic techniques are used). Theorem proving can deal with large or infinite state systems and systems with unspecified timing parameters; but interactive guidance from the user is then required.

For simplicity, we limit the discussion in the sequel to model-checking. However, the StateTime tool supports theorem proving using constraint logic [32,34].

The *Verify* tool [33,36] was the first model-checker for a subset of real-time temporal logic. It explicitly enumerates the global state reachability graph of any finite state TTM using the algorithm reported in [33], and then checks that the chart satisfies requirements specified as formulas of real-time temporal logic. Since states are enumerated explicitly, the tool is much slower than current symbolic techniques (Verify is written in Prolog whereas modern model-checkers are written in C). In the worst case, it generates a number of states proportional to the product of the upper time bounds of all the transitions (see discussion below).

7. The term "object" describes an entity with persistent state; it is not used in the sense of objected-oriented programming.

Although there are now more efficient tools that can be used (Section 6.0), Verify has one feature not found in other tools. When it checks response properties such as $p \Rightarrow \Diamond_{[l,u]} q$, the time bounds $[l, u]$ are left unspecified so that the tool finds the values for which the property is verified. If the property is unsatisfied for any finite upper bound, then a counterexample of the failing computation is provided.

The ability to leave bounds unspecified is useful. Experience dealing with actual systems continually shows that the putative specification one imagines to be true usually does not hold, particularly when it comes to timing. It is thus vital to have a tool that tells you for which bounds the property does hold. As parameters of the chart are adjusted, the engineer can continually calculate improvements in the bounds using this feature.

A recent addition to the StateTime tool is a translator from TTMcharts to the fair transition systems of STeP [30] using the algorithm presented in [39]. This algorithm can also be used in the future to produce translators from TTMs to other third party symbolic tools such as Spin [21]. STeP is the only available tool for checking linear temporal logic with past operators, and it also has good theorem proving facilities.

The translation to STeP allows for the following:

- Any unclocked property can be checked automatically.
- Clocked properties can also be checked automatically using the distinguished tick transition as a counter. For example, to check the clocked property $p \Rightarrow \Diamond_{[1,1]} q$ we may check the equivalent unclocked property:

  $$p \Rightarrow (\varepsilon \neq tick)\mathcal{U}(\varepsilon = tick \wedge (\varepsilon \neq tick\,\mathcal{U} q)) \quad \text{(Eq 3)}$$

  Since the model-checker is exponential in the size of the formula, this approach is impractical for large time bounds.
- For clocked properties involving large time bounds we use Verify where possible. Alternatively, these properties can be rewritten in terms of additional clock variables (as in [1]).
- Another possibility for checking clocked properties is by constructing an *observer* that detects violations of the timing properties. The latter approach increases the size of the reachability graph but decreases the size of the temporal logic formula that must be checked. This is a good trade-off as the model-checking algorithm for linear temporal logic is of order

  $$|M| \cdot e^{|r|} \quad \text{(Eq 4)}$$

  where $|M|$ is the size of the reachability graph and $|r|$ is the size of the temporal logic formula to be checked. The observer approach will be illustrated in the sequel.
- Finally, STeP can model-check certain infinite state systems as it does system reduction before applying the model-checker.

When computing conformance, we use Verify for obtaining the bounds of response properties, and STeP for unclocked properties (Section 5.4). Both tools provide counterexamples which are computations in which the property fails to hold. Counterexamples together with the simulation tool are helpful for debugging the design.

For Verify and STeP, the size of the reachability graph $|M|$ in (Eq. 4) can in the worst case be of order $m \cdot u$, where m is the number of states in the untimed reachability graph (all lower bounds zero and all upper time bounds infinity), and u is the product of all the upper time bounds. This theoretical limit is usually not reached for Verify because it has a heuristic for grouping together multiple ticks of the clock [33]. The heuristic works best when all the transitions have non-zero lower time bounds, and sometimes provides insensitivity to the magnitude of the upper time bounds.

5.0 Analysis and design of the delayed reactor trip

The informal description of the DRT (Section 2.0), which is taken from an actual requirements document, uses a mixture of timing diagrams, pseudocode and English language descriptions. In order to use the StateTime tool we must proceed in a disciplined fashion, distinguishing between *descriptions* of what already exists (by using TTMs), and *requirements* that specify what should be (by using RTTL).

For the DRT, the complete system under description (*sud*) is the parallel composition of the *plant* (relay, pressure and power sensors) and the *controller* (computer with its supervisory program as represented by the pseudocode). The design method proceeds in the following order: (a) use the Build tool to describe and simulate the *plant* as a TTM object, (b) write the requirements in RTTL of how the *plant* should behave, (c) reverse engineer the proposed pseudocode *controller* into a TTM object, and (d) use the analysis tools to prove that the chart of the system under design (plant and controller) conforms to the requirements.

The order in which the design method proceeds can be justified by the following considerations. The plant is an already existing entity. Its behaviour is best described constructively using TTMs. Our main concern is that the plant behave safely and reliably no matter what controller is used. Hence, it is important to write the requirements with respect to the entities of the plant in an implementation free manner (using RTTL), before designing the controller. We can then try out the proposed pseudocode as a candidate controller, but we are free to select any controller that will meet the requirements. Since the proposed controller is to be implemented on a computer, it is also constructively described as a TTM.

Before following the above design method step by step, we first give an overview of the complete system so that the reader will have an appreciation for the end product of the method. The chart for the system under design (Figure 5) is the AND-composition of the plant, controller and an “observer”, i.e.

$$sud = plant \,\|\, control \,\|\, obs \qquad \text{(Eq 5)}$$

The observer (*obs*) watches the system for deficient behaviour without interfering with its operation (its use will be explained in Section 5.4 which deals with conformance testing). The relationship between the variables of the plant and controller are shown in Figure 6.

FIGURE 5. The TTMchart of the system under design (edit view)

sud = plant || control || obs
The object variable of *control* is F_1 where $type(F_1) = \{normal\_full, fail\}$. See Figure 7 for further details of the plant. The internal structure of *normal_full* in the control chart is described in more detail in Figure 12.

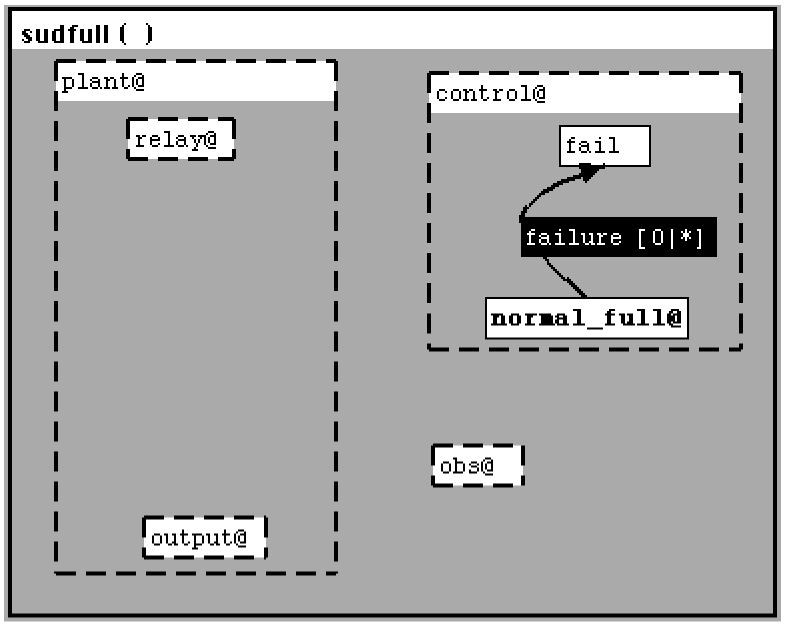
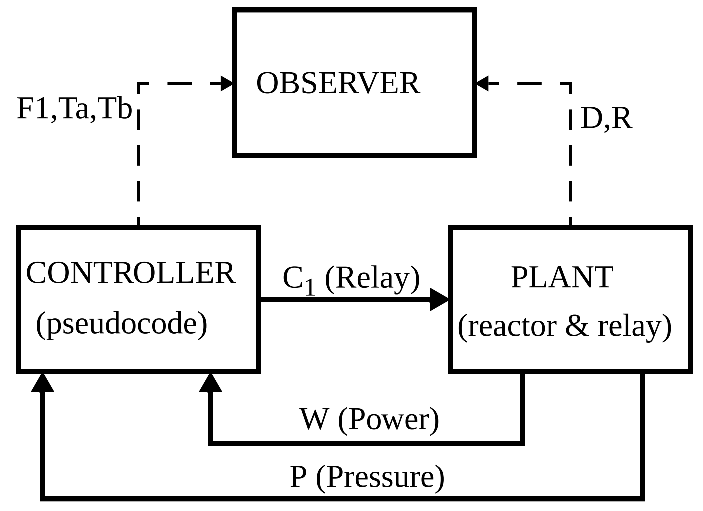


FIGURE 6. Connection diagram for the system under design



5.1 Using the Build tool to describe the DRT plant

The *plant* object (Figure 7) is itself AND-composed of various sub-objects, i.e. $plant = relay \| output$. The dashed lines in the edit view of the plant indicate that the *relay* object runs concurrently with the *output* object (AND-composition). The *relay* object can be in one of two control states closed or open called *activities*. Activities are the lowest level object (they have no internal structure). The *output* object

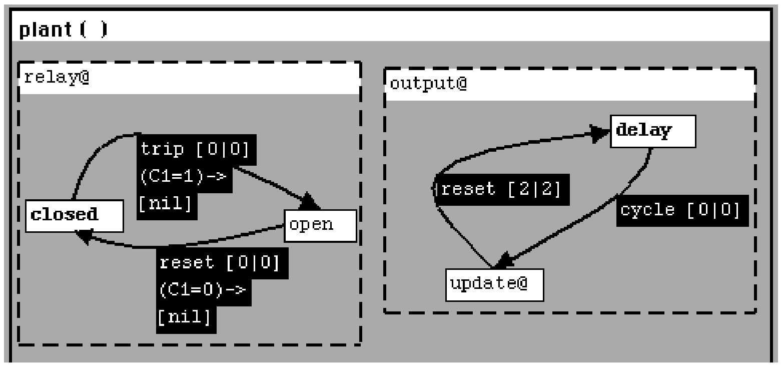
FIGURE 7. TTMchart description of the DRT plant

plant(C1;R,W,P,D)=relay(C1;R) || output(;D,P,W)

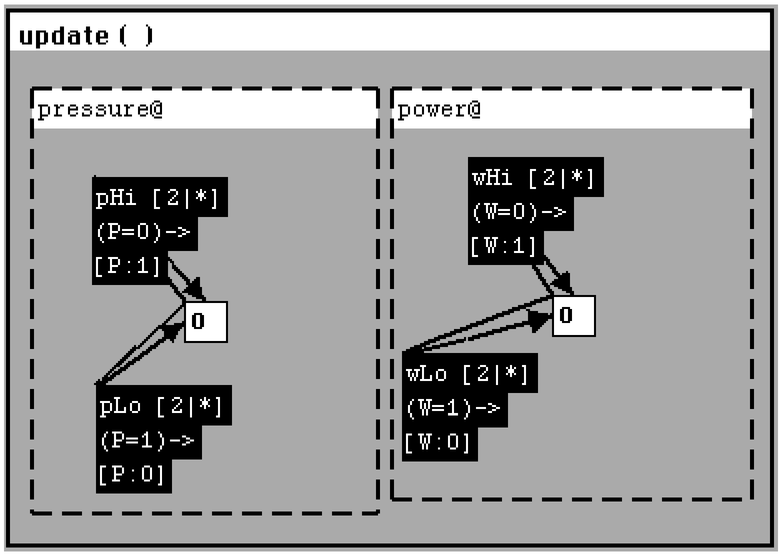
where R=relay position, C1=command to move relay, D=object variable of output, P=pressure and W=power.

The relay is opened (closed) immediately upon receiving the C1 command to do so. When D=delay, the last sensor updates have just been made and there are two ticks to the next change.

type(R)={open, closed}, type(D)={delay,update}, and type(C1)=type(P)=type(W) ={0,1}



Zooming in to update@ produces the view below (P and W are updated every two ticks).



is XOR-decomposed into sub-objects *delay* and *update@*. The *update* object has further internal structure as indicated by the "@" suffix.

A structured object m, which is XOR-decomposed into sub-objects given by $m_0, m_1, \ldots m_n$, must have an associated *object variable*, and one of the sub-objects must be designated as the *default*. The default of m indicates where it begins executing when first entered, unless otherwise specified. For example, the *relay* object variable is R where type(R) = {closed, open}. The activity **closed** is the default (defaults are shown in bold in edit views). Hence initially $(R = closed)$ is true.

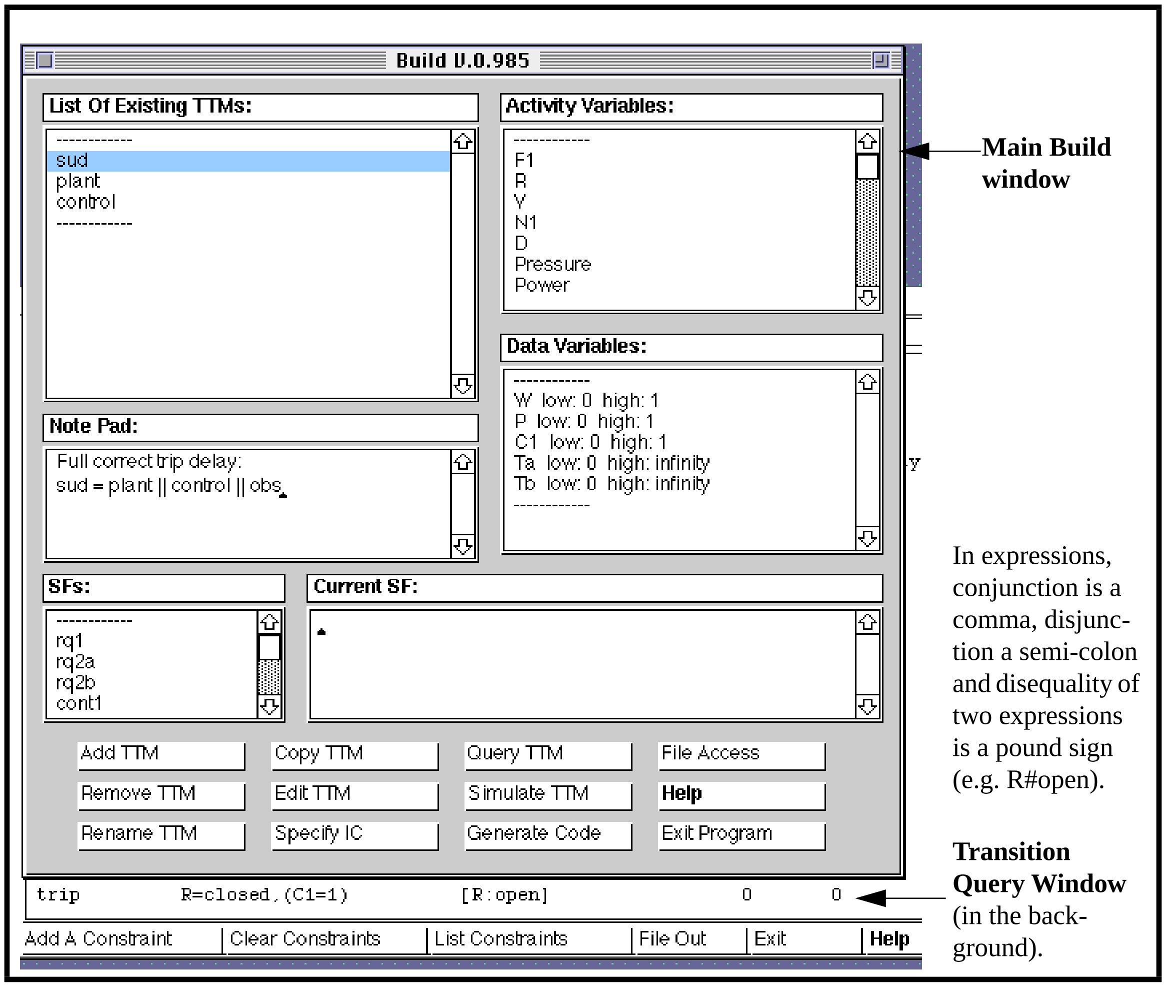
The transitions in chart objects are called *events*, e.g. the events of the *relay* object are trip and reset. The events may test and set variables such as the input data variable C1 where type(C1) = {0,1}. The *guard* of the trip event is (C1=1); if the environment sets C1 the relay will trip thus initiating the shutdown process.

The Build tool translates each event of the various objects into transitions. For example, the transition relation associated with the trip event in *relay* is:

$$\rho_{trip}\colon (R = closed) \wedge (C1 = 1) \wedge (R' = open) \wedge (\varepsilon' = trip)$$

The main window of the Build tool (Figure 8) has a Query TTM button which

FIGURE 8. Main window of the Build tool



Build V.0.985

List Of Existing TTMs:
```
------------
sud
plant
control
------------
```

Activity Variables:
```
------------
F1
R
V
N1
D
Pressure
Power
```

Data Variables:
```
------------
W  low: 0  high: 1
P  low: 0  high: 1
C1  low: 0  high: 1
Ta  low: 0  high: infinity
Tb  low: 0  high: infinity
------------
```

Note Pad:
```
Full correct trip delay:
sud = plant || control || obs
```

SFs:
```
------------
rq1
rq2a
rq2b
cont1
```

Current SF:

```
trip          R=closed,(C1=1)         [R:open]              0        0
```

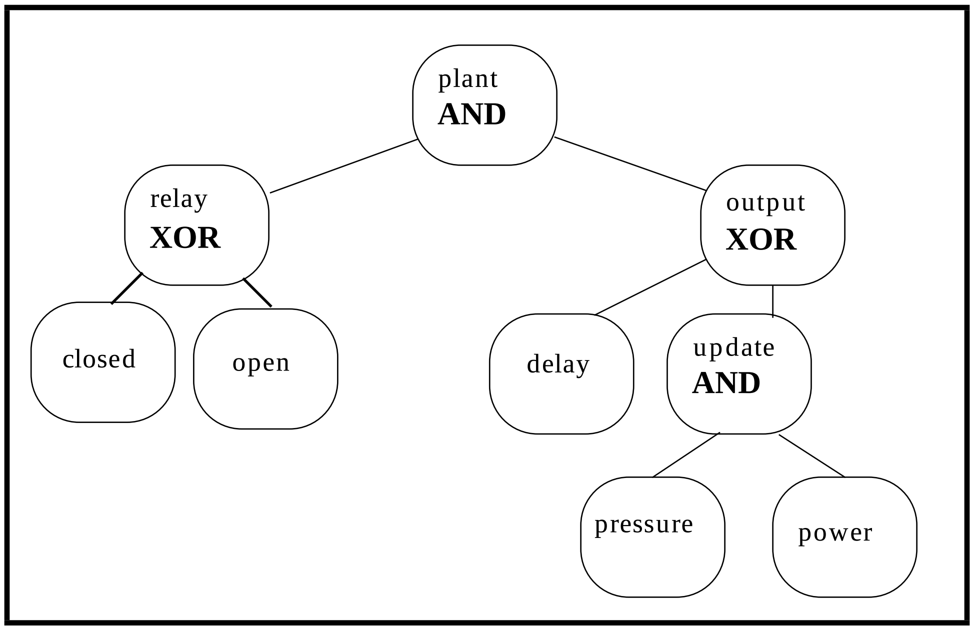
In expressions, conjunction is a comma, disjunction a semi-colon and disequality of two expressions is a pound sign (e.g. R#open).

displays a list of the transitions in update format. The query window (behind the main window) shows that the enabling condition of trip is (R=closed, C1=1) and the lower and upper time bounds are zero (i.e. once the trip transition is enabled it is taken before the next tick of the clock).

The activities (lowest level objects) can themselves be given structure, in which case they also become sub-objects. For example, the *output* object is the XOR-composition of activity delay and sub-object *update@* (Figure 7). The "@" symbol at the end of an object name indicates that it has internal structure as shown in the second picture of Figure 7. The *update* object describes how the power (W) and pressure (P) sensor values may be updated every two ticks of the clock (thus capturing the assumption that signals are filtered). For example, the spontaneous transition pHi[2,∞] may set the power high after 2 clock ticks, after which the timed transition reset[2,2] forces the object out of the update mode.

The hierarchy for the plant objects is shown in Figure 9. The user can navigate

FIGURE 9. Hierarchy of plant objects



up and down the hierarchy from any level. At any point in the development, previously created charts can be imported as sub-objects, or existing objects can be refined by zooming in. There is no restriction on the number of levels or the type (AND or XOR) of sub-object that can be inserted at each level. The main window (Figure 8) can be used to enter initial conditions and to simulate the chart.

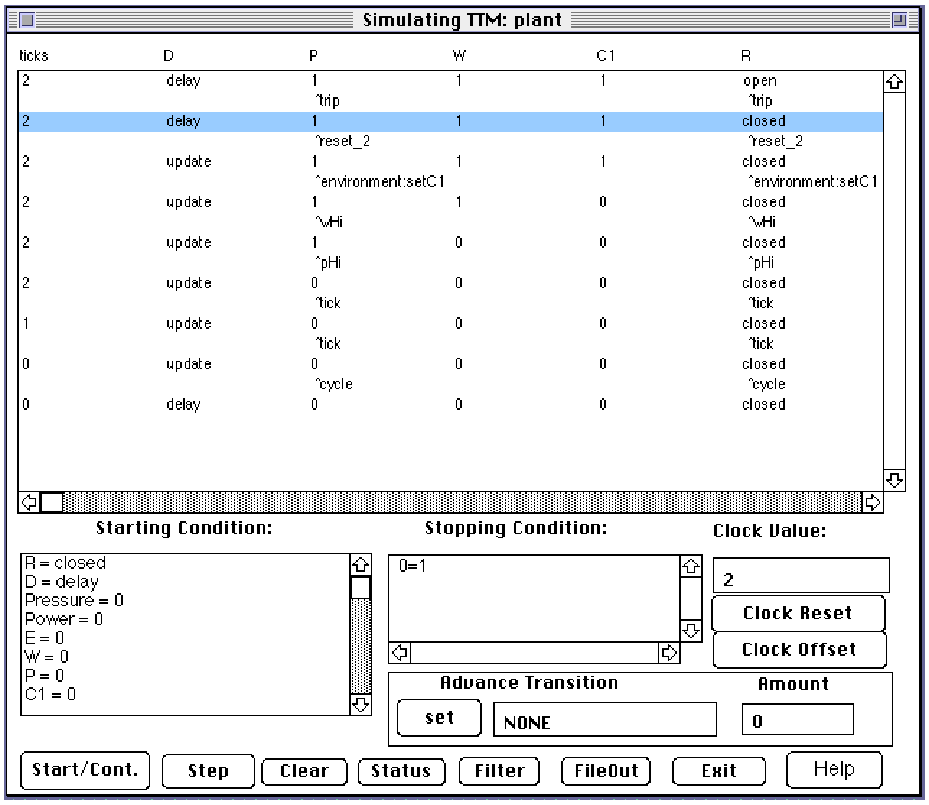
Figure 10 shows a computation of the plant in a simulation window. The computation starts with the cycle event which puts the plant into update mode. The last transition taken is trip which opens the relay (top of the screen). In between these two transitions, the clock ticks twice, the power and pressure go high and the trip transition is enabled and finally taken. For the computation shown in the figure, the environment sets the control value *C1* to open the relay at the appropriate time. However, there are also plant trajectories where the relay is not opened when it should be. For this reason, a controller is needed.

At certain points in the simulation, user input is required. For example, after two ticks of the clock, the user can nondeterministically choose whether to let the power or pressure go high, or whether to advance the clock.

Each time a new element is added to or deleted from the chart, the Build tool checks the syntactic correctness of the element. For example, data variables cannot be deleted if they are in use, the elements of types must be distinct, and expressions such as guards or updates must be correct. Events in a chart cannot have both their source and destination in parallel (AND-composed) objects.

FIGURE 10. Simulating the plant produces a computation

Initial state is at the bottom of the window. Last state is at the top.



Simulating TTM: plant

```
ticks   D        P                      W    C1   R
2       delay    1                      1    1    open
                 ^trip                                ^trip
2       delay    1                      1    1    closed
                 ^reset_2                             ^reset_2
2       update   1                      1    1    closed
                 ^environment:setC1                   ^environment:setC1
2       update   1                      1    0    closed
                 ^wHi                                 ^wHi
2       update   1                      0    0    closed
                 ^pHi                                 ^pHi
2       update   0                      0    0    closed
                 ^tick                                ^tick
1       update   0                      0    0    closed
                 ^tick                                ^tick
0       update   0                      0    0    closed
                 ^cycle                               ^cycle
0       delay    0                      0    0    closed
```

Starting Condition:

```
R = closed
D = delay
Pressure = 0
Power = 0
E = 0
W = 0
P = 0
C1 = 0
```

Stopping Condition: 0=1

Clock Value: 2

Advance Transition: NONE

Amount: 0

5.2 The DRT requirements

The informal requirements for the DRT (Section 2.0) must now be made precise. The first requirement may be written in real-time temporal logic (RTTL) as:

$$\mathbf{R1}\text{: } (bothHi \wedge \Diamond_{30} powerHi) \Rightarrow \Diamond_{[30,\,32]} \Box_{<20} (R = open) \qquad \text{(Eq. 6)}$$

where the precise definitions of $bothHi$, $powerHi$ are given below (Eq. 7). The above formula asserts that whenever a critical high is sensed, and 30 clock ticks later the power is still high, then the relay is opened within 32 ticks from the critical state and remains open until the 20th tick.

The actual system has a 3-microprocessor majority voting controller. For such systems we need to be able to express properties such as: "if at most one controller fails, the system will still satisfy its requirements", i.e. the above requirement becomes $(\Box\neg fail) \rightarrow \mathbf{R1}$. In the sequel we discuss a single microprocessor controller[8], and refer the reader to the conference paper [38] on how the majority voting system can also be verified using compositional techniques and StateTime.

How should the predicates $bothHi$ and $powerHi$ be defined? The intuitively obvious definition $bothHi \overset{\text{def}}{=} (P = 1 \wedge W = 1)$ is not realizable by practical control-

8. The failure transition (see Figure 5) can be ignored for the single controller case as it is only needed for defining *fail* in the majority voting system.

lers because they cannot respond to instantaneous changes in the sensor values (the microprocessor described in Section 2.0 responds after one clock tick).

After an update, the *output* object (with object variable D) resides in activity delay. The clause $(D = delay)$ can therefore be inserted as a conjunct of *bothHi*, thus indicating that the antecedent of (Eq. 6) is measured from a state from which the power and pressure remain constant for two clock ticks[9] (see Figure 7).

The controller cannot respond to a critical high if it is in the middle of counting its timeouts — thus the controller initial condition $init(control)$ must also be a conjunct of *bothHi*. The correct definitions for the components of (Eq. 6) are thus given by:

$$\begin{aligned} bothHi &\overset{\text{def}}{=} init(control) \wedge R = closed \wedge D = delay \wedge (P = 1 \wedge W = 1) \\ PowerHi &\overset{\text{def}}{=} (D = delay \wedge W = 1) \end{aligned} \qquad \text{(Eq. 7)}$$

The state-formula $init(control)$ cannot be fully specified until a proposed controller is suggested. For the controller of the next subsection, it is:

$$init(control) \overset{\text{def}}{=} (F_1 = normal\_full \wedge T_a = 0 \wedge T_b = 0) \qquad \text{(Eq 8)}$$

Although the initialization of the controller cannot be fully specified until the controller is developed, we can nevertheless describe a “sanity check” that the state-formula $init(control)$ must satisfy:

$$\textbf{[R3]}: \neg init(control) \Rightarrow \Diamond_{\leq 52} init(control) \qquad \text{(Eq. 9)}$$

[R3] requires that any controller must not deadlock under normal operation, i.e. must always return to its initial state after both timers have finished their count.

The second informal requirement for the DRT can be specified as:

$$\textbf{[R2]}: powerLo \Rightarrow \Diamond_{\leq 2}(R = closed) \qquad \text{(Eq. 10)}$$

where $powerLo \overset{\text{def}}{=} (D = 0 \wedge W = 0) \wedge init(control)$.

5.3 The DRT controller

In the previous subsections we showed how the plant can be described using TTMs and how the requirements can be specified in RTTL. The next phase involves designing the controller so that the system will conform to its requirements. In the case of the DRT, a candidate for the controller is already provided by the pseudocode (Figure 2), which is a loop that is executed every one tick of the clock. The pseudocode makes use of two integer counters *Ta* and *Tb* for the two time-outs of 30 and 20 clock ticks respectively.

When the original pseudocode (Figure 2) was translated into a TTM object, and composed together with the plant object, the Verify tool found that requirement [R1] failed to hold. On looking at the failing computation, it became obvious that the problem was due to the fact that the transition that opens the relay after counter *Ta* has timed out has an unnecessary dependence on the pressure. The sit-

9. Alternatively, we can write $\Box_{<2}(P = 1 \wedge W = 1)$ to indicate that the pressure and power must remain high for at least two clock ticks.

uation can be remedied by moving the “P=1 {pressure is high}” statement in the second line (Figure 2) to the fifth line (Figure 11).

FIGURE 11. Corrected pseudocode for the computer to control the DRT

```
Every one tick of the clock Do:
If W=1 {power is high}
then
    If counter Ta is reset then
        If counter Tb is reset then
            If P=1 {pressure is high} then
                increment Ta {Transition : $\mu$1}
            EndIf
        Else
            If counter Tb has timed out then
                reset Tb {Transition : $\gamma$}
            Else
                increment Tb
                open Relay {Transition:  $\mu$2}
            Endif
        Endif
    Else
        If counter Ta has timed out then
            open Relay {Transition: $\alpha$}
            reset Ta
            increment Tb
        Else
            increment Ta {Transition : $\mu$1}
        Endif
else
    If counter Ta is reset then
        If counter Tb is reset then
            close Relay {Transition : ß}
        Else
            If counter Tb has timed out then
                reset Tb {Transition : p2}
            Else
                increment Tb
                open Relay {Transition:$\mu$2}
            Endif
        Endif
    Else
        If counter Ta has timed out then
            reset Ta    {Transition : p1}
        Else
            increment Ta {Transition : $\mu$1}
        Endif
```

The TTM object corresponding to the corrected code is shown in Figure 12. Each time the microprocessor passes through the code it performs a group of operations (such as incrementing or resetting counters or opening and closing the relay). These operations define the update function of transitions. The lower and upper time bounds of each transition is 1, as the transition (if enabled) is performed exactly every 0.1 seconds.

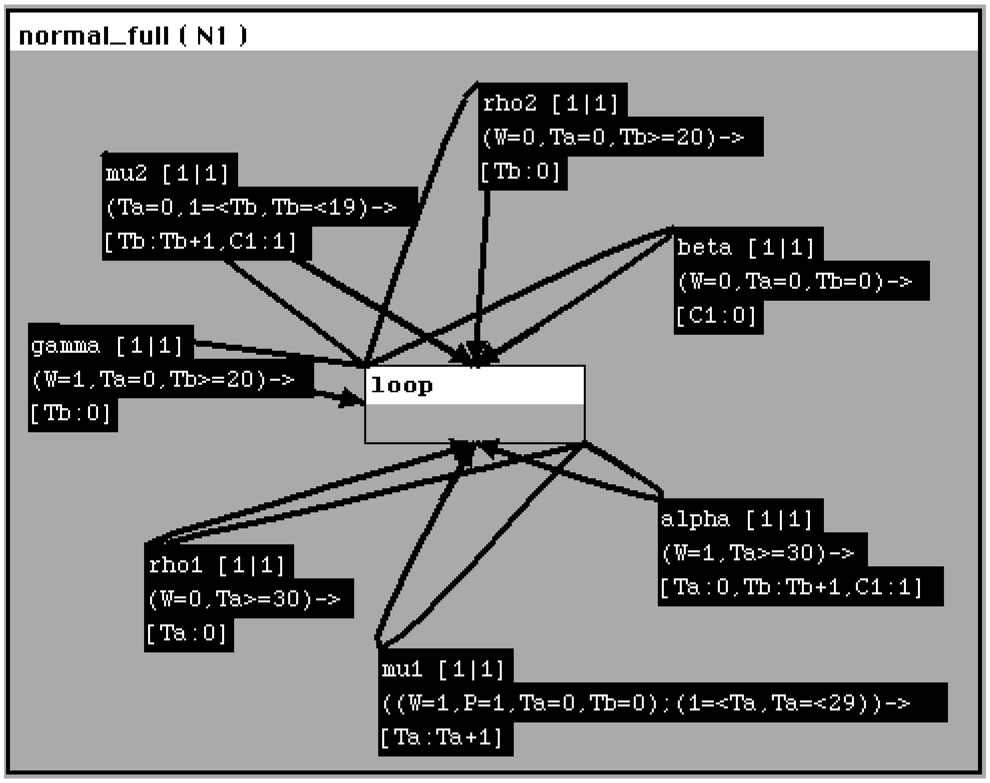
The comments in the pseudocode (Figure 11) indicate how statements of the pseudocode are translated into the transitions of the TTM (Figure 12). The enabling condition for the transitions can be computed by taking the conjunction of the conditions specified by the relevant “If” statements. For example, the transition μ_2, which is responsible for incrementing the counter T_b and opening the relay, has the update function $[T_b{:}T_b + 1, C_1{:}1]$. Its enabling condition is computed as the disjunction of its occurrence in the *if-then* part of the main loop as well as its occurrence in the *else* part, i.e.

$$\begin{aligned} enb(\mu_2) &\equiv (W = 1 \wedge T_a = 0 \wedge 1 \le T_b \le 19) \vee (W = 0 \wedge T_a = 0 \wedge 1 \le T_b \le 19) \\ &\equiv (T_a = 0 \wedge 1 \le T_b \le 19) \end{aligned}$$

The enabling condition can be trivially simplified to $enb(\mu_2) \equiv (T_a = 0 \wedge 1 \le T_b \le 19)$ because $type(W) = \{0, 1\}$. The enabling condition of μ_2 is very different for the faulty code (Figure 2) and shows an unnecessary dependence on the pressure.

FIGURE 12. Result of transforming the controller pseudocode into a TTM

Provided the controller is in its normal operating mode ($F_1 = normal\_full$) one of the self-looping events executes precisely every one tick of the clock.



5.4 Checking conformance

The main window of the Build tool (Figure 8) has a Generate Code button that can be used to translate the chart of the system under design into transition systems (as ASCII text files) that can be checked by the Verify and STeP tools. Both Verify and SteP take the text files directly as input.

The Verify tool found that the original pseudocode (Figure 2) did not meet the first requirement [R1]. The resulting counterexample was used to correct the code (Figure 11). The Verify tool was then used to show conformance of the corrected code, i.e. all three requirements [R1, R2 and R3] were shown to be valid. Using only the Verify tool the conformance check takes 35 minutes (Table 1).

Using a combination of Verify [R2 and R3] and STeP [R1], conformance of all three properties can be checked in under 4.4 minutes (Table 1). STeP cannot directly check the response properties [R2 and R3], but it was significantly faster than Verify for unclocked requirement [R1] where an observer was used (2.2 minutes versus 32.9 minutes for Verify). We discuss below in more detail how the verification was done.

The requirements [R2 and R3] were checked directly by Verify, without the need of an observer. The requirement [R2], given by $powerLo \Rightarrow \Diamond_{[0,2]}(R = closed)$, can be submitted to Verify without having to specify the bounds. Verify returns the bounds [0,1]; hence [R2] is valid because its bounds of [0,2] are more permis-

TABLE 1. Times for the model-checking tools Verify and STeP to check conformance (using a Sun Ultra1/160MB workstation)

Requirements	Verify tool		STeP tool
[R2] and [R3]	Generate graph (6943 reachable states)	1.58 min.	Not Applicable. (Does not do most permissive bound calculations.)
	[R2] check	.16 min.	
	[R3] check	.45 min.	
	Total Time for [R2,R3]	**2.19 min.**	
[R1] (with an observer and a fail transition)	Generate graph (48,489 states)	26.36 min.	
	[R1] check	6.49 min.	62,550 states in:
	Total for [R1] check	**32.85 min.**	**2.2 min**
Total time for Verify tool to check conformance (of all requirements R1, R2 and R3)		**35.05 min**	
Total time for Verify + STeP to check conformance			**4.39 min**

sive. In fact, the more stringent property given by $powerLo \Rightarrow \Diamond_{\leq 1}(R = closed)$ is also valid. The output of the Verify tool for [R2] is shown in Figure 13.

FIGURE 13. Abbreviated output from the Verify tool for requirement [R2]

```
% Checking requirement [R2]: powerLo ==> <>relayClosed
% powerLo: Ta=0,Tb=0,D=0,W=0
% relayClosed: R=closed
| ?- rtrfRG2(sud, powerLo,relayClosed, L, H).
reading in reachability graph ...
sending results to sud.output file...
L = 0,
H = 1
%... checked in .16 minutes
```

The ability of Verify to return the bounds was especially useful in the case of response property [R3] which has an upper time bound of 52. Initially, this requirement was written with an upper time bound of 50 (i.e. $max(T_a + T_b)$). However, Verify returned the bounds [0,52] indicating that the more stringent upper time bound could not be satisfied on certain computations. There are two possibilities when a requirement fails to hold:

(a) Either, the controller is incorrect,

(b) or, the requirement must be reformulated (in the case of R3 by adding an additional two ticks to the upper time bound).

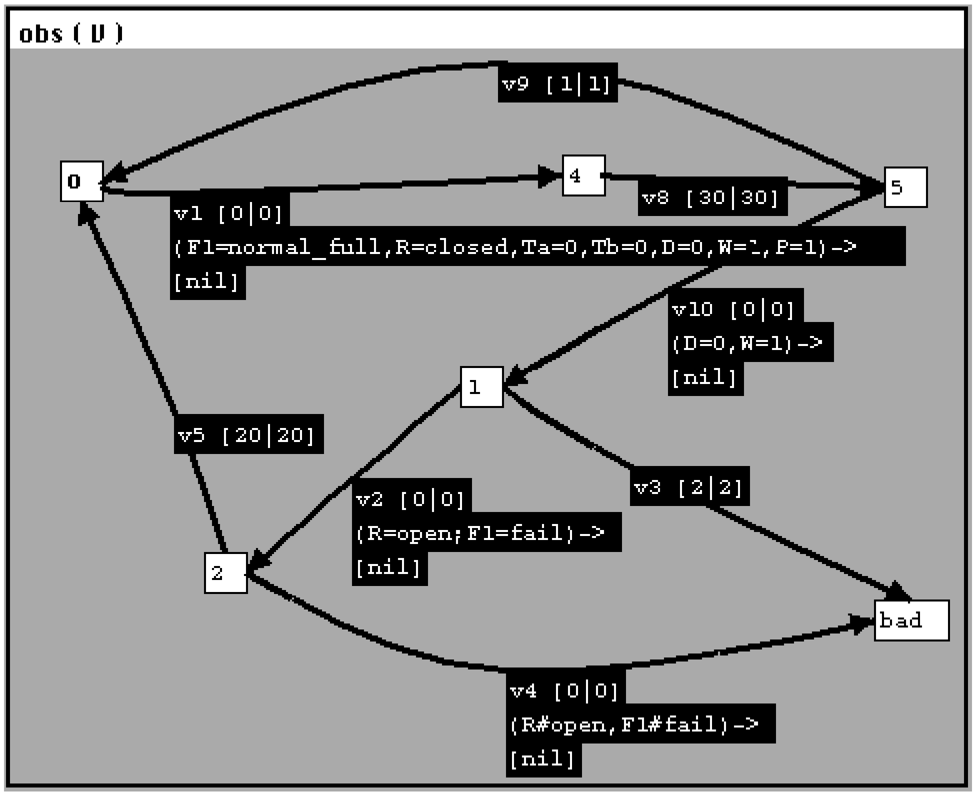
In this case, we reformulated [R3] as per case (b). At the 30 tick time-out point, the relay is commanded to open by transition α if the power is high. Because power and pressure are filtered, the power update may be delayed up to two ticks, i.e. the update may occur at any time in the interval [30, 32] of the controller cycle. Thus there may be a delay of up to 2 clock ticks before the second time-out T_b begins. Hence, the controller may delay up to an additional 2 ticks beyond 50 before returning to its initial state.

The requirement [R1] (see (Eq. 6)) is a clocked property that Verify is unable to check directly. One possibility is to use the STeP tool as explained in (Eq. 3). However, the corresponding unclocked property is impractical because of the need to count up to 52 ticks of the clock. We therefore use an observer instead.

The TTMchart *obs* (Figure 14) moves to a "bad" state when it observes violations of [R1]. The validity of [R1] can then be demonstrated by proving the validity of the unclocked invariance property $\Box(V \neq bad)$. The observer merely watches the system without interfering with its operation (hence its events have no assignments to system variables in its transformation functions). The invariance $\Box(V \neq bad)$ can be checked either by Verify or STeP. As shown in Table 1, STeP checks this kind of unclocked property much more efficiently. STeP is also able to prove the property $\Box(V \neq bad)$ using the theorem prover.

FIGURE 14. Observer to check requirement R1

The object variable of the observer is V, where type(V)={0,1,2,4,5,bad}. When *V=bad*, requirement 1 is false, i.e. (Eq. 6) is false. The events *v1* and *v10* detect when the antecedents *bothHi* and *PowerHi* of (Eq. 6) are true. At *V=5*, the observer waits 1 tick before resetting (via event *v9*) as *powerHi* must occur between the 30th and 31st tick of the clock. The events *v3* and *v4* trigger if the relay fails to open within two ticks or fails to stay open for 20 ticks of the clock.



The use of an observer increases the number of reachable states that must be generated. However, the resulting invariance property $\Box(V \neq bad)$ involves fewer logical connectives, and hence can be checked more efficiently than more complex formulas with more connectives (Eq. 4).

The approach of building a model (the TTMchart), stating the global requirements (in RTTL), and then checking for conformance, rarely proceeds in a smooth straight line. Usually, the initial modelling attempts are either wrong or fail to capture pertinent behaviour. Once an appropriate model is obtained, the initially stated requirements can either be wrong or incomplete. In fact, there is no formal method that can close the gap between the model and the actual system. At best, we can validate the model to some extent by simulation and putative challenges.

RTTL specifications are incremental. If after developing a set of requirements, we suddenly realize that the resulting specification is incomplete, the situation can be rectified by adding the missing property to the requirements as an additional conjunct, without having to recheck the other requirements. For example, the three *sud* requirements [R1, R2 and R3] are incomplete. An additional property that must be satisfied is: “the relay should not open unnecessarily” — which is given by the “waiting-for” property

$$\textbf{[R4]:}\ (R = closed \wedge W = 0) \Rightarrow (R = closed)\mathcal{W}(W = 1 \wedge P = 1) \qquad \text{(Eq. 11)}$$

The above property was submitted to the verifier and found to be valid. This property can be directly checked either by Verify or STeP without the need for an observer.

A useful validation method involves posing a “challenge” to the system with putative theorems. The fact that the relay should not open unnecessarily (Eq. 11) is one such putative theorem. In the beginning phases, most putative theorems will fail to be proven. The verifier then returns information, such as the failing computation, which is useful for debugging and correcting the problem.

Consider the complete system of Figure 5 which includes the observer and the failure transition[10]. If an unclocked version of the third requirement given by $\neg init(control) \Rightarrow \Diamond init(control)$ is submitted to the verifier, then the counterexample of Figure 15 is produced, which shows that the failure transition leads to a recursive loop in which the goal $init(control)$ is never reached. If the putative theorem $\Box(\varepsilon \neq failure) \rightarrow [\neg init(control) \Rightarrow \Diamond init(control)]$ is submitted, then the verifier responds that the property is valid. It is often useful to use the Build simulation facility in conjunction with the counterexample facility to debug the system.

Simulation was in fact performed regularly as the model of the DRT was developed, and played an important part in developing the model. For example, the plant update function was incorrectly designed in the first approximation so that only one of (but not both) power and pressure could change every two clock ticks. This modelling error was quickly revealed in early simulations.

6.0 Comparison of StateTime to other tools

One motivation for constructing the StateTime tool was that statechart-like visual languages are useful for hierarchical, concurrent and nondeterministic descriptions of timed reactive systems. An industrial strength tool called Statemate [15] is available for statecharts [14]. The tool described in this paper extends

10. Until now, we ignored the failure transition, as we dealt with the case of a single microprocessor controller.

FIGURE 15. Abbreviated view of a counterexample

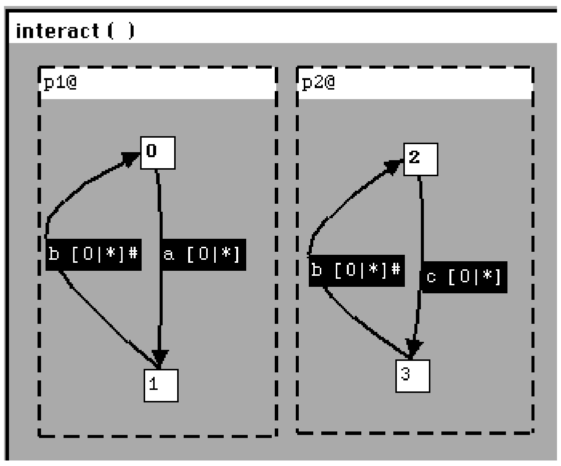
```
PATH COUNTEREXAMPLE for !init(control) ==> <>(init(control)):
trans: INITIAL
   state: [F,0,1,0,0,0,0,0,1,0,0,5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
trans: CYCLE
   state: [T,0,0,0,0,0,0,0,1,0,0,5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
trans: RESET
   state: [F,0,0,0,0,0,0,0,1,0,0,5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
trans: TICK
   state: [F,0,0,0,0,0,0,0,1,0,0,5,0,0,1,0,0,0,1,0,1,0,0,0,0,0,0,1,0]
trans: BETA
   state: [T,0,0,0,0,0,0,0,1,0,0,5,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,1,0] ...
trans: RESET1
   state: [T,0,1,0,0,0,0,0,1,0,0,5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,0]
trans: RESET
   state: [F,0,1,0,0,0,0,0,1,0,0,5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
trans: FAILURE
   state: [T,0,1,1,0,0,0,0,1,0,0,5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
LOOP:
trans: RESET
   state: [F,0,1,1,0,0,0,0,1,0,0,5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
trans: CYCLE ....
```

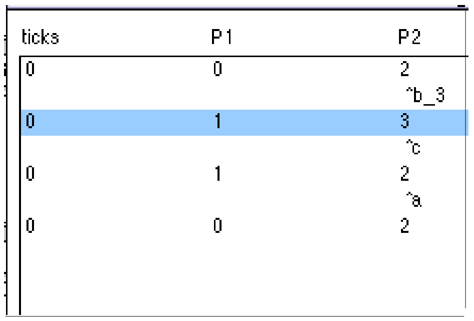
statecharts with a variety of timing properties not supported by Statemate while at the same time providing better analytical tools (model-checking and theorem proving) and a simpler semantics (via the TTMs of Section 3.3).

TTMcharts support shared interactions in the style of CSP or the Ada rendezvous (see the shared transition in Figure 16). The broadcast semantics of state-

FIGURE 16. Shared transitions allow objects to synchronize

Events a and c are local. Event b is shared





```
ticks        P1         P2
0            0          2
                         ^b_3
0            1          3
                         ^c
0            1          2
                         ^a
0            0          2
```

The event b (occurring in objects p1 and p2) is declared to be shared (indicated by the “#” symbol at the end of the event name). As shown in the simulation above, when the corresponding transition b_3 is taken, both p1 and p2 change to activities 0 and 2 simultaneously.

charts, and the division of steps into microsteps, makes it a useful tool for describing certain systems (e.g. ethernet protocols) that would require more constructs for their description using TTMs. However, the resulting statechart semantics is quite complex; there are various ways of providing a semantics for statecharts each with its own difficulties [22,42].

Statemate is a mature tool whereas StateTime is an incomplete prototype by comparison. The most important deficiency in StateTime is that, presently, only integer and finite enumeration types are available for data variables, whereas Statemate has the full range of types available in normal programming languages. Also, Statemate automatically generates code (to C and Ada) for the statechart descriptions. Statemate has the ability to display interconnections and data flow between modules using activity charts.

Statemate does not provide the same flexibility as StateTime for modelling timing constraints. For example, Statemate provides exact delay and schedule constructs, but no facility to directly distinguish between spontaneous, just and timed events. All untimed events are immediately executed as a sequence of microsteps before the clock is incremented. Statemate cannot express timed transitions in a direct manner. For example, to represent a transition $\tau[2, 5]$, two timeouts (one for the lower and one for the upper time bound) and some intermediate actions and states are needed.

There is also a fundamental difference between StateTime and Statemate with respect to verification. The Statemate reachability test can check whether there is a computation from the initial state to a specified condition. But it does not have the rich model-checking and theorem proving facilities of StateTime. The ability of StateTime to check for the most permissive bounds of response properties is an example of a a useful analytical technique. But many ordinary deadlock, safety and liveness properties cannot easily be captured by reachability analysis alone.

Other commercial tools have been developed incorporating structured methods for specifying real-time systems requirements [16,44,48]. These tools are in actual use, and have been successful on the whole in removing ambiguities in the requirements. However, these methods are at best semi-formal. They lack a precise semantics and rigorous verification methods (e.g. through model-checking, proof calculi or algebraic bisimulations).

An important part of tools such as Statemate, Objectime [44], and Observ [46] is the use of visual structuring techniques of state machines that are intuitively appealing to engineers. Research prototype tools such as Modechart [17,23] and ExSpect [47] also use graphical methods based on statecharts or Petri nets, and in addition support various forms of formal analysis.

There are many ways to model reactive systems and mechanically calculate their properties. At one end of the spectrum, we can test a system's behaviour on selected inputs (simulation or execution). In the middle of the spectrum (for relatively small finite state systems) we can model-check systems by systematically enumerating all possible behaviours. At the other end of the spectrum, we can conduct a mechanically checked proof that all (possibly infinitely many) system behaviours satisfy the requirements. In the StateTime toolset, analysis can take place anywhere in the spectrum (simulation, model-checking and theorem proving) as needed.

The Verify tool is slow relative to more recent model-checking tools. A first step towards improving the model-checking facility was to use the algorithm developed in [39] to translate TTMs into untimed fair transition systems. Hence untimed tools such as STeP, Spin [21] and SMV [6] can then augment Verify (as illustrated in Section 5.4).

We are currently investigating ways of extending StateTime by adding translators to recent analysis tools for timed automata. The SMC procedure [18] reduces the problem of model checking timed automata (with real-valued clocks) to the verification of finite state region automata. Since the clocks range over the real numbers, the state space is potentially infinite, and the state sets — called regions — must be represented symbolically rather than enumeratively. The convex data regions can be represented by integer matrices and manipulated using standard matrix operations. The SMC procedure has been implemented in such tools as Kronos [11], Cospan [3], HyTech [2] and HSIS.

Although, in theory, the computational complexity of the verification problem is proportional to the magnitudes of the clock upper bounds, in practice the performance can often be insensitive to the size of the delays more often than for the discrete clocks used by Verify. However, dense clocks also introduce unavoidable inefficiencies in comparison to discrete time. Dense clocks have an exponential dependency on the number of clocks used. Also, singular punctuality properties such as $p \Rightarrow \Diamond_{[1,1]}q$ are undecidable [35].

Most of the tools allow for both explicit enumeration of states and symbolic methods. Typically these tools can enumerate up to 10 million states with workstations that have 1GB of RAM in about 15 minutes.

If symbolic methods are used, much larger state spaces can sometimes be analyzed. For example, SMV and Cospan use binary decision diagrams (BDDs). However, BDDs are sensitive to variable ordering and to data variables and arithmetic operations (e.g. the two integer variables *Ta* and *Tb* of the DRT described in Section 5.3).

Both Cospan and SMV directly describe *synchronous* systems (all enabled transitions are taken simultaneously and that counts as one tick of the clock), which make them more suitable for hardware verification. In order to use them on interleaved systems or programming code, program counters must be introduced and used to disable all transitions except for the current step. This often leads to inefficient use of the BDD techniques, which are more suited to synchronous systems.

SMV supports branching time temporal logic as the specification language. Cospan uses automata theoretic methods for verification. The system is described by an automaton P, and the “task” which P is intended to perform by another automaton T. Conformance can be checked by testing the formal language containment $\mathcal{L}(P) \subset \mathcal{L}(T)$.

Spin supports both explicit state enumeration as well as partial order reductions (PO). POs are a family of techniques for diminishing the state-space explosion problem for model-checking concurrent programs. It is based on the observation that execution sequences of a concurrent program can be grouped together into equivalence classes that are indistinguishable by the property to be checked. The reduction procedure constructs a smaller state-space by generating at least one representative node for each equivalence class. POs seem better than BDDs for dealing with interleaved concurrent code, whereas BDDs appear to be better than POs for synchronous hardware. Comparing the various tools will require careful analysis and testing [10].

Spin supports linear temporal logic (LTL) model checking which is exponential in the size of the temporal logic specification (number of connectives). The branching time CTL logic supported by SMV can be checked in time proportional to the size of the specification. CTL and LTL are incomparable; each can express properties that the other cannot. LTL is better at expressing certain fairness properties, whereas CTL can specify the existence of a computation satisfying a property more easily.

STeP's model-checker is based on LTL. Since STeP also has a theorem prover, the system is first submitted to a preprocessor before going to the model-checker. Where possible, the preprocessor simplifies the data types, eliminates unnecessary variables, instantiates parameters or bounded quantifiers, and simplifies the temporal logic specification using the theorem prover. This means that we need not know the limits of data variables in advance, and allows certain infinite systems to be model-checked.

Some tools (e.g. Cospan) have translators for hardware description languages (e.g. Verilog), and other tools (e.g. Spin) have front-end programming languages. However, the model-checking or automata theoretic tools discussed in this section do not support visual descriptions; rather, they describe systems in ASCII text by low level transition descriptions. Graphical state diagrams often provide important information to reviewers that is difficult to derive from low level transition information alone. For an industrial case study, see the air collision avoidance system reported in [27]. Visual tools such as StateTime will continue to be important in the commercial setting. One possibility for improving the StateTime tool would be to allow programming code to be inserted in the lowest level objects (i.e. in chart activities); but more research is needed to determine the most appropriate programming constructs for reactive systems.

7.0 Conclusions

StateTime is a prototype toolset that allows for the design of real-time discrete event systems using an executable visual formalism (the Build tool). A timing hierarchy of spontaneous, just and forced timed events, and a variety of computational notions such as concurrency, nondeterminism, process interaction and communication can be represented.

The combination of model-checking (the Verify tool) and theorem proving allows for the treatment of finite and infinite systems as well as systems with unspecified time bounds. The Verify tool can compute the most permissive bounds of response properties, which is useful for debugging timing properties.

The reactor shutdown system illustrates how a variety of informal descriptions (timing diagrams, pseudocode, and English language specifications) can be represented in a unified way using the StateTime tool.

The StateTime toolset is in the process of being enhanced with other tools, in addition to Verify and STeP, that will improve its analysis capabilities. As discussed in the previous section, the various analysis tools complement each others' weaknesses.

Systems of realistic size present a challenge to tools such as StateTime, involving as they do the combinatorial explosion of states. Recently, methods have been developed within the TTM / RTTL framework, for dealing with larger systems by decomposing them into TTM modules. TTM modules consist of an interface specification, a body, and an RTTL behavioural specification. The recent advances mentioned below still need to be integrated into StateTime:

- The current tool can be used compositionally, where the correctness of the system can be obtained from the correctness of its modules [37,38], but this process is not fully automated yet (the module environment must still be generated by hand).
- Also, using algebraic abstraction (quotient systems), compositionally consistent model reduction techniques can be applied to modules, that preserve a class of RTTL properties to be verified. The abstract version of the module is more amenable to model-checking, as its state-space is often much smaller than that of the original [26].

The nuclear reactor shutdown system had to be reverse engineered because the implementation of the controller (pseudocode) was supplied in the requirements document. The composition of TTM modules, and the model reduction techniques mentioned above will not only help with the state explosion problem of already designed systems, but will allow for better *a priori* design methods.

An iterative design method initially describes the system at a high level of abstraction. Simulations can be performed to validate the model, and conformance to requirements can be checked. The model is iteratively refined until it reaches the level where it can be implemented. At every level of abstraction more simulations are performed, and properties specific to that level are verified. Also, the model at each level can be decomposed into modules, and each module can then be refined independently.

8.0 References

[1] Alur, R. and T.A. Henzinger. "A Really Temporal Logic." *Journal of the ACM*, 41(1): 181-204, 1994.

[2] Alur, R., T.A. Henzinger, and P.-H. Ho. "Automatic Symbolic Verification of Embedded Systems." *IEEE Transactions on Software Engineering*, 22(3): 181-201, 1996.

[3] Alur, R. and R.P. Kurshan. "Timing Analysis with Cospan." In *Hybrid Systems III*, ed. R. Alur, T.A. Henzinger, and E. Sontag. LNCS 1066 Springer Verlag, 1996.

[4] Boussinot, F. and R.d. Simone. "The SL Synchronous Language." *IEEE Trans. on Software Engineering*, 22(10): 256-266, 1996.

[5] Brave, Y. and M. Heymann. “Control of Discrete Event Systems Modelled as Hierarchical State Machines.” *IEEE Transactions on Automatic Control*, 38(12): 1803-1819, 1993.

[6] Burch, J.R., E.M. Clarke, K.L. MacMillan, D.L. Dill, and L.J. Hwang. “Symbolic Model Checking: 10^20 States and Beyond.” *Information and Computation*, 98(2): 142-170, 1992.

[7] Cassandras, C., S. Lafortune, and G. Olsder. “Introduction to the Modelling, Control and Optimization of Discrete Event Systems.” In *Trends in Control: A European Perspective*, ed. A. Isidori. 217-291. Springer-Verlag, 1995.

[8] Cassandras, C.G. *Discrete Event Systems: Modeling and Performance Analysis*. Irwin Inc. and Aksen, Homewood, IL, 1993.

[9] Chandy, K.M. and J. Misra. *Parallel Program Design*. Addison-Wesley, 1988.

[10] Corbett, J.C. “Evaluating Deadlock Detection Methods for Concurrent Software.” *IEEE Trans. on Software Engineering*, 22(3): 161-180, 1996.

[11] Daws, C. and S. Yovine. “Two Examples of Timed Automata Using Kronos.” In *Proc. 16th Annual Real Time Systems Symposium*, IEEE CS Press, 66-75, 1995.

[12] Dyck, D.D. and P.E. Caines. “The Logical Control of an Elevator.” *IEEE Trans. on Automatic Control*, 40(3): 480-486, 1995.

[13] Hadj-Alouane, N.B., S. Lafortune, and F. Lin. “Variable Lookahead Supervisory Control with State Information.” *IEEE Trans. on Automatic Control*, 39(12): 2398-2410, 1994.

[14] Harel, D. “Statecharts: A Visual Formalism for Complex Systems.” *Science of Computer Programming*, 8:231-274, 1987.

[15] Harel, D., H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and M. Trachtenbrot. “Statemate: a working Environment for the Development of Complex Reactive Systems.” *IEEE Transactions on Software Engineering*, 16:403–414, 1990.

[16] Hatley, D.J. and I.A. Pirbhai. *Strategies for Real-Time System Specification*. Dorset House Publishing Co., New York, 1988.

[17] Heitmeyer, C. and B. Labaw. “Requirements Specification of Hard Real-Time Systems: Experience with a Language and a Verifier.” In *Foundations of Real-Time Computing: Formal Specifications and Methods*, Kluwer Academic Publishers, 1991.

[18] Henzinger, T.A., X. Nicollin, J. Sifakis, and S. Yovine. “Symbolic Model Checking for Real-Time Systems.” *Information and Computation*, 111(2): 193-244, 1994.

[19] Hoare, C.A.R. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, N.J., 1985.

[20] Holloway, L.E. and B.H. Krogh. “Synthesis of Feedback Control Logic for a Class of Controlled Petri Nets.” *IEEE Trans. on Automatic Control*, 35(4): 514-523, 1990.

[21] Holzmann, G.J. *Design and Validation of Protocols*. Prentice Hall, 1990.

[22] Hooman, J.J.M., S. Ramesh, and W.P.d. Roever. “A Compositional Axiomatization of Statecharts.” *Theoretical Computer Science*, 101(2): 289-335, 1992.

[23] Jahanian, F. and A.K. Mok. “Modechart: A Specification Language for Real-Time Systems.” *IEEE Transactions on Software Engineering*, 20(12): 933-947, 1994.

[24] Lawford, M. “Transformational Equivalence of Timed Transition Models.” Systems Control Group, Department of Electrical Engineering, University of Toronto. TR-9202 (M.A.Sc. thesis), 1992.

[25] Lawford, M. and W.M. Wonham. “Equivalence Preserving Transformations for Timed Transition Models.” *IEEE Trans. on Automatic Control*, 40(7): 1167-1179, 1995.

[26] Lawford, M., W.M. Wonham, and J.S. Ostroff. “State-Event Labels for Labelled Transition Systems.” In *Proc. 33rd IEEE Conference on Decision and Control*, Orlando, FL, IEEE Control System Society, 3642-3648, 1994.

[27] Leveson, N.G., M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. “Requirements Specification for Process-Control Systems.” *IEEE Transactions on Software Engineering*, 20(9): 684-707, 1994.

[28] Lin, J.-Y. and D. Ionescu. “A Reachability Synthesis Procedure for Discrete Event Systems in a Temporal Logic Framework.” *IEEE Trans. on Systems, Man and Cybernetics*, 24(9): 1397-1406, 1994.

[29] Manna, Z. *Mathematical Theory of Computation*. McGraw-Hill, New-York, 1974.

[30] Manna, Z. “STeP: The Stanford Temporal Prover.” Dep. of Computer Science, Stanford University. STAN-CS-TR-94-1518, 1994.

[31] Manna, Z. and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, 1992.

[32] Ostroff, J.S. *Temporal Logic for Real-Time Systems*. Advanced Software Development Series, ed. J. Kramer. Research Studies Press Limited (distributed by John Wiley and Sons), England, 1989.

[33] Ostroff, J.S. “Deciding properties of Timed Transition Models.” *IEEE Transactions on Parallel and Distributed Systems*, 1(2): 170-183, 1990.

[34] Ostroff, J.S. “Systematic Development of Real-Time Discrete Event Systems.” In *Proceedings of the ECC91 European Control Conference*, Grenoble, Hermes Press, 522–533, 1991.

[35] Ostroff, J.S. “Design of Real-Time Safety Critical Systems.” *The Journal of Systems and Software*, 18(1): 33–60, 1992.

[36] Ostroff, J.S. “A Verifier for Real-Time Properties.” *Real-Time Journal*, 4:5–35, 1992.

[37] Ostroff, J.S. “Automated Modular Specification and Verification of Real-Time Reactive Systems.” In *Proc. Workshop on Industrial Strength Formal Specification Techniques WIFT'95*, IEEE Computer Society Press, 108-121, 1995.

[38] Ostroff, J.S. “A CASE Tool for the Design of Safety-Critical Systems.” In *Proc. Seventh International Workshop on Computer Aided Software Engineering CASE'95*, IEEE Computer Society Press, 370-380 (see also http://www.cs.yorku.ca/pub/ostroff/papers.95/case95.extended.pdf for an extended version of the paper), 1995.

[39] Ostroff, J.S. and H.K. Ng. “The Design of Real-Time Systems Using Standard Untimed Theories.” In *Preprints Third AMAST Workshop on Real-Time Systems*, Salt Lake City, Utah, ONR and Iowa University, 1996.

[40] Ostroff, J.S. and W.M. Wonham. “A Framework for Real-Time Discrete Event Control.” *IEEE Transactions on Automatic Control*, 35(4): 386–397, 1990.

[41] Parnas, D.L., G.J.K. Asmis, and J. Madey. “Assessment of Safety-Critical Software in Nuclear Power Plants.” *Nuclear Safety*, 32(2): 189-198, 1991.

[42] Pnueli, A. and M. Shalev. “What is in a Step?” In *Theoretical Aspects of Computer Software*, 244-264. Springer-Verlag, 1991.

[43] Ramadge, P.J.G. and W.M. Wonham. “The Control of Discrete Event Systems.” *Proc. of the IEEE*, 77(1): 1989.

[44] Selic, B., G. Gullekson, J. McGee, and I. Engelberg. “ROOM: An Object-Oriented Methodology for Developing Real-Time Systems.” In *CASE’92 Fifth International Workshop on Computer-Aided Software Engineering*, Montreal, IEEE Computer Society Press, 230-240, 1992.

[45] Thistle, J.G. “Logical Aspects of Control of Discrete Event Systems: A Survey of Tools and Techniques.” In *11th Int'l Conf. on Analysis and Optimization of Systems: Discrete Event Systems*, ed. H. Cohen and J.-P. Quadrat. LNCIS No. 199. Springer-Verlag, 1994.

[46] Tyszberowicz, S. and A. Yehudai. “OBSERV — A Prototyping Language and Environment.” *ACM Transactions on Software Engineering Methodology*, 1(3): 269-309, 1992.

[47] van der Aalst, W.M.P. “Timed Coloured Petri Nets and their Application to Logistics.” Ph.D, Eindhoven University of Technology, 1992.

[48] Ward, P. and S. Mellor. *Structured Development for Real-Time Systems*. Yourdon Press, New York, 1985.