To appear in: *Theories and Experiences for Real-Time Systems Development*.
Revision of paper presented at: *First AMAST Workshop on Real-Time Systems, Univ. of Iowa, Nov. 93.*

Chapter 3

Visual Tools for Verifying Real-Time Systems

Jonathan S. Ostroff.

3.1 Introduction

Computers are increasingly used to monitor and control safety critical systems. Real-time software controls aircraft, shuts down nuclear power reactors in emergencies, keeps telephone networks running, and monitors hospital patients. The use of computers in such systems offers considerable benefits, but also poses serious risks to life and the environment [15].

Visual tools based on extended state machines, Petri nets and statecharts have been proposed for modelling real-time systems. The statechart approach has been found particularly useful because of its appealing hierarchical, communication and concurrency constructs. There are already industrial strength tools available including Statemate [4] and ObjectTime [17].

Statemate [4] is one of the few available commercial tools that is based on a formal model (statecharts), allows for dynamic execution in which triggers can be provided interactively on the fly, and which can formally verify various properties including reachability of conditions, deadlock, nondeterminism, usage of elements and racing.

The *StateTime* tool discussed in this paper is merely a prototype and thus cannot be compared to Statemate in many important respects. For example, in StateTime only integer types are available for data variables, whereas Statemate has the full range of types available in normal programming languages. However, StateTime (while retaining the hierarchical and concurrent constructs of statecharts) has facilities for expressing and verifying certain types of real-time properties that Statemate does not deal with, at least not directly.

Statemate has time-out and scheduled actions. These types of events allow for an exact delay of a specified period after which an event occurs. By contrast StateTime has a much richer hierarchy of timing properties:

Usually, in any given real-time system, three very different types of timing constraints may need to be asserted on system transitions. In order of increasing stringency of timing they are: spontaneous, just and timed transitions.

Spontaneous transitions may occur at any point in time that they are enabled, or they may never occur. An example is the event of a device failure. In the sequel, spontaneous transitions are indicated by the fact that their upper time bound is infinity (∞).

Just transitions must eventually occur if they are continually enabled. For example, a clock must always eventually tick. Justice is qualitative in the sense that although a just transition must occur, no finite bound on the time of occurrence is given.

Timed transitions must occur within an interval specified by a lower and an upper time bound.

Statemate does not deal with the contrast between justice conditions and spontaneous events. Nor is it able to express timed events in a direct manner. For example, to represent a transition $\tau\,[2, 5]$ with lower bound 2 and upper bound 5, two time-outs and some intermediate events and states are needed.

A more fundamental difference appears in the manner in which properties are verified. In Statemate, the reachability test can check whether there is *a* path to condition from the initial state. By contrast, StateTime generates the complete reachability graph, and can therefore check that a certain property holds in *all* computations.

StateTime checks a small subset of temporal logic properties without the need for a watchdog. A watchdog is an observer that has access to all the system variables without affecting them. Usually, adding a watchdog compounds the problem of combinatorial explosion of states.

StateTime is based on the TTM/RTTL framework [11,10,9,15], which has a somewhat simpler semantics than that used by Statemate (this makes verification easier, but removes some of the expressiveness of Statemate).

The TTM/RTTL framework has a precise notion of real-time, coupled with the ability to deal with a variety of models of computation (e.g. concurrent processes using shared variables, Petri Nets and CSP). TTMs are timed transition models, which can be used to represent concurrent processes, non-deterministic behaviour, communication between modules, real-time constraints, and structured programs. RTTL is real-time temporal logic. Temporal logic has been found to be a useful specification language that can express a variety of properties, including freedom from deadlock, mutual exclusion of critical regions, liveness properties (such as processes that eventually access their critical regions and process fairness), and real-time response.

The system under design (SUD) is usually divided into two parts: the PLANT and the CONTROLLER. The CONTROLLER is that part of SUD that is currently unknown and must be designed.The PLANT consists of the environment in

which the controller must function. The designers job is: given a PLANT and a specification of correct plant behavior, design a CONTROLLER so that SUD satisfies its specification.

StateTime Terminology

Here is a brief review of the terms that are used in the sequel:

1. The **TTM/RTTL** framework — the underlying mathematical theory that the StateTime toolset is based on. TTMs are Timed Transition Models. RTTL is Real-Time Temporal Logic. TTMs are mathematical models of networks of interacting distributed processes. RTTL is a rigorous specification language for stating how the models ought to behave. Without an underlying mathematical theory, the Statetime toolset would not be able to execute systems and verify their correctness.
2. **TTMchart**s — a visual language for representing TTMs. Graphical notions are provided for representing states (called *activities*), events, concurrent processes, hierarchy, timing and program statements (assignments). TTMcharts is based on Statecharts, but with a different notion of timing and process interaction. A TTM is a mathematical entity. A TTMchart is a concrete visual representation of that mathematical entity.
3. The **StateTime** toolset consists of many tools. The ones used in this paper are:
   (a) The **BUILD** tool — a tool that provides automated support for drawing and executing (simulating) TTMcharts.
   (b) The **VERIFY** tool — a tool that automatically verifies that a *finite state* TTMchart satisfies an RTTL property.

The term "TTM" and "TTMchart" are often used interchangeably where it is clear from the context what is meant.

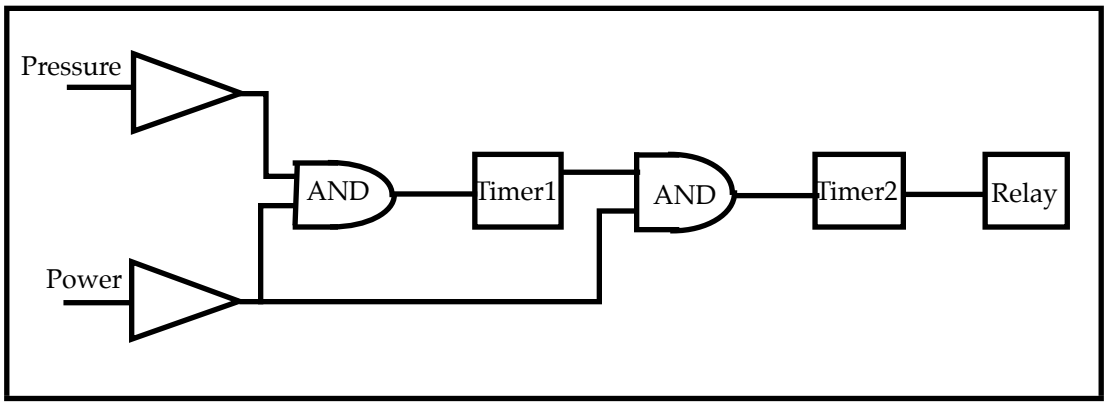
3.2 Using BUILD to model the problem

The first step in the design process is to discover and describe as much as possible about the problem domain. Informal requirements must be translated into suitable TTMs and RTTL specifications.

The delayed reactor trip (DRT) problem was first described by Mark Lawford in [7]. It is an excellent example that is small enough to be described in this paper yet non-trivial. Lawford developed behaviour preserving transformations for TTMs with which he was able to discover a flaw in the proposed design. However, the theory cannot be fully automated as no set of transformations is complete for proving observation equivalence between the actual implementation and its abstract specification. We will analyze the problem from a temporal logic (RTTL) perspective, and will attempt to use completely automated verification procedures to check the correctness of the implementation.

The delayed reactor trip for the CANDU nuclear reactors is currently implemented in hardware using timers, comparators and logic gates as shown in Figure 1. The new DRT system is to be implemented in future on a microprocessor

FIGURE 1. Analog implementation of the delay relay trip system DRT (the "controller").



system. 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 specifications:

[S1] 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.

Specification S1 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 specification S2 states:

[S2] If the power reaches an acceptable level then the relay should be closed (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:

[S3] The controller should never deadlock. For example, if after the power and pressure have exceeded their critical values, and the system has waited 3 seconds to check the power level again, if the power is below its

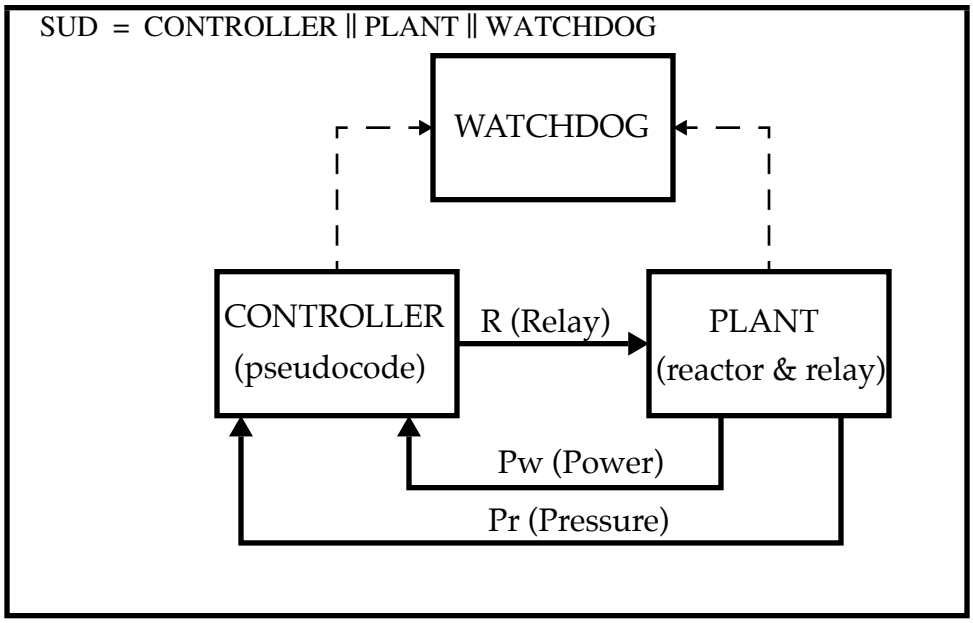
critical limit, then the system must reset and go back to monitoring its inputs.

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.

It is possible to try to analyze the complete system of three concurrent microprocessors using the TTM/RTTL approach. However, it is preferable to start by first checking that each individual processor on its own achieves proper control. It is important in general to verify components before proceeding to the larger picture. In addition to "theoretical correctness", this has important practical ramifications. Larger systems have greater state spaces to explore that may be beyond the current limits of automated verification. If a component can be verified to be correct in all its detail, then a reduced order model of it may be used when checking the component in the broader context, thus reducing exponential explosion of states.

The new DRT software controller is to be implemented on a microprocessor system with a cycle time of 100ms. The software controller samples the inputs and passes through a block of control code every 0.1 seconds. It is assumed that the input signals have been properly filtered and that the sampling rate is sufficiently fast to ensure adequate control.

FIGURE 2. Relationship between controller, plant and watchdog of SUD



Lawford obtained the pseudocode for the proposed software controller from the actual CANDU requirements document for the construction of the DRT [7]. The pseudocode as reported by Lawford is shown in Figure 3. The code mimics the original analog implementation by using integer variables c1 and c2 in place of Timer1 and Timer2 respectively. The program also makes use of the variables

FIGURE 3. Pseudocode for the DRT taken from Lawford's report [7].

```
If Pressure ≥ DSP then
    If Power ≥ PT then
        If counter c1 is reset then
            If counter c2 is reset then
                increment c1                    ]Transition : μ1
            Else
                If counter c2 has timed out then
                    reset c2                    ]Transition : γ
                Else
                    increment c2                ]Transition : μ2
                    open Relay                  ]
                Endif
            Endif
        Else
            If counter c1 has timed out then
                open Relay                      |
                reset c1                        ]Transition : α
                increment c2                    ]
            Else
                increment c1                    ]Transition : μ1
            Endif
        Endif
    Endif
Else
    If counter c1 is reset then
        If counter c2 is reset then
            close Relay                         ]Transition : β
        Else
            If counter c2 has timed out then
                close Relay                     ]Transition : ρ2
                reset c2                        ]
            Else
                increment c2                    ]Transition : μ2
                open Relay                      ]
            Endif
        Endif
    Else
        If counter c1 has timed out then
            reset c1                            ]Transition : ρ1
        Else
            increment c1                        ]Transition : μ1
        Endif
    Endif
Endif
```

Pr (Pressure), *Pw* (Power) and *R* (Relay) for the sampled inputs (Pressure and Power) and output (Relay) of the controller.

The DRT system under design (SUD) consists of the parallel composition of three components, i.e.

$$SUD = CONTROLLER \parallel PLANT \parallel WATCHDOG$$

The relationship between plant and controller is shown in Figure 2. The CONTROLLER is the abovementioned pseudocode for the microprocessor as displayed in Figure 3. The PLANT is the environment in which the controller operates, i.e. the nuclear reactor which generates the power and pressure variables, and the relay that opens or closes depending on the value of the relay variable *R* set by the controller. The WATCHDOG is a non-invasive observer of the plant and controller variables (i.e. it has access to all the system variables but does not in any way change or control them). The WATCHDOG is used for verification only and is not part of the actual implementation (this will be explained further in the sequel).

Modelling the plant

The first step is always to model the plant. The power and pressure variables are assumed to be filtered. This will be modelled by allowing them to be updated every two ticks of the clock, where one tick of the clock is 100ms. The StateTime tool BUILD is used to construct the model of the plant as shown in Figure 4, and in Figure 5. The dotted lines around the activity "power_update" in Figure 4 indicate that it is composed in parallel with the activity "pressure_update", i.e.

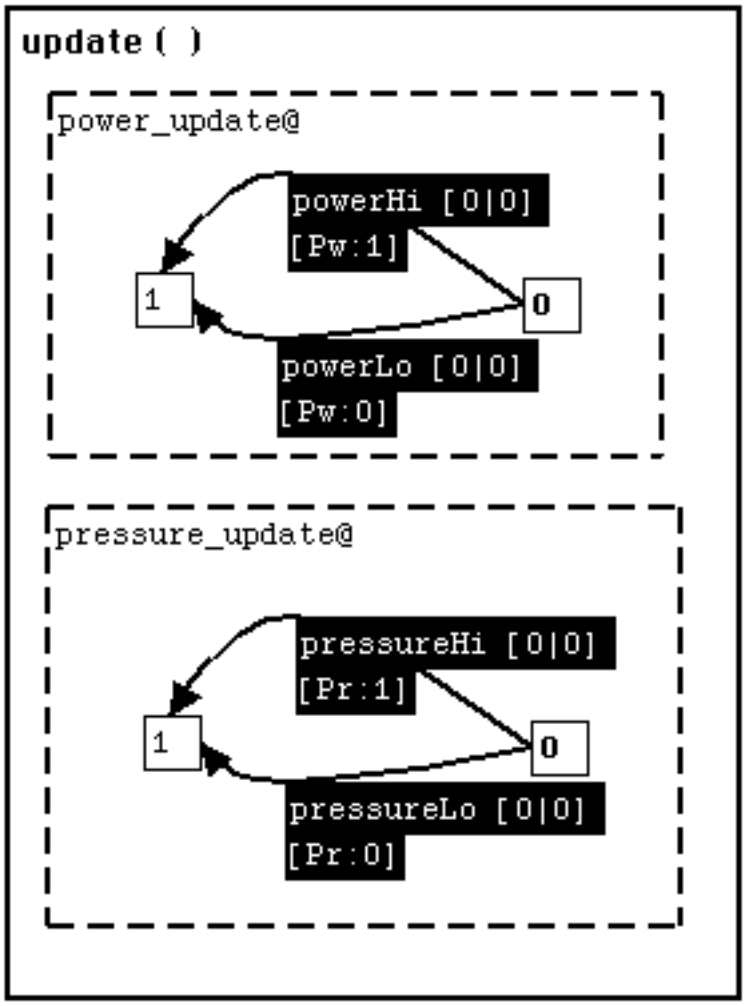
$$update = power_update \parallel pressure_update$$

To be in the super-activity "update", is to be in the sub-activities "power_update" and "pressure_update" at the same time (AND-decomposition). The default sub-activity of "power_update" is "**0**" (default activities are shown in bold), i.e. each time "power_update" is invoked it is assumed to start in "**0**", from which the power variable *Pw* can be assigned 0 (meaning the power is within an acceptable range), or 1 (meaning that the power is too high). The transition "powerHi[0,0]" takes "power_update" from its sub-activity "**0**" to "1" at the same time assigning 1 to the variable *Pw*. The lower and upper time bounds are both zero indicating that "powerHi" is taken before the next tick of the clock. Similarly, the "pressure-update" activity describes how the pressure variable *Pr* is updated.

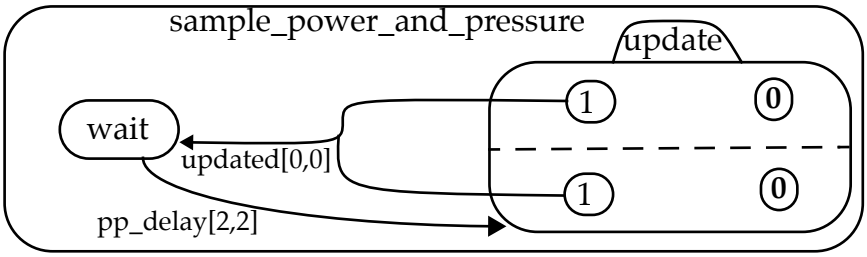
The "update" activity (of Figure 4) is used to build the super activity labelled "sample_power_and_pressure" as shown in Figure 5, which is the XOR-composition of the sub-activities "wait" and "update". To be in the XOR activity "sample_power_and_pressure" is to be either in "wait" or "update" but not both simultaneously. The symbol "@" at the end of "update@" in Figure 5 indicates that "update" has internal structure, i.e. "zooming in" to "update" will display Figure 4.

The transition *updated[0,0]* exits from the outer contour of the structured activity "update". The default meaning is that no matter where in the structured activity "update" the two threads of control currently resides, the transition *updated[0,0]* is eligible to occur. However, a special detail view of the transition *updated[0,0]* can be invoked in which the default behaviour can be changed. In this case, the transition is changed so that it is eligible to occur only when both

FIGURE 4. The "update" activity is AND-decomposed into the update for the power variable *Pw* and the pressure variable *Pr*.

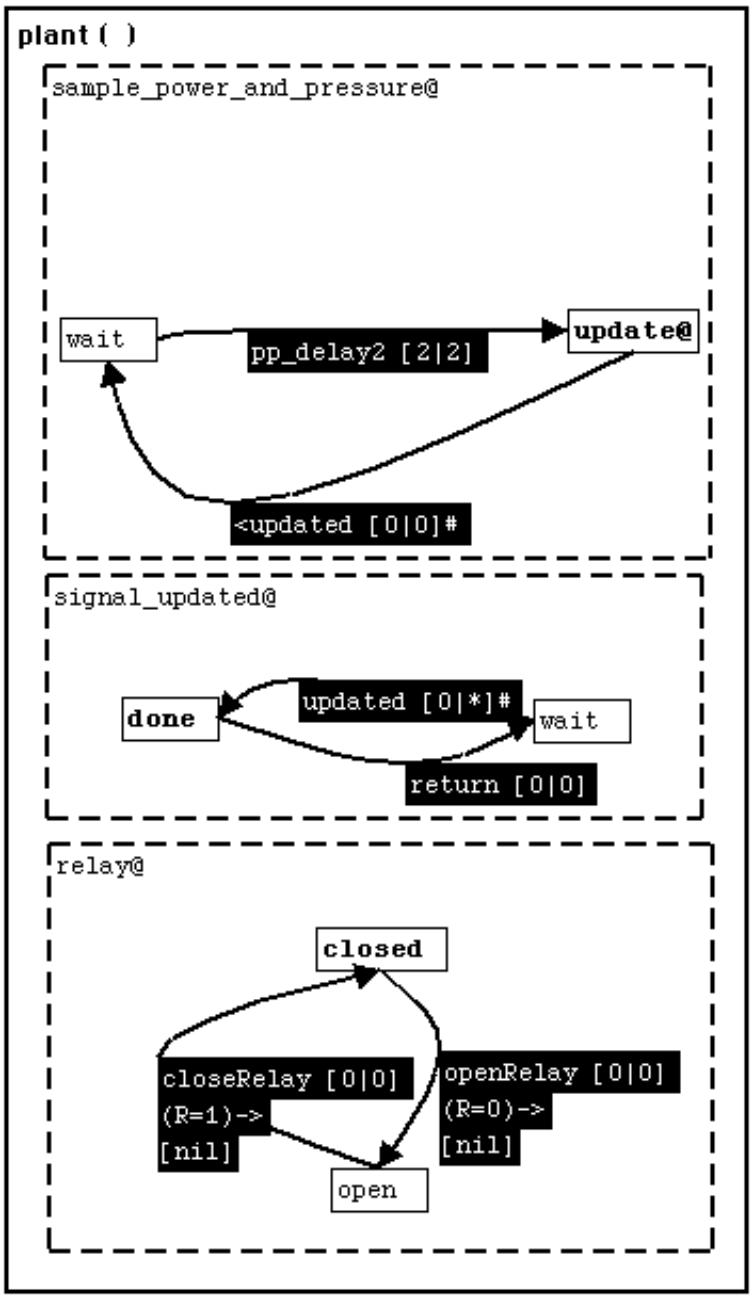


"power_update" and "pressure_update" are in their sub-activity 1 as shown in the diagram below (hence the transition *updated[0,0]* occurs immediately after the



variables are both sampled). The transition *pp_delay2[2,2]* has the outer countour of activity "update" as its destination, meaning that when it is taken it goes to the

FIGURE 5. DTR plant is the AND-composition of the relay, power and pressure updates.



default activities of “update” (indicated in bold). This default behaviour can also be changed. The transition *pp_delay2* must wait for two ticks before it is taken.

The plant is defined as the AND-composition given by

$$\text{plant} = \text{relay} \parallel \text{sample_power_and_pressure} \parallel \text{signal_update}$$

Transitions may be declared *local* or *shared*. A transition that is declared shared is synchronized with any other transitions of the same name in concurrent activities. All the shared component transitions block until they are all simultaneously eligible and all are then taken together. Shared transitions can thus be used to represent the rendezvous in Ada or the CSP notion of synchronization. In this respect the TTM model differs from statecharts which communicate via broadcasting.

In Figure 5, the component transitions *updated[0,∞]#* in the "signal_update" activity, and *updated[0,0]#* in the concurrent "sample_power_and_pressure"activity, are partners in a composite shared transition (the suffix # indicates that they are both declared shared). Both partners block until they synchronize and are taken simultaneously.

The time bounds of the composite transition is the maximum of the individual component lower bounds, and the minimum of the component transitions upper time bounds. Hence the time bounds of the composite *updated* transition is [0,0]. Using these bound constraints, component transitions with tighter bounds may be thought of as "forcing transitions" that constrain the less tightly bound components.

The component transition *updated[0,∞]* in the "signal_update" activity is a spontaneous transition, i.e. from the point of view of "signal_update" alone it may occur at any moment or never. Of course, once "signal_update" is inserted into the broader context of the plant, the *updated[0,∞]* component is constrained by its forcing partner.

The activity "plant" may be thought of as the root activity. Sub-activities of "plant" such as "sample_power_and_pressure"are structured activities. And leaf activities such as "wait" are basic as they have no further internal structure. All activities, except for basic ones, have *activity variables*. For example, the activity variable of "signal_update" is S, with $type(S) = \{\text{done}, \text{wait}\}$. To express the fact that the plant is in the activity "done", we may write $(S = \text{done})$, which is true whenever the next update to the power and pressure variables is exactly in two ticks of the clock.

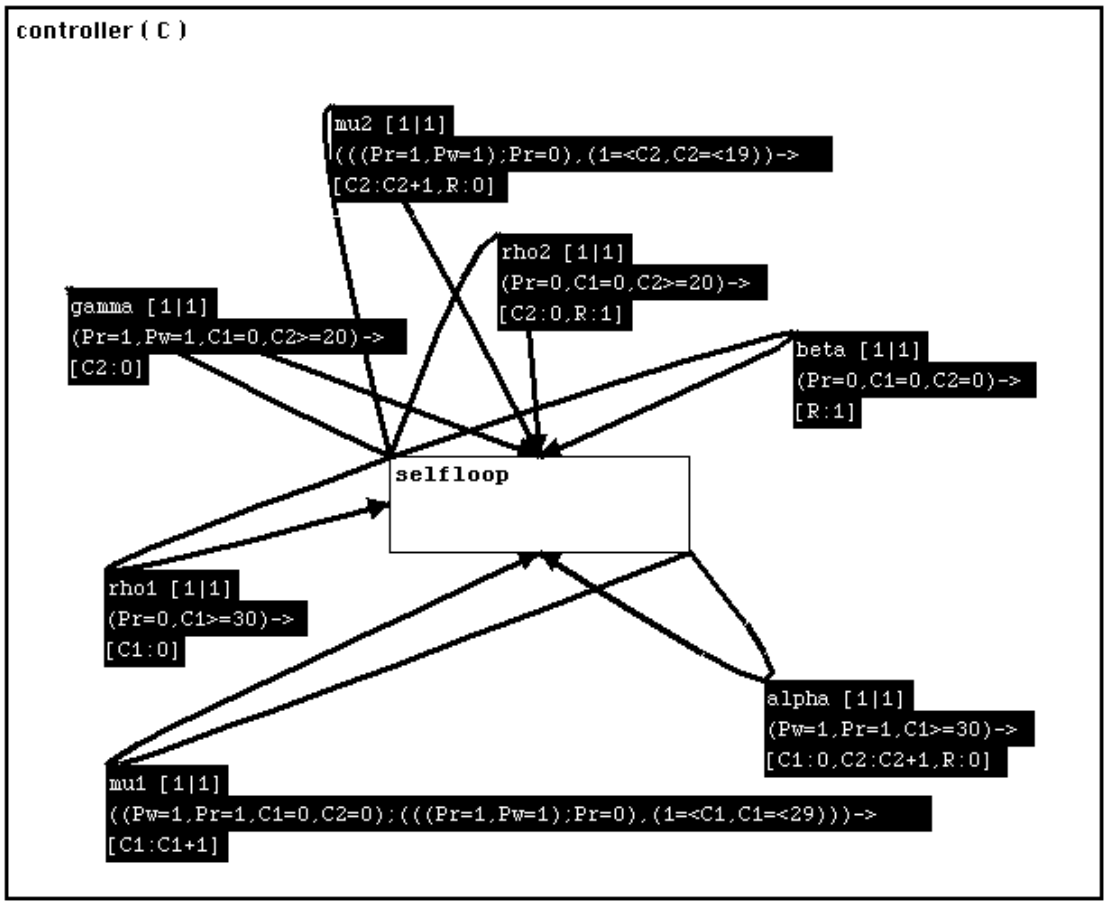
There are much simpler ways of being able to assert when no change in the plant data variables will occur for two ticks of the clock. These simpler models also speed up the automated verification as their reachability graphs are smaller. However, the above description allowed us to illustrate the hierarchical and synchronization features of the BUILD tool.

The software controller

Having modelled the PLANT of Figure 2, the next step is to obtain a TTM representation of the software CONTROLLER. The pseudocode of Figure 3 can be represented by the TTMchart "controller" shown in Figure 6 (see [7] for how this is done).

With each pass through the code, the microprocessor picks out one of the labelled blocks of code. The block chosen is the one whose enabling condition is satisfied. The program then loops back to the start and re-evaluates all the enabling conditions in the next cycle. The program structure is that of a large case

FIGURE 6. Faulty controller based on the proposed pseudocode for the microprocessor



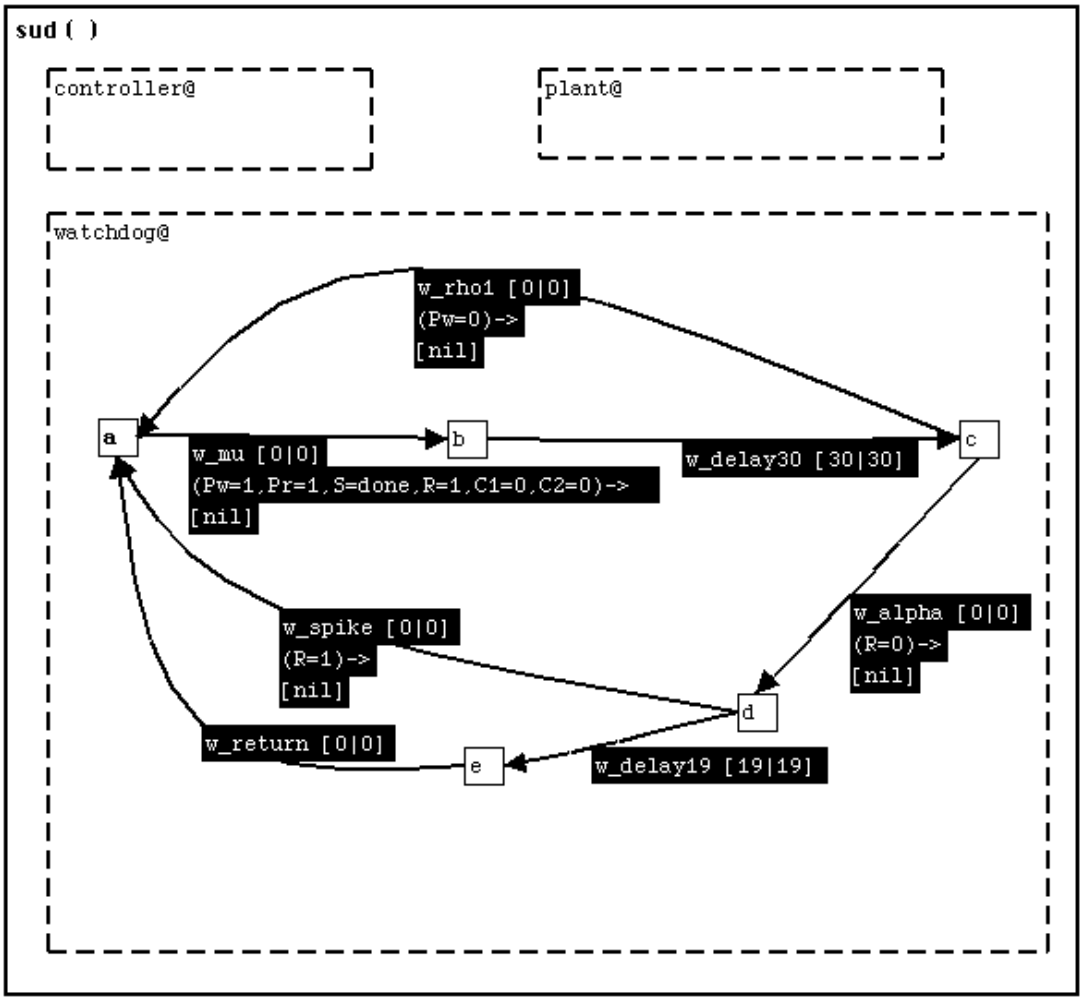
statement repeatedly executed. Hence each transition has a lower and upper time bound of one.

Conditions such as $Pressure \geq DSP$ (pressure exceeds delayed set point) and $Power \geq PT$ (power exceeds the power threshold) can be represented by $(Pr = 1)$ and $(Pw = 1)$ respectively ("1" represents beyond the critical threshold and "0" represents normal levels), as generated by the plant. In the guards of transitions a comma stands for conjunction and a semi-colon for disjunction. A transformation function such as [C2:C2+1, R:0] in transition *mu2[1,1]* of Figure 6 stands for simultaneous assignment (i.e. when the transition is taken variable C2 is incremented by one and R is assigned zero).

In general, it is relatively easy to transform real-time programs written in Ada, Petri nets or CSP-style Occam code into TTMs (see references [9,8]), and this process can in principle be automated.

The final TTMchart "sud" is obtained by AND-composing "plant" and "controller" as shown in Figure 7. The watchdog will be explained in the next subsection that deals with the RTTL specifications.

FIGURE 7. The complete system under design sud = controller || plant || watchdog



Temporal Logic Specifications

The informal specifications S1, S2 and S3 must now be translated into RTTL specifications.

The VERIFY tool works in conjunction with the BUILD tool to verify a TTM-chart. The current tool verifies a small but important set of specifications including safety (e.g. deadlock free), liveness (e.g. accessibility) and real-time response properties. The verifier is currently being extended to handle arbitrary real-time temporal logic properties. An important feature of the verifier is that it can deal with data variables directly.

The VERIFY tool computes the graph of all states that are reachable from the initial states of the TTM. Some of the specifications that VERIFY can check are:

f_1 entails *henceforth* f_2	$f_1 \Rightarrow \Box f_2$	In any reachable state s in which the state-formula f_1 is true, the formula f_2 must also be true in s and in all states reachable from s.
f_1 entails eventually f_2 within l to u ticks (*real-time response*). Given f_1 and f_2, VERIFY returns the minimum value for l and the maximum value for u.	$f_1 \Rightarrow \Diamond_{[l,u]} f_2$	In any reachable state s in which f_1 is true, all computations subsequent to s have in them a state s' which is at least l ticks but no more than u ticks after s, and f_2 is true in s'.

Consider specification S2 which states that:

[S2] If the power reaches an acceptable level then the relay should be closed (thus allowing the reactor to operate once more).

At first glance, S2 may be written:

$$(Pw = 0) \Rightarrow \Diamond_{\leq 1} (Relay = closed)$$

i.e. if the power level is normal, then within one tick (100ms) the relay must be closed. The problem with this assertion is that it will not always be true in every computation of the DTR, because even if the power level is acceptable in a given state, the level may become critical in the next state and hence the relay should not be closed. Furthermore, the microprocessor may not be fast enough to detect such instantaneous changes, even presuming that the above assertion is the correct one to enforce. Rather, we must assert that whenever the power is normal for a sufficiently long period of time, then the relay must be closed. S2 should therefore be written

$$\Box_{<2} (Pw = 0) \Rightarrow \Diamond_{\leq 1} (Relay = \text{closed})$$

meaning if any state is reached from which the power is low for up to two ticks of the clock, then eventually in all subsequent computations from that state, the relay must be closed within one tick. Using the activity variable S of "signal_update, we may re-write the above property as:

$$\text{[S2]:} \quad (Pw = 0 \wedge S = \text{done} \wedge \text{Reset}) \Rightarrow \Diamond_{\leq 1} (Relay = \text{closed}) \qquad \text{(EQ 1)}$$

using $\text{Reset} \stackrel{\text{def}}{=} (R = 0 \wedge C_1 = 0 = C_2)$, and the fact that $(S = \text{done})$ is true only when there are still two ticks to the next update. "Reset" is in the antecedent because the microprocessor does not close the relay while it is simulating the timing (Timer1 and Timer2). S2 as written in (EQ 1) is in a form that VERIFY can check.

The specification S1 can be written as

$$\text{BothHi} \wedge \text{Reset} \Rightarrow \Diamond_{=30}[\text{PowerHi} \rightarrow \Diamond_{\leq 1}(\text{RelayOpen} \wedge \Box_{<20}\text{RelayOpen})] \quad \text{(EQ 2)}$$

where $\text{BothHi} \stackrel{\text{def}}{\equiv} (Pw = 1 \wedge Pr = 1 \wedge S = \text{done})$,
$\text{Reset} \stackrel{\text{def}}{\equiv} (R = 1 \wedge C_1 = 0 = C_2)$, $\text{PowerHi} \stackrel{\text{def}}{\equiv} (Pw = 1 \wedge S = \text{done})$,
and $\text{RelayOpen} \stackrel{\text{def}}{\equiv} (R = 0)$.

Since (EQ 2) cannot be directly checked by the VERIFY tool, a watchdog (that observes but does not affect the plant or controller) must be constructed as shown in Figure 7. At activity "a" the watchdog detects when $(\text{BothHi} \wedge \text{Reset})$ becomes true, and then delays for 30 ticks (3 seconds). At this point, the watchdog is located at basic activity "c" (i.e. $(W = c)$ where W is the activity variable of the watchdog). At "c", the transition w_rho1[0,0] is immediately taken if the power is low, and w_alpha[0,0] is taken if the relay is open $(R = 0)$. If w_alpha is taken, then w_spike[0,0] checks that the relay is not opened for 19 ticks of the clock. After the 20th tick, the watchdog should be in activity "e". Thus (EQ 2) can be checked by verifying that

$$\text{[S1]:} \quad [(W = c) \wedge (Pw = 1 \wedge S = \text{done})] \Rightarrow \Diamond_{=20}(W = e) \quad \text{(EQ 3)}$$

which is in a format suitable for the VERIFY tool. Using the concept of a watchdog, most properties of interest can be checked in this way. However, there is a cost associated with adding the watchdog, as the size of the reachability graph is increased substantially.

The informal specification S3 requires that the system as a whole (SUD) not deadlock. The verifier is able to directly check that

$$\text{[S3a]:} \quad \Box(\text{enabled}) \quad \text{(EQ 4)}$$

which checks that in every reachable state there is at least one transition other than tick that is enabled. (If only a clock tick is enabled in a state, then SUD deadlocks in that state as all that it can ever do is tick).

Furthermore, it is advisable to check that the watchdog taken by itself never deadlocks. More specifically it should be the case that

$$\text{[S3b]:} \quad (W \neq a) \Rightarrow \Diamond_{\leq 50}(W = a) \quad \text{(EQ 5)}$$

Finally, the relay should not be opened unnecessarily, i.e.

$$((Relay = closed) \wedge (Pw = 0)) \Rightarrow (Relay = \text{closed}) \mathcal{W} (Pw = 1 \wedge Pr = 1) \quad \text{(EQ 6)}$$

$(p \mathcal{W} q)$ asserts that p is *waiting-for* q, i.e. p remains true unless q becomes true. The waiting-for specification can also be checked by the verifier.

The specifications S1,S2, S3a and S3b can all be listed using the BUILD tool. The BUILD tool can also be used to simulate (dynamically execute) the system under design (SUD). Simulation is important for ensuring that the model works in the expected fashion.

3.3 Using the VERIFY tool

We have now modelled the DTR by

$$SUD = CONTROLLER \parallel PLANT \parallel WATCHDOG$$

The verification problem is: does every computation of SUD satisfy the specifications S1,S2, S3a and S3b. The VERIFY tool can be used to do the check.

The VERIFY tool takes its input from the BUILD tool. VERIFY was written as an undergraduate project in Prolog. Its execution times are very slow (see footnote a. in Table 1). A new version is currently under construction using Smalltalk (the same language that BUILD is written in) that will verify arbitrary temporal properties without the need for a watchdog. Preliminary tests indicate that the new version is faster than the Prolog based version.

We provide here the results for the old version of VERIFY. The performance figures are provided in Table 1.

TABLE 1. VERIFY tool performance statistics on a Sun 10/21

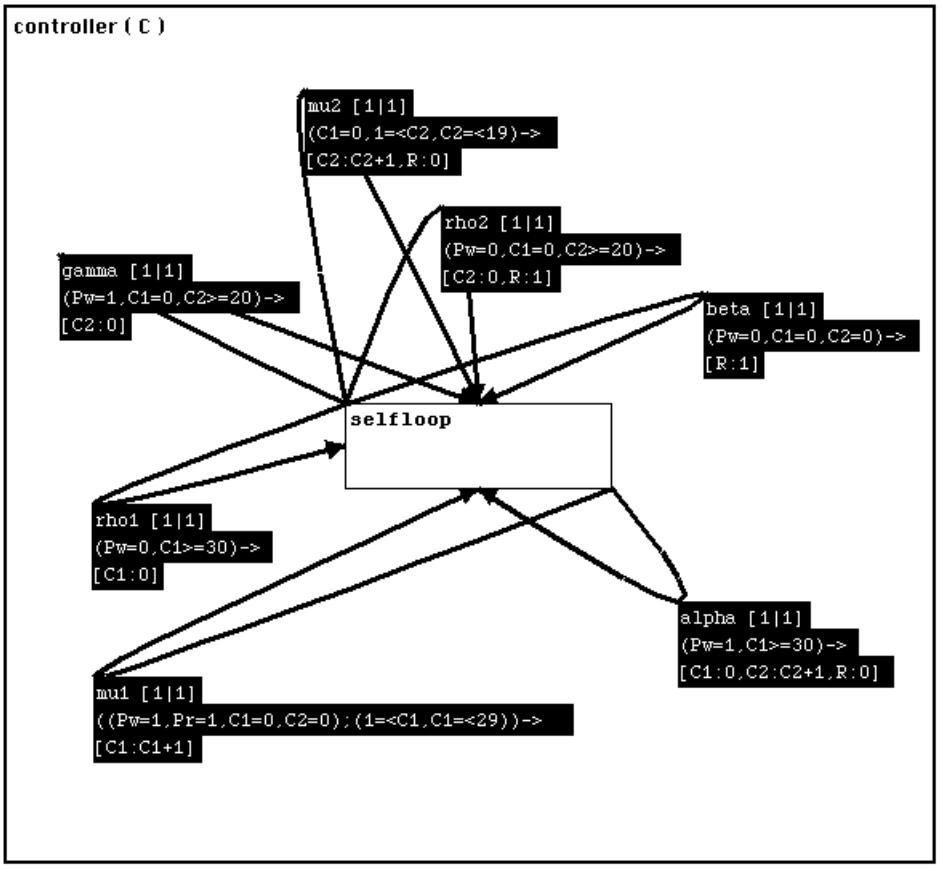
Check	Suggested Controller (Figure 6)	Revised Controller (Figure 8)
Reachability graph generation:	96.4 minutes Unique States: 4,448 Unique Histories: 1,103 Total States: 29,369 Total Edges: 60,361	19.2 minutes Unique States: 3,259 Unique Histories: 602 Total States: 5,902 Total Edges: 11,939
[S1] Open Relay	Property *fails* after 39.8 minutes[a]	*Succeeds* after 2.4 minutes
[S2] Close relay		*Succeeds* after 2.7 minutes
[S3a] No Deadlock		*Succeeds* after 1.9 minutes
[S3b] Watchdog health		*Succeeds* after 3.22 minutes
Total Time	2.3 hours to determine that the pseudocode is *incorrect*.	29.4 minutes to check that the revised controller satisfies its specifications.

a. The current version of VERIFY dumps a compressed version of the reachability graph into a results file on disk. Each time a property is checked, the results file must be compiled in and decompressed by the Prolog verification code. This is very inefficient as the graph should be kept in the RAM memory. The time to consult and decompress the file for S1 is 22 minutes. Hence, with a slight change in the program the actual time to check S1 would be substantially less, viz. 39.8–22=17.2 minutes (rather than 39.8). There are many other inefficiencies in the current code.

The pseudocode suggested for the microprocessor controller (Figure 6) is shown to be incorrect as the property S1 fails to be satisfied. VERIFY indicates where the failure takes place. Based on this debugging information, corrections can be made to the controller. A revised controller is suggested by Lawford in [7] which is obtained by a set of behaviour preserving transformations. The revised controller is shown in Figure 8. This revised controller satisfies all four properties S1,S2,S3a and S3b. The revised controller corresponds to moving the guard $Pressure \geq DSP$ that appears on the first line of the pseudocode down to the fourth line where it guards the increment c2 event μ_1.

It is interesting to note that the incorrect controller has a reachability graph consisting of 89,730 states and edges. The revised controller is much smaller consisting of 17,841 states and edges.

FIGURE 8. Successful controller



3.4 Discussion

The StateTime tool was able to automatically verify that the proposed pseudocode is incorrect and check that the proposed software meets its temporal logic specifications.

The 2.3 hours that it took to detect the failure is obviously much too long to deal with the just under 30,000 states— however, some very preliminary tests on a new version of the verifier indicate that we may be able to achieve significant increases in efficiency while extending the range of properties checked (this work is still at a very early stage of development).

The heuristic algebraic reduction techniques proposed in [7] should be extended to concurrent TTMs and included in the toolset. The algebraic approach

may provide a significant aid to beating combinatorial explosion of states where it can be applied. The pseudocode can be replaced by its reduced order model thereby reducing the size of the reachability graph.

It is important to realize that the delayed trip reactor was a tiny part of a large requirements document. It is not surprising that a switch in one of the guards causes the system to catastrophically fail, and that such an error is easy to overlook.

It can be argued that a competent software engineer would detect the problem, in much less time than it took to do the formal analysis, by walking through the code. However, the more subtle the problem the more likely it is to be overlooked.

Reducing the pseudocode to a TTM. treats the code as a large case statement, reminiscent of Dijkstra-like guarded commands [3]. Let us assume that the software engineer is able (by inspection) to determine that there is an unnecessary dependency on pressure in the guard of α in Figure 6. The controller does not open the relay unless $(Pr = 1)$, whereas the specification (S1) requires that only the power must be checked. The only other transition that can open the relay is μ_2 — but μ_2 must be preceded by α (which is not taken unless the pressure is also critical). The following fix is therefore proposed: remove the conjunct $(Pr = 1)$ from the guard of α. However, when applying the VERIFY tool to this fix, the system was still shown not to satisfy its requirements.

By applying the formal methodology illustrated in this paper we are still by no means certain that the revised controller is correct. Perhaps there are additional RTTL specifications that should be added to the list (e.g. we did not check (EQ 6)).

However, we are able to increase our confidence in the correctness of the system. RTTL specifications are *incremental*. If after developing a specification, we suddenly realize that the specification is incomplete, the situation can be rectified by adding the missing property to the specification as additional conjuncts. All that is needed is to verify this additional conjunct (the others do not have to be done over again). If all of these conjuncts are shown to be valid, then our conjoined specification also has the property of consistency.

3.5 REFERENCES

[1] D. Coleman, F. Hayes, and S. Bear. Introducing Objectcharts or How to Use Statecharts in Object Oriented Design. *IEEE Transactions on Software Engineering*, 18(1): 9–18, January 1992.

[2] W.-P. de Roever. Foundations of Computer Science: Leaving the Ivory Tower. *EATCS Bulletin*, 44, June 1991.

[3] D. Gries. *The Science of Programmin*g. Springer-Verlag, 1985.

[4] D. Harel, *et al*. Statemate: a Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, 16:403–414, 1990.

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

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

[7] M. Lawford. Transformational Equivalence of Timed Transition Models. Systems Control Group, Department of Electrical Engineering, University of Toronto, TR# 9202, 1992.

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

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

[10] J.S. Ostroff. Deciding Properties of Timed Transition Models. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):170–183, April 1990.

[11] J.S. Ostroff and W.M. Wonham. A Framework for Real-Time Discrete Event Control. *IEEE Transactions on Automatic Control*, April 1990.

[12] J.S. Ostroff. A logic for real-time discrete event processes. *IEEE Control Systems Magazine*, June 1990.

[13] J.S. Ostroff. Constraint Logic Programming for Reasoning about Discrete Event Processes. *The Journal of Logic Programming*, II:3&4, October/November 1991.

[14] J.S. Ostroff. Verification of safety critical systems using TTM/RTTL. *Proceedings of the REX Workshop "Real-Time: Theory in Practice"*, LNCS 600, Springer-Verlag, 1992.

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

[16] J.S. Ostroff. A Verifier for Real-Time Properties.*The Journal of Real-Time Systems*, Volume 4, pages 5–35, 1992.

[17] B. Selic, *et al.* ROOM: An Object-Oriented Methodology for Developing Real-Time Systems. *CASE'92 Fifth International Workshop on Computer-Aided Software Engineering*, IEEE Computer Society Press, 1992.

[18] Tools Fair: Technical Code Generation. *IEEE Software*, p. 75, May 1992.

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

[20] W.M.P. v.d. Aalst. Timed Colored Petri Nets and their Application to Logistics. Ph.D Thesis, Eindhoven University of Technology, 1992.

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