Abstraction and Composition of Discrete Real-Time Systems

Jonathan S. Ostroff[1]
Department Of Computer Science, York University,
4700 Keele Street, North York Ontario, Canada, M3J 1P3.
Email: jonathan@cs.yorku.ca Tel: 416-736-2100 X77882 Fax: 416-736-5872.
Electronic Technical Report — CS-ETR-95-02.

Abstract: This paper extends the TTM/RTTL deductive and model-checking framework for real-time reactive systems with a structured design method using the notions of real-time reactive *modules*, module *abstraction* and module *composition*. Equivalence transformations are used to obtain abstract systems, and a composition theorem is provided for deducing global properties from module specifications. The StateTime tool is used for checking module correctness. Abstraction and composition are applied to an actual industrial example involving the delay reactor trip for a nuclear reactor consisting of three independent microprocessors based controllers. Timing, concurrency, integer data, communication and nondeterminism are all important elements of the problem. While the StateTime tool can verify a single microprocessor controller (under 100k states and edges), the complete 3-version system suffers from a combinatorial explosion of states. By contrast, the proposed design method is able to verify the example, and scales up to larger systems.

Keywords: Real-time reactive systems, formal methods tools, statecharts, temporal logic, modules, abstraction, refinement, composition, model checking.

1. This research was supported with the help of NSERC (National Science and Engineering Research Council of Canada). This paper is an extended version of a paper titled “A CASE Tool for the Design of Safety-Critical Systems”, *Proc. Seventh International Workshop on Computer Aided Software Engineering CASE'95*, IEEE Computer Society Press, 1995.

Table of Contents

1.0 Introduction

Safety critical systems such as flight controllers, nuclear reactors and radiation therapy machines are reactive systems involving high levels of natural concurrency, real-time constraints, nondeterminism and communication. Such systems must operate reliably as lives, the environment and property are at stake.

As more software is used to control safety critical systems, thus replacing standard hardware safety interlocks, more accidents are likely to occur. For example, when the designers of the Therac-25 radiation therapy machines eliminated standard hardware safety mechanisms, bugs in the software and other engineering problems resulted in seven accidents involving massive radiation overdoses and four deaths [19].

Reliability of life-critical applications is often translated into a probability of failure on the order of 10^{-9} in a one hour mission (this is called ultrareliability). Quantification of ultrareliability is infeasible using statistical testing methods. For example, life testing on ten specimens of the software would take 114,155 years to achieve ultrareliability [5,20].

In response to the need for quality assurance, some regulatory agencies now recommend (e.g. the European Space Agency) and some even require (e.g. interim standard UK MoD 00-55) the use of formal methods, i.e. the use of applied mathematics and logic for specification, design and verification [3]. North American agencies have not yet on the whole required the use of formal methods to the same extent as their European counterparts. However, we can expect to see this change in the future. For example, the Atomic Energy Control Board of Canada mandated the use of formal specifications and reviews of the Darlington nuclear reactor [8,33].

Modern conceptions of formal methods do not claim unequivocal correctness, but are concerned with a balanced life-cycle approach to assurance. Thus, the use of formal methods complement but do not replace standard software engineering practice including disciplined design, documentation, testing and review.

Formal methods help provide precise system specifications free of implementation detail, as well as the ability to *calculate* with specifications. For example, having specified a telephone directory, one might want to “challenge” the specification with a putative theorem that asserts that adding a phone number, and then deleting it returns the original directory. Calculation, in turn, is only practical if there are tools available to the designer to assist and automate the application of a formal method. Manual proofs of complex theorems or hand exploration of large state spaces is just not feasible. Without the appropriate tools, it is doubtful that the full benefits of formal methods can be achieved.

TTM/RTTL is a comprehensive framework for the specification, development and verification of real-time reactive programs and devices found in embedded, safety critical, or concurrent systems. The framework consists of a generic computational model called timed transition models (TTMs), an abstract specification language called real-time temporal logic (RTTL), and a deductive proof system [24,31]. The framework has heuristics, which have been mechanized using constraint logic, for aiding the designer in the systematic development of infinite state systems, and decision procedures for automatic verification of finite state systems (model-checking). A toolset called StateTime [27,28,29,30] provides automated support for visual specification, simulation and verification in the framework.

A major dichotomy in formal methods is between the use of *proof methodologies* (e.g. [6,22,32]) and by contrast smart *state-space exploration* (e.g. using partial ordering, model-checking and symbolic execution [4,7,11]). Usually, state space exploration is suitable for dealing with small finite state systems (up to a million states), while proof methods must be used on infinite state systems. In StateTime, theorem proving and model-checking are only loosely combined. Much tighter integration has been proposed in other tools [14,21,34].

While the combination of theorem proving and model-checking is an important component of tool support, perhaps the most important need is the ability to structure a large system into smaller parts, and the ability to abstract out the important detail, while ensuring that whatever is computed about the abstraction can also be asserted of the original part.

1.1 Purpose and scope of this paper

Early work in formal methods produced proof systems or model-checking methods that could only be applied to *closed* systems, i.e. systems in which the complete program together with its environment is fully specified. Structured systems require the ability of these proof methods to work on *open* systems, i.e. systems in which only part of the behaviour is specified.

In this paper, we extend the TTM/RTTL deductive and model-checking framework to deal with open real-time reactive systems. The notions of a real-time reactive *module*, module *abstraction* and module *composition* are defined, allowing for the structured development of complex systems. Equivalence transformations are used to obtain abstract systems, and a composition theorem is provided for deducing global properties from module specifications. The StateTime tool is used for checking module correctness.

Abstraction and composition are applied to an actual industrial example involving the delay reactor trip (DRT) for a nuclear reactor involving the use of three independent microprocessors that check sensor readings, with the final decision to shutdown based on a majority vote. Timing, concurrency, communication and nondeterminism are all important elements of the problem. While the StateTime tool can verify a single microprocessor controller (under 100k states and edges), the complete 3-version closed system suffers from a combinatorial explosion of states. However, the modular version of the problems can be verified using abstraction and composition. Although the StateTime tool was originally intended for closed systems, this paper shows how to use it in the open modular setting. We summarize below some of the main concepts:

- (*Real-Time Modules*). A real-time reactive module has an *interface*, a *body* and a *specification*. The *interface* specifies input and output variables and their modes of interaction with the environment. The *body* is specified as a timed transition model (TTM). The *specification* is a formula in real-time temporal logic (RTTL) of the module behaviour.
- (*Abstraction*) A major technique for reducing system complexity is the notion of abstracting out irrelevant behaviour while preserving the behaviour of interest. The designer must be allowed to move between levels of abstraction of systems (e.g. via equivalence preserving transformations) to allow comparison of two systems. Such

flexibility would allow the designer to project out extraneous behaviour to obtain *abstract* high level modules, or conversely, to *refine* high level modules into workable implementations.

- (*Composition*) The properties of a complex system should be deducible from the specifications of its component modules, without any further information about the internal structure of these modules.

The modular approach proposed in this paper is based on the TTM/RTTL framework [24,31] for closed systems. The notion of a real-time module is an extension of the reactive modules defined in [22]. The notion of module abstraction and equivalence preserving transformations is based on the work in [16,17].

The StateTime tool allows for the construction of TTMcharts (that are automatically translated into TTMs) which are similar to statecharts [9]. This visual approach to specifying systems is appealing to engineers, who may be resistant to the use of process algebras or formal logics (e.g. Z, Hoare logic, weakest preconditions, VDM, B, and temporal logic). The visual approach to specification was found useful in the design of the FAA mandated Traffic Alert and Collision Avoidance System [18].

TTMcharts differ from statecharts. The (non-blocking) broadcast communication in statecharts is replaced by (blocking) synchronization as in the Ada rendezvous or CSP message passing. A richer class of timing properties can be directly expressed in TTMcharts than in an available statechart tool called Statemate [10]. An event τ can have a closed time interval as a firing condition (e.g. $\tau[3, 7]$), or be spontaneous (e.g. $\tau[0, \infty]$). A spontaneous transition may occur at any moment or never. Statemate [10] allows for deadlock detection and reachability analysis. However, it does not allow for real-time temporal logic model-checking as in StateTime.

There are two important features of StateTime that make it useful in system design. Firstly, the finite state verifier need not know the time bounds of specifications *a priori*. For example, consider the property $p \Rightarrow \Diamond_{[2, 4]} q$ which states that 2 to 4 ticks after every p-state there must be a q-state. To check this property, the verifier is supplied with p and q (but not the time bounds). The verifier then returns the minimal and maximal bounds for which the property is valid. Thus if any changes are made to the system, the verification can be resubmitted and the new bounds are obtained automatically. Further, if a property fails to hold, a failing computation can be obtained to debug the design. Second, any TTMchart can be executed or simulated, even if it is incomplete. The execution facility is important for early sanity checks of the system under design.

Other compositional systems for real-time systems include the assertional and CSP style proof system of [13], which now has tool support [12] for the assertional style. A composition rule (stronger than the one in this paper) for dealing with assumption/guarantee specifications is provided in the TLA formalism of [1]. A major difference between TLA and the TTM/RTTL approach is that in TLA both programs and specifications are written as formulas in temporal logic. The TTM/RTTL approach is by contrast a dual approach — programs are specified visually as TTMs, and specifications are represented in the RTTL logic. Time is specified in RTTL using special purpose labelled temporal operators, whereas in TLA time is a variable.

1.2 The design method

Abstraction and composition provide a means for structured design. We provide below a summary of the main features of the design method, using the delay reactor trip (DRT) to illustrate the main principles.

A *computation* is an infinite sequence of states describing a possible execution of a module. The behaviour of a module m is completely described by the set of all its legal computations. There are two ways to describe the set of module computations: either (a) by the timed transition model $\mathbf{T}(m)$ (associated with the module m), or (b) by the real-time temporal logic specification $\mathbf{S}(m)$ of the module. Both $\mathbf{T}(m)$ and $\mathbf{S}(m)$ fully specify the legal computations of the module. The TTM description is a lower programming level description, whereas the RTTL description is more abstract. Hence, the global requirements are normally specified in RTTL, while implementation is more conveniently performed using TTMs.

In a single language framework, both programs and specifications are formulas in logic. To prove that m' is an abstraction of m (or alternatively m is a refinement of m') we need only prove the validity of the temporal logic formula $\mathbf{S}(m) \rightarrow \mathbf{S}(m')$. This works if module specifications such as $\mathbf{S}(m)$ and $\mathbf{S}(m')$ allow for arbitrary behaviour on the part of the environment including "stuttering" steps that leave the state unchanged. In a dual language framework such as TTM/RTTL, there is an additional freedom to deal with the TTM $\mathbf{T}(m)$ associated with the module. We may thus use intuitively appealing equivalence transformations to obtain the more abstract TTM $\mathbf{T}(m')$.

The parallel composition $m_1 \| m_2$ of two modules is represented in logic by the conjunction $\mathbf{S}(m_1) \wedge \mathbf{S}(m_2)$. Composition is conjunction. In the dual language framework, there is the additional flexibility of retaining the process oriented description $m_1 \| m_2$, computing an abstract simpler module m' equivalent to m_1 thereby obtaining $m' \| m_2$. The module m' may be model-checkable for its specification $\mathbf{S}(m')$ (whereas m_1 may be too big too check). The flexibility to use both process oriented TTM descriptions and logic based RTTL formulas, allows the designer to choose the most concise format according to need. A dual framework retains the flexibility to deal with processes or their logical equivalent, but must then provide rules to calculate with both.

The design method used on the DRT example is as follows:

- Model the *plant* (relay, pressure and power sensors, etc.) as a module whose body is a TTMchart.
- Model the *controller.* In the case of the DRT, the *controller* module consists of three independent microprocessors (each modelled as a module), and the majority voting algorithm (yet another module). A pseudocode outline of the proposed microprocessor code is supplied by the manufacturer of the reactor from which the body of the microprocessor modules as TTMcharts can be constructed. The plant is that part of the system that is a given. The controller corresponds to that part of the system that must be designed.
- The system under design (sud) is defined as $sud = plant \| controller$.
- State the global requirements for sud as an RTTL formula **R**.

- Verify that *sud* satisfies **R**. The reachability graph of *sud* is too big for the StateTime verifier to check. Hence, the various modules must be checked independently for compliance to their module specifications. Where a module is too big to be model-checked, the equivalence preserving transformations can be used to simplify the module. Finally, the global requirement **R** must be shown to follow from the individual module specifications.
- The above approach is bottom-up as the modules of the controller were already known, and the only question is whether the complete system satisfies its global requirements. If the pseudocode had not been given, then we could do a top-down design starting with the abstract version of the pseudocode for each microprocessor, and then using the equivalence transformations to obtain the implemented code.

The rest of this paper is organized as follows. Section 2 summarizes the features of the StateTime tool, TTMs and RTTL. Section 3 presents the closed system verification of the DRT using a single controller. The closed system verification shows how the StateTime tool is used, and provides a benchmark with which to compare the modular approach. Section 4 defines the notion of a real-time reactive module. Sections 5 and 6 define the notions of module abstraction and composition respectively. In Section 7, the 3-version DRT is verified using the modular design method. Section 8 presents conclusions and a discussion of future work.

2.0 The StateTime toolset, TTMcharts and RTTL

The StateTime toolset [27] consists of various tools for designing real-time safety critical systems. The *Build* tool is used to construct TTMcharts, and the *Verify* tool can model-check these charts for various properties specified in real-time temporal logic. The *Develop* tool [26] is used as a theorem prover for infinite state systems.

A typical chart built with the StateTime tool is shown in Fig. 1, where hierarchy (clustered activities, default states and XOR-composition), concurrency (AND-composition), synchronization (shared events), and timing are illustrated. The TTMchart can be mapped into a TTM. A TTM consists of a set of variables (activity and data variables), an initial condition, and a set of transitions corresponding to the chart events. The transitions of the TTM corresponding to the chart of Fig. 1 are shown in Fig. 2. Transitions have an enabling condition, transformation function and time bounds, on the basis of which the formal semantics and timed reachability graph of the TTM can be defined [25].

What is called a “state” with respect to statecharts is called an *activity* in TTMcharts. This is because the term state is used in TTMs to refer to a global snapshot at any instant of all the activity and data variables of a chart.

A *computation* of a TTMchart is an infinite sequence $s_0 s_1 s_2 \ldots s_i \ldots$ of global states s_i, starting in an initial state, with successor states computed by the transition of enabled events. The tick transition occurs an infinite number of times in such a computation. The tick transition always eventually fires even if there are no other eligible events. All other transition time bounds are given relative to number of occurrences of the tick transition. For example, each time a transition $\tau[2, 4]$ becomes and remains enabled, it is pending execution until two clock ticks have occurred. Thereafter, before the fifth clock tick, either τ must be taken or become disabled.

FIGURE 1. Example of a TTMchart m=m1||m2

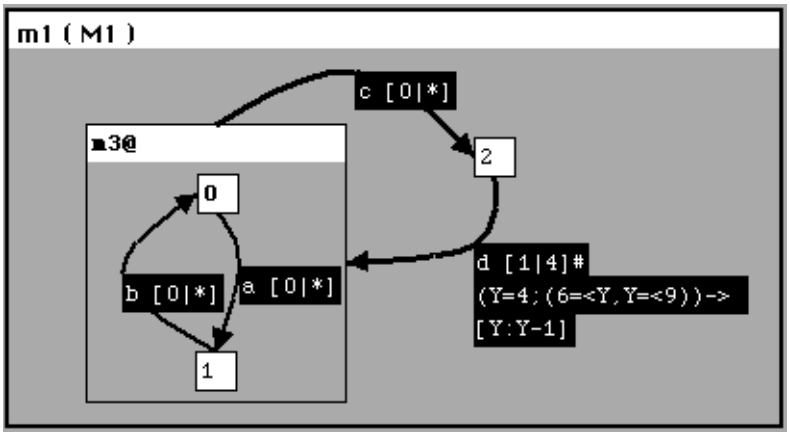
TTMcharts can be developed top down or bottom up. Working top down, the root activity m is AND-decomposed into subactivities m1 and m2, i.e. m = m1 || m2 (see bottom picture). AND-composition is indicated by dashed boxes . The root activity m is also called a TTMchart.

If we zoom in to the structured activity m1 of the chart m we get the picture directly below. The activity m1 is XOR-decomposed into the structured activity m3 and the leaf activity 2. An activity with internal structure is followed by the "@" symbol. Leaf activities have no internal structure.

Y is an integer data variable. The event d in m1 has a guard (Y=4;(6=<Y,Y=<9)). When d is taken it does the assignment Y:=Y-1, and leads to the default activity **0** of **m3**. The activity 0 is the default of m3, and m3 is the default of m1 (default activities are in bold). In guards such as (Y=4;(6=<Y,Y=<9)), the comma stands for conjunction and the semicolon for disjunction.

The superactivity m3 is an abstraction of activities 0 and 1, describing the common property that event c transforms them to activity 2. Conversely, this can be seen as a refinement: m3 is refined to consist of 0 and 1.

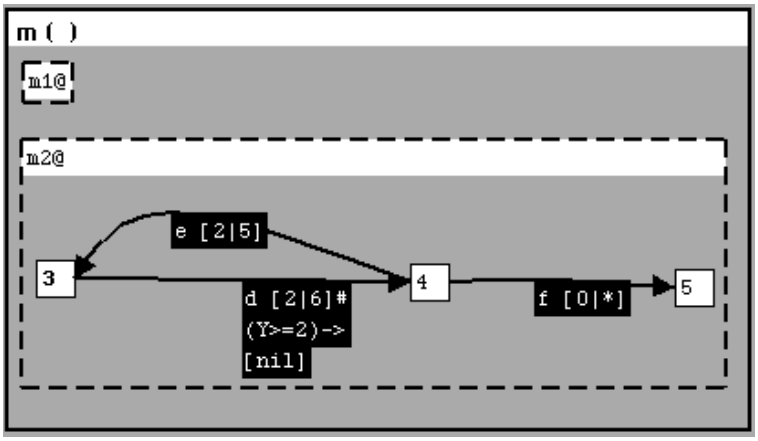
Each structured activity has its own activity variable. Thus M1, M2, and M3 are the activity variables of activities m1, m2, m3 respectively. If execution is at activity 1 then it may do either b or c (nondeterminism).



Each activity variable has its corresponding type that it ranges over, e.g. type(M1) = {m3,2}, and type(M3) = {0,1}.

The event d is declared a shared event (indicated by the symbol "#"), i.e. it will synchronize with any other shared event d in a parallel activity (e.g. see m2 below).

An upper time bound of infinity is denoted by the symbol "*".



The component event d in m2 is declared shared, thus synchronizing with the corresponding component d in m1.

The event e, which is local to m2, is taken after having been in activity 4 for between two and five ticks of the clock, unless it is preempted by event f which can be taken anytime (the upper time bound of event f is infinity).

The shared event d is taken after having been in activities 3 and 2 simultaneously (with both guards continuously true) for between two and four ticks of the clock. The lower bound is the maximum of the two component events in m1 and m2 and the upper time bound is the minimum of the two components, i.e. d[2,4]. A TTMchart can be converted into a TTM. A TTM consists of a set of variables, an initial condition, and a set of transitions. Each transition of the TTM corresponds to an event in the chart (or pair of synchronizing events). A transition has an enabling condition, transformation function, and lower and upper time bounds as shown in the table of Fig. 2.

FIGURE 2. TTM transitions corresponding to the events of chart m in Fig. 1

Transition:	Enabling Condition:	Transformation Func.:	Lower:	Upper:
a	M1=m3,(M3=0)	[M3:1]	0	infinity
b	M1=m3,(M3=1)	[M3:0]	0	infinity
c	M1=m3	[M1:2]	0	infinity
d#	M1=2,(Y=4;(6=<Y,Y=<9)), M2=3,(Y>=2)	[M1:m3,M3:0,Y:Y-1,M2:4]	2	4
e	M2=4	[M2:3]	2	5
f	M2=4	[M2:5]	0	infinity

State-formulas are boolean valued expressions in the activity and data variables. Given activity variables such as M1, M2 and M3 and data variable Y of chart m (see Fig. 1) an example of a state-formula is:

$$(\mathsf{M1} = \mathsf{m3} \wedge \mathsf{M3} = 1) \wedge (\mathsf{Y} < 7) \qquad \text{(Eq. 1)}$$

which asserts that the chart is in subactivity 1 of the clustered activity m3 and the data variable Y is less than 7. The above formula is true in a state <M1:m3, M2:3, M3:1, Y:4>.

RTTL formulas are constructed from state-formulas together with special temporal logic operators such as $\Box$ (*henceforth*) and $\Diamond$ (*eventually*). For example, the chart m (of Fig. 1) can be checked automatically by the Verify tool for the real-time response property:

$$(\mathsf{M1} = \mathsf{m3} \wedge \mathsf{M3} = 1 \wedge \mathsf{Y} < 7) \Rightarrow \Diamond_{[3,\,9]}(\mathsf{M2} = 5)\,. \qquad \text{(Eq. 2)}$$

The symbol $\rightarrow$ is the ordinary propositional conditional connective, whereas the symbol $\Rightarrow$ is the modal *entails* operator. Thus $(p \Rightarrow \Diamond q) \stackrel{\text{def}}{=} \Box(p \rightarrow \Diamond q)$. The formula $p \rightarrow \Diamond q$ asserts that: if p is true in the *initial* state of a computation, then there is some subsequent state in which q holds. The stronger formula $(p \Rightarrow \Diamond q)$ asserts that: if p is true in *any* state of the computation, then eventually q must hold in some subsequent state.

Some examples of real-time temporal logic properties are given below:

[Real-time response $p \Rightarrow \Diamond_{[3,\,5]} q$ **]:** between three and five ticks after every *p*-state there must be a *q*-state (i.e. *q* must be true before the 6-th tick of the clock).

[Exact time $\Diamond_5 p$ **]**: is an abbreviation for $\Diamond_{[5,\,5]} p$, i.e. p is true in exactly 5 ticks. Similarly, $\Diamond_{\leq 5} p$ is an abbreviation for $\Diamond_{[0,\,5]} p$, i.e. p must become true before the sixth tick of the clock from now. The formula $p \rightarrow \Diamond_0 q$ asserts that if the initial state is a *p*-state, then it must be followed by a later *q*-state before the next clock tick; there may be many states between the *p* and *q*-states. The $\Diamond_0$ operator is useful in modular specifications that must allow "stuttering" steps, i.e. steps taken by the environment which leave the module state unchanged.

[Invariance $p \Rightarrow \Box q$ **]:** *p* entails henceforth *q*.

[**Limited invariance** $\Box_{<t} p$ **]:** p holds true up to the *t*-th tick of the clock. Thereafter, *p*'s truth value is unconstrained.

The reader is referred to [24,25,28] for a complete treatment of the TTM/RTTL framework and the Verify tool.

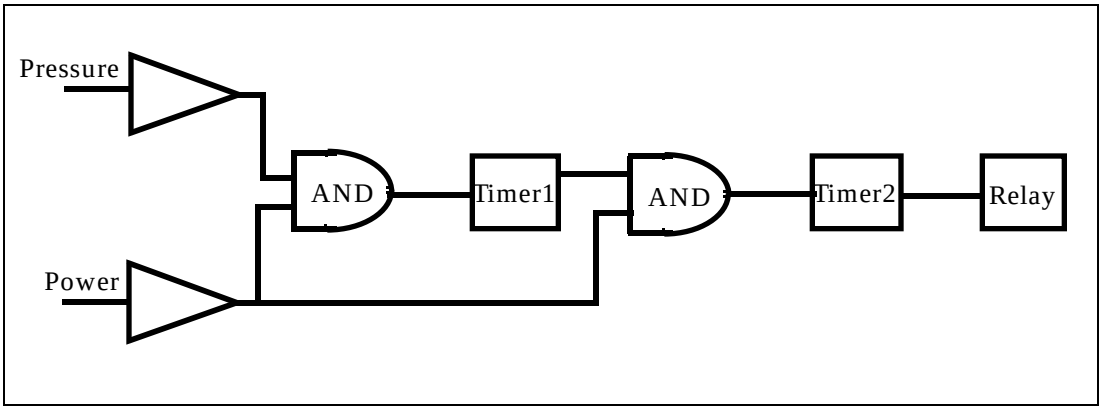
3.0 A nuclear reactor shutdown system

In earlier 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. A semi-formal approach based on function tables was used to certify the software of one such commercial system [33]. In the sequel we present a temporal logic approach to verifying the time critical part of an shutdown system.

The delayed reactor trip (DRT) problem was first described by Lawford *et. al.* [16]. Lawford developed behaviour preserving transformations for TTMs with which he was able to discover a flaw in the proposed design [15]. 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. We will analyze the problem from a temporal logic perspective (RTTL), and will use completely automated verification procedures (and the Verify tool) to check the correctness of the implementation [29].

The delayed reactor trip for the nuclear reactors used to be implemented in hardware using timers, comparators and logic gates as shown in Fig. 3. The new DRT system is

FIGURE 3. Analog implementation of the delay relay trip system DRT (the "controller").



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 reaches 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 system has waited 3 seconds to check the power level again, if the power is below its critical limit, then the system 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. In Sect. 7.0, we design and verify a majority voting version consisting of three microprocessor controllers.

Fig. 7 provides a modified version of the pseudocode taken from the original requirements document[1], which is 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 sampling rate is sufficient to ensure adequate control. In the formal model, one tick of the clock will represent 100ms.

3.1 Formal model of the DRT as a TTM

The Build tool can be used to construct the TTMchart of the complete system under design *sud* (Fig. 4) defined as the parallel composition of the “plant” (relay, pressure and power sensors) and the “controller” (computer with its supervisory program), i.e.

$$sud = plant \parallel control \parallel obs \qquad \text{(Eq. 3)}$$

The chart *obs* is an “observer” that watches the system without interfering with its operation. It is used in the sequel to help verify *sud* (Sect. 3.2.1). The relationship between the variables of the plant and controller are shown in Fig. 5.

1. The pseudocode originally proposed in the actual requirements document was shown to be incorrect using the TTM/RTTL framework [15,29].

FIGURE 4. The TTM specification of the system under design (sud)

sud = plant||control||obs
The activity variable of *control* is F_1 where $type(F_1) = \{normal\_full, fail\}$. See Fig. 6 for further details of the plant. The internal structure of *normal_full* in the control chart is described in more detail in Fig. 8.

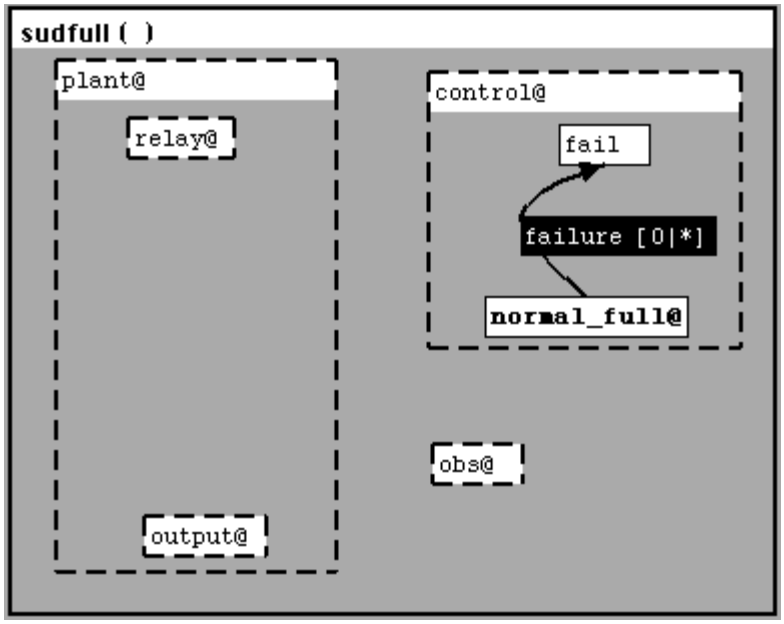
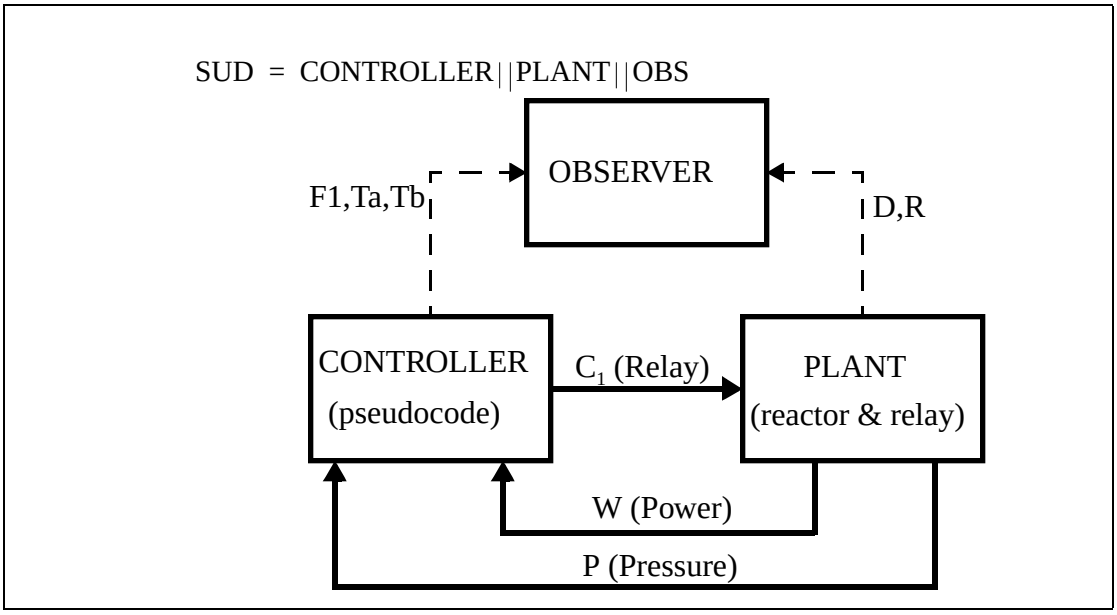


FIGURE 5. Data flow diagram for the plant, controller and observer



The *plant* is itself composed of various charts, i.e. $plant = relay \| output$ (Fig. 6). The plant chart specifies that the power and pressure sensor values are updated every two ticks of the clock (thus capturing the assumption that signals are filtered).

The controller executes the pseudocode of Fig. 7, every one tick of the clock. In addition it is possible for the controller to fail (in which case $F_1 = 0$). The pseudocode makes use of two integer counters *Ta* and *Tb* for the two timeouts of 30 and 20 clock ticks respec-

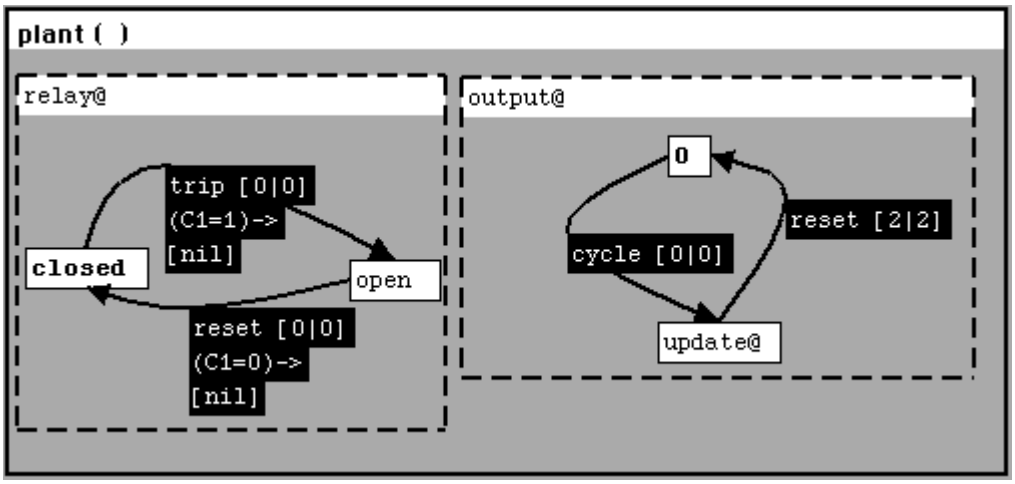
FIGURE 6. The DTR plant specification as a TTM

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

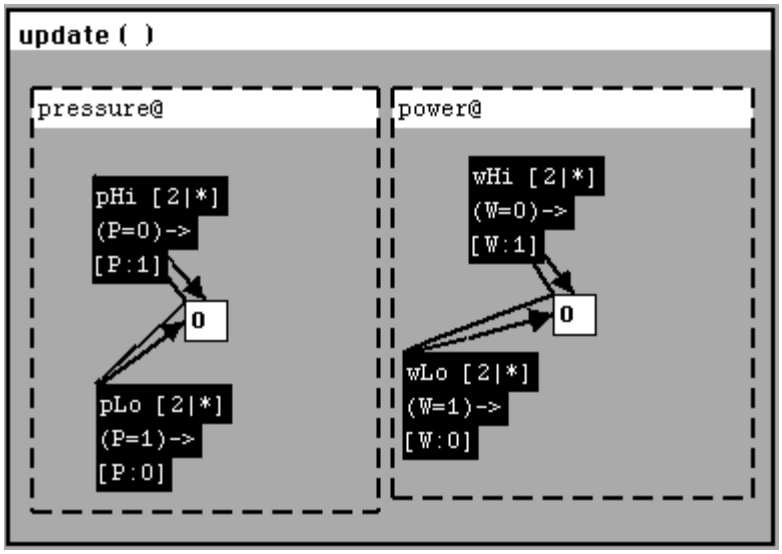
where *R*=relay position, *C1* =command to move relay, *D*=activity variable of *output*, *P*=pressure and *W*=power.

The relay is opened (closed) immediately upon receiving the *C1* command to do so. When *D*=0, the last sensor updates have just been made and there are two ticks to the next change.

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



Inner Structure of update@ — pressure P and power W are updated once every two ticks



tively. The pseudocode is in turn translated into an equivalent TTM *full_controller* (Fig. 8) [16].

3.2 Global System Requirements

The first requirement may be written in real-time temporal logic (RTTL) as:

$$\textbf{R1: } \Box\neg fail_1 \rightarrow [bothHi \wedge \Diamond_{30} powerHi \Rightarrow \Diamond_{[30,\,32]}\Box_{<20}(R = open)] \qquad \text{(Eq. 4)}$$

where the precise definitions of $fail_1, bothHi, powerHi$ are given below (Eq. 5). The above formula asserts that provided the controller never fails, whenever a critical high is

FIGURE 7. Pseudocode for the computer to control the DRT

Every one tick Do:
If W=1 /* power is high */ **then** CodeA **Else** CodeB

CodeA:

```
If counter Ta is reset then
    If counter Tb is reset then
        If P=1 {pressure is high} then
            increment Ta {Transition : μ1}
        EndIf
    Else
        If counter Tb has timed out then
            reset Tb {Transition : γ}
        Else
            increment Tb {Transition:  μ2}
            open Relay
        Endif
    Endif
Else
    If counter Ta has timed out then
        open Relay {Transition: α}
        reset Ta
        increment Tb
    Else
        increment Ta {Transition : μ1}
    Endif
```

CodeB:

```
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 {Transition : μ2}
            open Relay
        Endif
    Endif
Else
    If counter Ta has timed out then
        reset Ta    {Transition : p1}
    Else
        increment Ta {Transition : μ1}
    Endif
```

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 for 20 ticks.

How should the predicates *bothHi* and *powerHi* be defined? The intuitively obvious definition $bothHi \stackrel{\text{def}}{=} (P = 1 \wedge W = 1)$ is not valid because the controller cannot respond to instantaneous changes in the sensor values. The clause $(D = 0)$ must therefore be a conjunct of *bothHi*, thus indicating that the antecedent is measured from a state from which the power and pressure remain constant for two clock ticks (see Fig. 6). Alternatively, we could have written $\Box_{<2}(P = 1 \wedge W = 1)$ to indicate that the pressure and power must remain high for at least two clock ticks.

The controller cannot respond to a critical high if it is in the middle of counting its timeouts — thus the controller initialization clause $init(control)$ is a conjunct of *bothHi* in the antecedent.

The correct definitions for (Eq. 4) are thus given by:

$$\begin{aligned} init(control) &\stackrel{\text{def}}{=} (F_1 = normal\_full \wedge T_a = 0 \wedge T_b = 0) \\ fail_1 &\stackrel{\text{def}}{=} (F_1 = fail) \\ bothHi &\stackrel{\text{def}}{=} init(control) \wedge R = closed \wedge D = 0 \wedge (P = 1 \wedge W = 1) \\ PowerHi &\stackrel{\text{def}}{=} (D = 0 \wedge W = 1) \end{aligned} \qquad \text{(Eq. 5)}$$

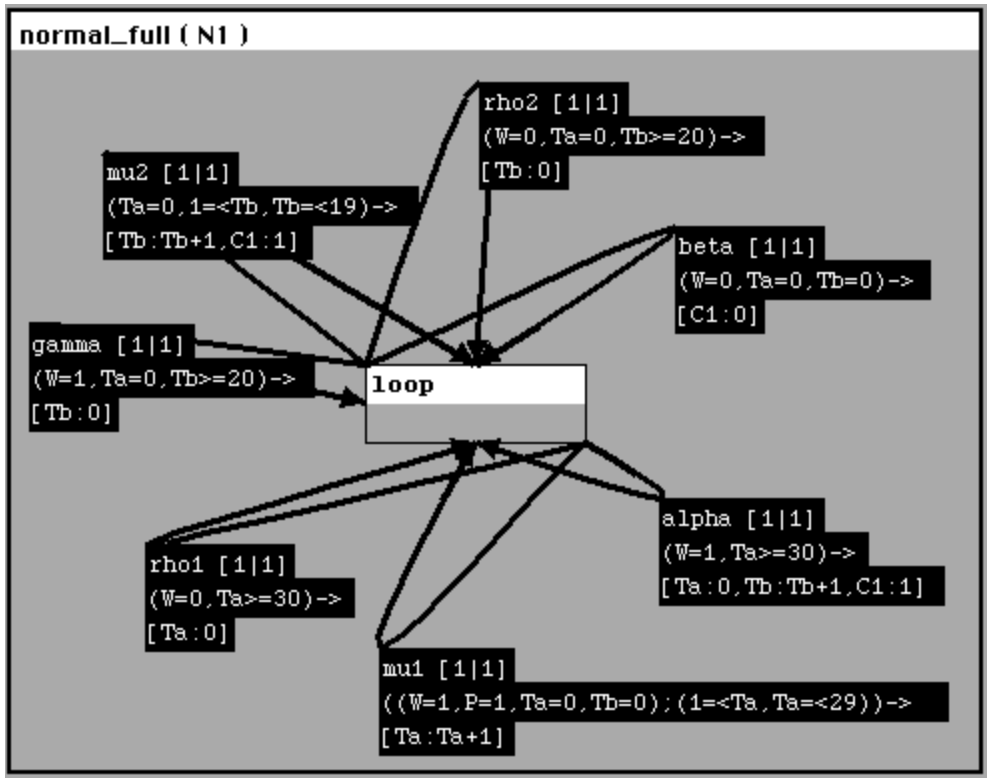
The stateformula $init(control)$ must satisfy the requirement

$$\textbf{[R3]}: \Box\neg fail_1 \rightarrow [\neg init(control) \Rightarrow \Diamond_{\leq 52} init(control)] \qquad \text{(Eq. 6)}$$

[R3] is required so as to ensure that the controller does not deadlock under normal operation. The second requirement is specified as:

FIGURE 8. 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.



$$\textbf{[R2]}: \Box\neg fail_1 \rightarrow [powerLo \Rightarrow \Diamond_{\leq 2}(R = closed)] \qquad \text{(Eq. 7)}$$

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

3.2.1 Checking the correctness of the specifications.

Once the Build tool has been used to construct the complete model, the requirements must be proven correct. The Verify tool is used for this purpose. The Verify tool can model-check temporal logic properties such as invariance ($p_1 \Rightarrow \Box p_2$) and real time response ($p_1 \Rightarrow \Diamond_{[l,u]} p_2$). An important feature of the verifier is that the lower and upper time bounds *l* and *u* in the real-time response property need not be specified — the tool will figure out the minimal and maximal bounds respectively for a specification pair p_1, p_2. This ability of the verifier to return the bound values is useful as it saves the specifier from having to know in advance what the timing constraints are. For example, the RTTL property of *sud* given by

$$(init(control) \wedge D = 0 \wedge W = 0) \rightarrow \Diamond_{\leq 2}(R = closed \vee F_1 = fail) \qquad \text{(Eq. 8)}$$

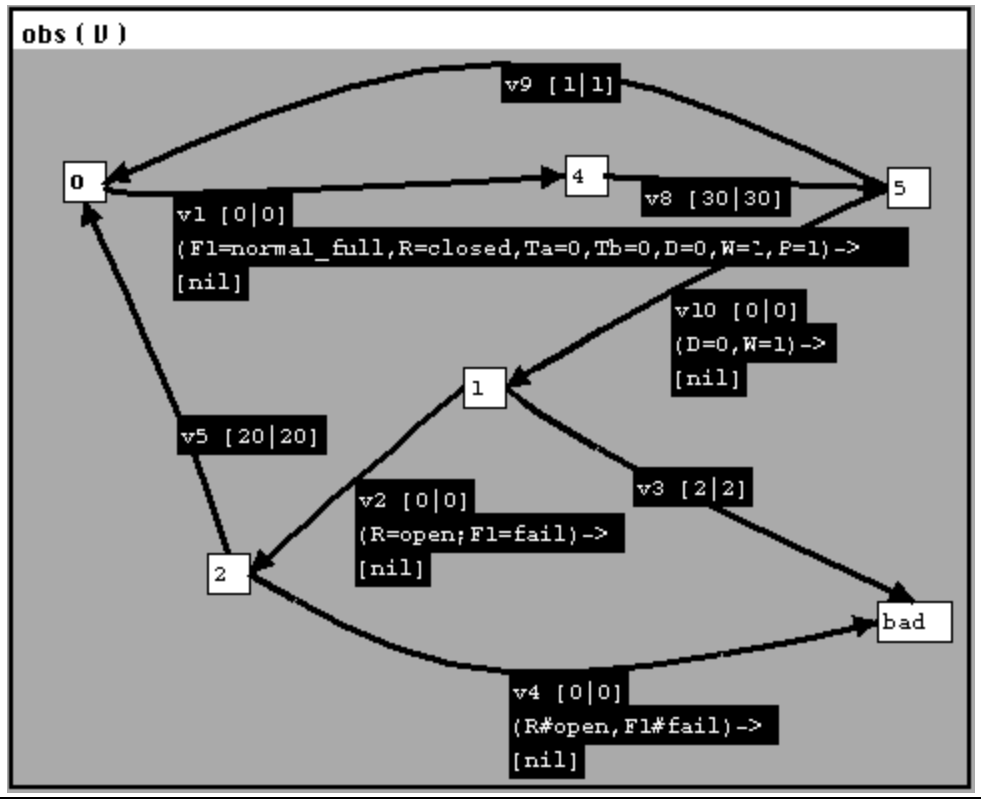
can be submitted to the verifier (without having to specify any bounds). The verifier then returns the bounds [0,1]. We then know by temporal logic reasoning that (Eq. 8) is also true because its bounds of [0,2] are less stringent, and in fact the more stringent consequent $\Diamond_{\leq 1}(R = closed \vee F_1 = fail)$ can be used instead.

The temporal logic theorem $[p_1 \Rightarrow \Diamond_{\leq 2}(p_2 \vee p_0)] \rightarrow [\Box\neg p_0 \rightarrow (p_1 \Rightarrow \Diamond_{\leq 2} p_2)]$ entails that (Eq. 8) implies the validity of (Eq. 7) — hence requirement [R2] is valid. Requirement [R3] can be checked in a similar fashion.

The verifier can check a small but important set of RTTL properties, but requirement [R1] in (Eq. 4) is not one them. However, an observer can be constructed that will detect all behaviour that is a counterexample to the property to be verified. The chart *obs* (Fig. 9)

FIGURE 9. Observer to check requirement 1

The activity 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. 4) is false. The events *v1* and *v10* detect when the antecedents *bothHi* and *PowerHi* of (Eq. 4) 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.



moves to a “bad” state when it observes violations of the property. The validity of [R1] can then be demonstrated by model-checking the validity of $\Box(V \neq bad)$. 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 use of an observer increases the size of the reachability graph. However, any model checking algorithm for arbitrary formulas will also add additional complexity to the verification.

3.2.2 Incremental specifications and putative challenges

The approach of (a) building a model (the TTMchart), (b) stating the global requirements (in RTTL), and then (c) verifying that the model satisfies its requirements (i.e. formal correctness), 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 the best of our abilities. The StateTime tool does provide important methods for validating the model, including simulation, challenging the system with putative theorems, and incremental specification.

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 regenerate the reachability graph. For example, the three requirements for *sud* presented above are incomplete. An additional property that must be satisfied is: "the relay should not open unnecessarily" — which is given by the *unless* (or "waiting-for") property

$$\textbf{[R4]:}\ (R = closed \wedge W = 0) \Rightarrow (R = closed)\,\mathcal{W}(W = 1 \wedge P = 1) \qquad \text{(Eq. 9)}$$

The above property was submitted to the verifier and found to be valid. Such *unless* property can be checked directly by the verifier without the need for an observer.

A useful validation method involves posing a "challenge" to the system with putative theorems. Thus, if the model is correct, then such and such a property should hold. The fact that the relay should not open unnecessarily (Eq. 9) 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 trajectory, which is useful for debugging and correcting the problem.

Another important method of validation, is to use the simulation capability of the Build tool. 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.

4.0 Modules

In the previous section, the complete system under design was checked for correctness. As the systems get larger, so the problem of combinatorial explosion of states arises. For example, when a system of three independent controllers with majority voting control is designed, then the reachability graph is too large for the verifier to handle. It is in such situations that a modular approach together with abstraction and composition are vital.

In a modular approach, a system is composed of modules, and the global properties of the system are checked by verifying each module, without having to consider the complete system all at once. Such an approach usually reduces the size of the reachability graphs that must be generated, and ameliorates the problem of combinatorial explosion of states.

A module m is defined by its *interface* $\mathbf{I}(m)$, *body* $\mathbf{B}(m)$ and *specification* $\mathbf{S}(m)$:

1. The *interface* $\mathbf{I}(m)$ identifies the variables $\bar{y} = y_1, \ldots, y_k$ that are shared between module m and other modules. The types and modes of these variables must be identified. The mode of a variable includes whether it can be written to (**out**) or only read (**in**) by the module. If the module m has a declaration **out** y_1, then no other module in the envi-

ronment of *m* may have a writing reference to y_1. If two (or more) modules each write to *y*, then they must each have the declaration **external out** y_1, thus indicating that the external environment may also change y_1.

2. The *body* **B**(*m*) is a program that implements the module specifications. The program statements may refer only to variables declared *local* to the body, or to variables in the interface. Usually, the program is represented as a TTMchart. The initial condition of the module is written $init(m)$, and is the conjunction of all the initial conditions declared on both local and interface variables.

3. The *specification* **S**(*m*) of the module is an RTTL formula whose only free variables are in the interface. The specification specifies the expected behavior of the module. It is ultimately intended that the specification be *modularly-valid* (the precise definition is given in Sect. 4.1), i.e. hold true no matter what the behaviour of the external environment is.

Let $\bar{y} = y_1, \ldots, y_j, y_{j+1}, \ldots, y_k$, where $y_1, \ldots, y_j$ are the **external in** or **external out** variables (all the variables whose value may be interfered with by the environment), and where $y_{j+1}, \ldots y_k$ are the remainder (the **out** variables). We often refer to the module as $m(y_1, \ldots, y_j; y_{j+1}, \ldots y_k)$, where the semicolon divides the interfering variables from those that cannot be interfered with from the environment.

The reactor trip relay module *relay* is shown in Fig. 13. When the command to open the relay ($C = 1$) comes from the environment, then the relay is immediately opened ($R = open$) thus shutting down the reactor. The specification of the relay (Eq. 10) does not contain any *immediate* operators such as the next operator $\bigcirc$ in the consequent; instead, the operator $\diamondsuit_0$ is used. This is because the computations of a module may have stuttering steps that leave the state unchanged. Specifications must therefore allow such stuttering steps.

The following definition ensures that two modules communicate with each other in a way allowed by their respective interface declarations:

> **Definition 1:** Two modules are *interface compatible,* if on their shared variables they agree on type, their initial conditions are not inconsistent, and if one of the module declarations specifies an **out** mode, then the other specifies an **external** mode.

4.1 The computations of a module

Given a chart *M*, its corresponding TTM is denoted **T**(*M*). The computations of **T**(*M*) are infinite sequences of states such as

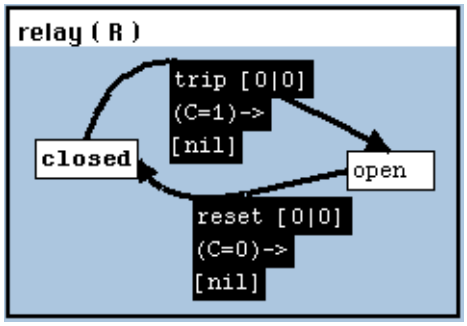
$$s_0 \xrightarrow{a} s_1 \xrightarrow{b} s_2 \xrightarrow{\text{tick}} s_3 \ldots, \text{ which we will also write as } [s_o,a][s_1, b][s_2, tick]\ldots$$

where $a, b, \ldots, tick$ are taken from the transition set of the TTM. Each computation must satisfy five requirements that define the legal behaviour of a TTM [24]. For example, the ticking requirement specifies that there must be an infinite number of tick transitions in any computation (i.e. time always advances). The set of all these computations defines the timed behaviour of the TTM.

In order to define the computations of a module *m*, we may add to the transition set of a chart *M* associated with the module body, an *environmental* transition τ_E, that pledges to

FIGURE 10. Relay module $relay(C;R)$ **for the DRT**

module relay
INTERFACE
external in C: {0,1} where initially $(C = 0)$ /* when the input command *C=1* is given, the relay is opened, and when *C=0* the relay is closed */
out R: {open, closed} where initially $(R = closed)$ /* R is the relay actvity variable that is exported as output */
BODY



SPECIFICATION

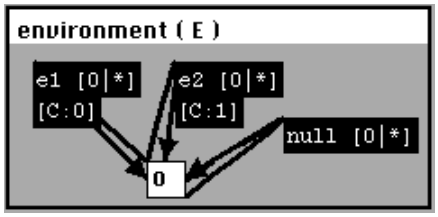
$$\mathbf{S}(relay(C;R)): \begin{bmatrix} \Box_{<t_1}(C = 1) \Rightarrow \Diamond_0 \Box_{<t_1}(R = open) \\ \Box_{<t_2}(C = 0) \Rightarrow \Diamond_0 \Box_{<t_2}(R = closed) \end{bmatrix} \text{ for any time constants } t_1, t_2 \geq 1 . \qquad \text{(Eq. 10)}$$

/* **Informal description:** The operator $\Diamond_0$ is needed in the consequent. Although the relay responds to a stimulus (i.e. a change in *C*) before the next clock tick, the reponse is not immediate but may occur a few states later (as actions of the environment are interleaved with actions of the relay). */
end module relay.

maintain the value of all local and non-external variables of the module. The transition τ_E is arbitrarily interleaved with the other transitions of the body chart, and may make arbitrary changes to any of the external variables. For example, a computation of the module *m* might look like: $[s_0, a][s_1, \tau_E][s_2, b][s_3, \tau_E][s_4, tick][s_5, \tau_E]\ldots$.

A transition is a 4-tuple consisting of an enabling condition, transformation function, and lower and upper time bounds. In general the transition τ_E is a nondeterministic function on the state-space Q, i.e. $h_{\tau_E}: Q \rightarrow 2^Q$, where h_{τ_E} is the transformation function of the environment transition. For example, in the case of the module *relay* (Fig. 13), the successor state to τ_E can (nondeterministically) have the module interface variable *C* set to either zero or one.

Although the Build tool only allows for deterministic transition functions in assignment format, nevertheless different transitions are executed nondeterministically. Hence, the *environment* module of *relay* is given by the chart:



The environmental chart has two spontaneous transitions *e1* and *e2* that may change the value of *C* at any instant. In addition, the environment may at any time execute the *stuttering* transition *null*. The stuttering transition pledges to leave unchanged all local and exter-

nal module variables, but may change any variables in the environment. Thus, when a stuttering transition occurs, in effect the module state is left unchanged. The three transitions *e1*, *e2* and *null* capture the nondeterministic behaviour of the original environmental transition τ_E.

> **Definition 2:** Let a module *m* have body chart *B(m)* and environmental chart *E(m)*. Then the TTM $\mathbf{T}(m)$ *associated with module m* is given by $\mathbf{T}(B(m) \parallel E(m))$.

Thus, the open module *m*, may be replaced by the closed system $\mathbf{T}(m)$ taken by composing the body chart with its environment. All the tools developed for closed systems may now be applied to the analysis of $\mathbf{T}(m)$ in the standard fashion.

The body chart *B(m)* may be replaced by any other chart (program) provided that the module with the new body has the same timed behaviour as the old with respect to the interface variables. The definition of modules given in this section allows for a *separation of the concerns* between the module user (i.e. systems analyst) and its implementor (i.e. the programmer).

- The module user specifies the module interface and specification — the module body can be ignored at the systems level.
- The implementor refines the module by programming a body that satisfies the module interface and specification. The body can be changed at any time provided it continues to satisfy the interface specification, without impacting the systems level.

5.0 Module abstractions

Abstraction is the notion of projecting out irrelevant behaviour while preserving the behaviour of interest, thereby reducing system complexity. The ability to move between levels of abstraction allows the designer to obtain *abstract* high level simple TTMs, or conversely, to *refine* high level TTMs into workable implementations.

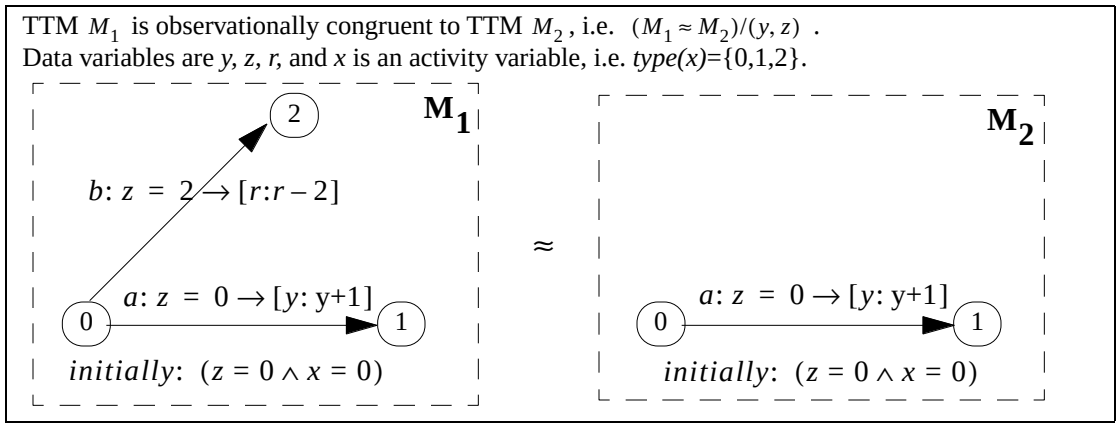
In this section, we apply the algebraic equivalence of TTMs developed in [16,17], and corresponding behaviour preserving transformations to define a notion of module abstraction. Module abstraction is then used to reduce the combinatorial explosion of states in the verification of the DRT example by an order of magnitude.

A *transformation* is a rule that changes a TTM M_1 into a new TTM M_2 without altering the timed behaviour of the TTM over a set of variables of interest $\bar{w}$. As an example, consider the transformation rule **TA/TD** — Transition Addition/ Transition Deletion — that transforms TTM M_1 into module M_2 (Fig. 11).

The initial condition of M_1 prevents the event b from ever becoming enabled. If event a has the same time bounds in both TTMs, then both have the same timed behaviour over the variables of interest y, z. We could therefore delete event b to transform M_1 into M_2 without changing the set of legal computations as projected onto *y* and *z*. Similarly, we can reverse the transformation by adding b to M_2.

The TTM M_2 does not contain any reference to the local variable r occurring in the more complex TTM M_1, and it has a simpler structure. We may think of the simpler TTM M_2 as an *abstraction* of M_1 (similarly M_1 is a *refinement* of M_2). In general, we may

FIGURE 11. Observational congruence of two TTMs



replace a complex system M_1 with an abstraction M_2 provided they are observationally equivalent on the variables of interest.

5.1 Observational congruence of TTMs

The above discussion provides an intuitive understanding of transformations that preserves the behaviour of interest. We refer the reader to [16] for a rigorous definition of what it means for two TTMs to be observationally equivalent in the above sense. A notion of equivalence is needed that distinguishes between deadlocking and non-deadlocking systems, and allows two equivalent TTMs to synchronize on *tick* transitions.

The notion of *observationally congruent* TTMs M_1 and M_2 on a set of variables of interests $\bar{w}$ (written $(M_1 \approx M_2)/\bar{w}$) is developed based on the process algebraic notion of the largest weak bisimulation relation [23]. However, the standard algebraic approach focuses on concurrent programs of uninterpreted actions. The definition of the weak bisimulation therefore had to be extended to deal with the data variables, states, timing and transitions of TTMs. Many of the TTM transformations have no direct analog in process algebras.

A transformation rule such as TA/TD is sound iff it transforms a given TTM into another TTM that is observationally congruent to the first. All the rules of transformation in [16] are sound. The rules are visually intuitive and can be safely used by a software engineer, without the need to understand process algebras and their bisimulations.

There is no complete set of transformations (such as TA/TD) for transforming systems in the general case. However, this does not prevent the theory from being useful in many practical settings. Where the reachability graph of the TTM is finite, the results of [17] provide an efficient algorithm for checking observational congruence of two TTMS.

5.2 Observational congruence of modules

The results of Sect. 5.1 may be applied to modules in a straightforward way. Given two modules $m(\bar{y})$ and $m'(\bar{y})$ (with different bodies but the same interface specification), we can compute if $m(\bar{y})$ is observationally congruent to $m'(\bar{y})$ as follows:

- For each of the modules, compute the associated timed transition models **T**(m) and **T**(m') as discussed in Sect. 4.1 and (Dfn. 2), by arbitrarily interleaving the respective environmental transitions.

- The variables of interest are the interface variables $\bar{y}$. Thus we may define $m(\bar{y}) \approx m'(\bar{y})$ iff $(\mathbf{T}(m) \approx \mathbf{T}(m'))/\bar{y}$. Since it is clear that the variables of interest are the interface variables, we may write more simply $m \approx m'$.

Let module m' be an abstraction of module m, i.e. $m'(\bar{y}) \approx m(\bar{y})$. Since the two modules are observationally equivalent, they have the same set of reduced computations modulo stuttering. The reduced computations can be obtained from the module computations $s_0 s_1 s_2 \ldots$ by replacing each state s_i by its projection onto the module interface variables $\bar{y}$.

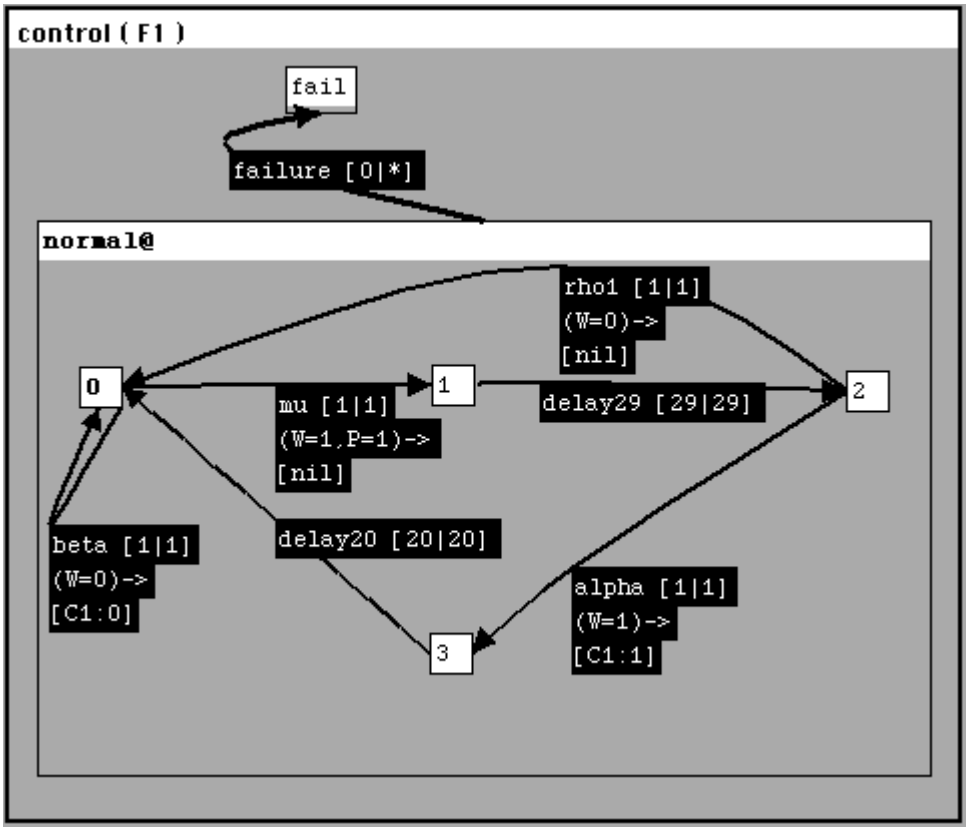
Any quantifier-free temporal logic formula p with no immediate operators is robust with respect to stuttering [22 page 261]. Hence, if the free variables of p are in $\bar{y}$, then we may model-check the abstraction m' for p, with the assurance that p will also hold for m. The module specifications generally satisfy the robustness to stuttering requirement, as they must be valid in the face of arbitrary moves on the part of the environment.

5.3 Abstraction simplifies the DRT controller verification

Applying the equivalence rules (such as TA/TD) to the DRT controller of Fig. 8 to project out the two counters T_a, T_b, produces the more abstract controller of Fig. 12 (see

FIGURE 12. Abstraction of the DRT controller in Fig. 8

The module $control(W, P; F_1, N_1, C_1)$ has initial condition $init(control) \stackrel{\text{def}}{=} (F_1 = normal \wedge N_1 = 0)$, where F_1, N_1 are activity variables with $type(F_1) = \{normal, fail\}$, and $type(N_1) = \{0, 1, 2, 3\}$



[16] for the details). The resulting controller is much simpler than the original, and can serve as a high level specification of the controller. The fact that Fig. 12 is observationally congruent to Fig. 8 is used by [16] as a proof that the pseudocode of Fig. 7 is correct.

However, until the controller is verified in the context of the plant (Fig. 6), we as yet have no guarantee of correctness. For example, if the pressure (P) and power (W) vary arbitrarily, then it is possible for the controller to remain at $(F_1 = normal \wedge N_1 = 2)$ (see Fig. 12) indefinitely ("livelock"). Since, in fact, the sensor variables remain stable for two clock ticks, no livelock occurs, and consequently requirement 3 (Eq. 6) is valid. Furthermore, the 3-version controller is not easily specified by an equivalent TTM. The satisfaction of the temporal logic requirements is therefore the key to system correctness.

The abstract controller helps reduce the combinatorial explosion of states when performing the verification of the requirements. Instead of constructing the reachability graph for $sud = plant \| control \| obs$ (Fig. 4 and Fig. 8) we may instead replace $control$ in sud with its observationally congruent abstraction Fig. 12, taking care to let the observer obs watch for the new initial condition $init(control)$ in Fig. 12. The resulting simpler TTM sud' may now be checked for the three requirements of Sect. 3.2. As can be seen from Table 1, sud' has a significantly smaller reachability graph than sud, and the requirements can be checked in time an order of magnitude smaller than the full system.

TABLE 1. Comparison of model-checking times for the DRT and its abstraction

Type of system under design	Size of graph (states + events)	Time to generate graph and check three requirements on a Sun Sparc 10
Full system sud.	73,403	116 minutes
System with abstract controller sud'.	16,155	16 minutes

Abstraction on its own will not solve the problem of verifying larger systems. For example, when the controlling system is changed to a 3-version system with majority voting, then the computation of the reachability graph is still beyond the power of the verifier, even when the abstract version of each microprocessor is used. To overcome the problem of combinatorial explosion of states, compositional reasoning must be used.

6.0 Compositional Reasoning

In Sect. 4.1 we defined the timed behaviour of a module by introducing the notion of an external environment transition that is allowed to make arbitrary changes to the interface variables. The related notion of "modular-validity" is the corresponding notion for module specifications:

> **Definition 3:** An RTTL formula p is *modularly-valid* for a given module m iff $[m \| m'] \vDash p$ for every module m' that is interface compatible with m. (The notation $m \vDash p$ denotes that p is modularly valid for module m). The conjunction of all modularly-valid properties of module m is denoted by $\mathbf{A}(m)$.

Modular validity ensures that m satisfies p independently of the behaviour of its environment, provided that its environment respects the constraints imposed by the interface specification. Usually, the formula p will refer only to the variables in the interface specification, and not to any of the local variables. An immediate consequence of (Dfn. 3) is the following:

> **(Theorem 1) [Composition Theorem]** Let m_1, m_2 be any two interface compatible modules. Then:

(a) $[(m_1 \models p_1) \wedge (m_2 \models p_2)] \rightarrow [m_1 \| m_2] \models (p_1 \wedge p_2)$, and
(b) $[(\models p_1 \wedge p_2 \rightarrow p) \wedge (m_1 \models p_1) \wedge (m_2 \models p_2)] \rightarrow [m_1 \| m_2] \models p$

Proof: (a) By assumption we have $m_1 \models p_1$ and $m_2 \models p_2$. Since m_1, m_2 are interface compatible, we have by (Dfn. 3) that $m_1 \| m_2 \models p_1$ and $m_1 \| m_2 \models p_2$ both hold. Hence, by propositional temporal logic the formula in (a) must also hold. (b) By assumption and (a) we have that $[m_1 \| m_2] \models (p_1 \wedge p_2)$ holds. Since $(p_1 \wedge p_2) \rightarrow p$, it then follows by propositional temporal logic that $[m_1 \| m_2] \models p$ must hold.

6.1 Modular model-checking

The definition of modular-validity appears to require that we should somehow consider the infinitely many interface compatible modules. However, the notion of the timed transition model $\mathbf{T}(m)$ associated with module m (Dfn. 2) presents a more direct approach, as it includes all possible interleavings with the environment. To check the modular-validity of $m \models p$ it suffices to:

- Use the Build tool to construct $\mathbf{T}(m)$.
- Then, use the Verify tool to check $\mathbf{T}(m) \models p$ (using standard closed system theory).

The behaviour of a module m is completely described by a particular set of computations. These computations can be interchangeably described as a timed transition model $\mathbf{T}(m)$ or by the conjunction of all modularly-valid temporal logic formulas $\mathbf{A}(m)$. In fact, $\mathbf{A}(m)$ is just the set of all RTTL formulas that satisfy $\mathbf{T}(m)$. In principal, there is a procedure to deduce from $\mathbf{T}(m)$, the set of formulas $\mathbf{A}(m)$ [24]. However, $\mathbf{A}(m)$ is usually quite large, and one would never want to expand its definition in practice. Part (c) of the following theorem allows for the conversion of modular-validities into ordinary validities conditional on $\mathbf{A}(m)$:

(Theorem 2)
(a) $m \models \mathbf{A}(m)$, and
(b) $[m_1 \| m_2] \models \mathbf{A}(m_1) \wedge \mathbf{A}(m_2)$ for compatible modules m_1, m_2.
(c) $[m \models p] \equiv [\models \mathbf{A}(m) \rightarrow p]$ for any RTTL property p.

Proof: (a) and (c) follow directly from the definition of $\mathbf{A}(m)$. (b) follows from (a) and the Composition Theorem.

6.2 Structured design of modules

A *top down* method for developing real-time systems may now be followed. To develop a program m satisfying p:

1. decompose m into two compatible modules $[\mathbf{I}(m_1), \mathbf{S}(m_1)]$ and $[\mathbf{I}(m_2), \mathbf{S}(m_2)]$ so that $\mathbf{S}(m_1) \wedge \mathbf{S}(m_2) \rightarrow p$.
2. Then develop bodies $\mathbf{B}(m_1)$ and $\mathbf{B}(m_2)$ that satisfy their specifications $[\mathbf{I}(m_1), \mathbf{S}(m_1)]$ and $[\mathbf{I}(m_2), \mathbf{S}(m_2)]$ respectively.
3. The final modules are then $[\mathbf{I}(m_1), \mathbf{B}(m_1), \mathbf{S}(m_1)]$ and $[\mathbf{I}(m_2), \mathbf{B}(m_2), \mathbf{S}(m_2)]$. The required program is $m = m_1 \| m_2$, which is guaranteed to satisfy p.

A team assigned to the implementation of a module m_i is given its interface $\mathbf{I}(m_i)$ and its temporal logic specification $\mathbf{S}(m_i)$. The job of the team is then to find a suitable body $\mathbf{B}(m_i)$. Many different bodies may satisfy the same specification. One possibility for the

team, is that they may start with a high-level abstract version of the body, and then use the equivalence preserving transformations to refine the body into real code.

6.3 Conditional Specifications

Let m_1 and m_2 be any two compatible modules that are to be composed together (thus m_2 may be thought of as the environment within which m_1 operates and vice versa). It is possible for module m_1 to have a modularly-valid specification of the form $\mathbf{A}(m_2) \rightarrow p$ where p is an RTTL formula. This specification asserts that if the environment of m_1 behaves according to m_2, then m_1 satisfies property p. Such a specification for m_1, when we know that $\mathbf{A}(m_2)$ is guaranteed by its environment, does not contradict our definition that the specification should hold independently of what the environment does. The property $\mathbf{A}(m_2) \rightarrow p$ indeed holds for all environments of m_1, even those that do not maintain the behaviour of m_2 — it is the validity of p that depends on the behaviour of the environment and not the validity of $\mathbf{A}(m_2) \rightarrow p$ which is always guaranteed.

A conditional specification of the form $\mathbf{A}(m_2) \rightarrow p$ for module m_1 is also known as an *assumption/guarantee* property, i.e. if the environment of m_1 can be assumed to behave like $\mathbf{A}(m_2)$, then m_1 is guaranteed to behave according to p.

> **(Theorem 3)** Let m_1 and m_2 be two compatible modules. Then $[m_1 \vDash \mathbf{A}(m_2) \rightarrow p] \equiv [m_1 \| m_2] \vDash p$.

Proof:

$[m_1 \vDash \mathbf{A}(m_2) \rightarrow p]$

$\equiv$ <(Th. 2) part (c) >

$\vDash \mathbf{A}(m_1) \rightarrow [\mathbf{A}(m_2) \rightarrow p]$

$\equiv$ < propositional temporal logic >

$\vDash \mathbf{A}(m_1) \wedge \mathbf{A}(m_2) \rightarrow p$

$\equiv$ < (Th. 2) part (b) and propositional temporal logic>

$[m_1 \| m_2] \vDash p$ Q.E.D.

The above theorem will be useful in the sequel. Consider a compatible system $sud = m_1 \| m_2 \| m_3$. It might not be possible to prove the modular-validity of property p for module m_1, but the weaker conditional property $\mathbf{A}(m_2) \rightarrow p$ may well hold. However, the formula $\mathbf{A}(m_2)$ may be quite long and complex, thus slowing down the model-checking.

An alternative approach would be to verify $[m_1 \| m_2] \vDash p$ using model-checking (Sect. 6.1), and then use (Th. 3) to obtain $m_1 \vDash \mathbf{A}(m_2) \rightarrow p$ as required. We may then conclude that $sud \vDash p$ by the Composition Theorem. Model-checking the composition of two modules $m_1 \| m_2$ is significantly better than having to check the complete system *sud* consisting of three modules, especially if m_3 is complex.

The Composition Theorem is not strong enough to deal with circularities in module specifications in which each module needs the behaviour of the other as an assumption to guarantee its behaviour. One approach is to introduce a stronger assumption/guarantee operator than implication [1], but then the new operator is outside the standard logic connectives. The stronger operator is usually necessary only when two modules have strong dependencies and linkages with each other. In such cases, we have found it easier to retain

the standard implication operator, and to verify the two modules taken together, as explained in the previous paragraph.

6.4 A small example illustrating the Composition Theorem

The module $majorVote(C_1, C_2, C_3; C)$ (Fig. 13) is part of the DRT controller which will be used in the sequel. The controller consists of three independent microprocessors, each one with independent sensors of reactor power and pressure. Each microprocessor controller $control_i$ signals through a variable C_i whether to open the relay (which shuts down the reactor), or to close the relay (allowing the reactor to be started up again). The **in** variables of *majorVote* are thus C_1, C_2, C_3, and the **out** variable is C, which is set to one when the majority of the microprocessor vote for opening the relay (i.e. when $C_1 + C_2 + C_3 \geq 2$). The specification $\mathbf{S}(majorVote)$ can be shown to be modularly-valid by model-checking (Sect. 6.1).

FIGURE 13. Module for majority voting logic

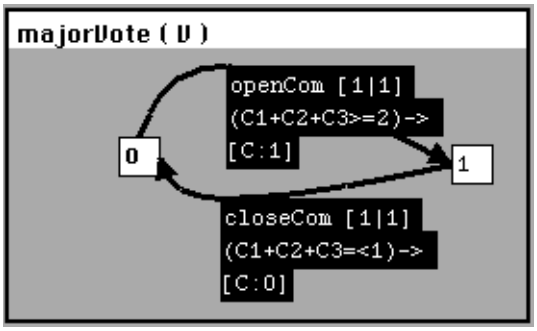
module majorVote
with bitType={0,1} /* 1 stands for a vote to open the relay, and 0 to close the relay. */

INTERFACE

external in C_1, C_2, C_3 : bitType;
out C : bitType where initially $(C = 0)$ /* Only *majorVote* can write to C. */

BODY

local v : bitType where initially $(V = 0)$ /* activity variable */.



SPECIFICATION

$$\mathbf{S}(majorVote): \left[\begin{array}{c} \Box_{<t_1}(C_1 + C_2 + C_3 \geq 2) \Rightarrow \Diamond_{\leq 1} \Box_{<t_1}(C = 1) \\ \wedge \Box_{<t_2}(C_1 + C_2 + C_3 \leq 1) \Rightarrow \Diamond_{\leq 1} \Box_{<t_2}(C = 0) \end{array} \right] \quad \text{(EQ 11)}$$

for any integer time constants $t_1, t_2 \geq 2$.

/*Informal description

The first line of the specification states that once the majority of microprocessor controllers vote to open the relay, and this vote remains in place for time t_1, then within one tick of the clock, the output variable C will be set so as to command the relay to open, and will remain set for t_1 ticks of the clock. The second line states a similar specification for closing the relay. */

end module

If we compose the relay module (Fig. 13) and the voting module (Fig. 13), we may use the modularly-valid module specifications (Eq. 10) and (Eq. 11), and the Composition Theorem to obtain the validity of

$$[majorVote \,\|\, relay] \models p \qquad \text{(Eq. 12)}$$

where p is defined by:

$$p: \left[\begin{array}{l} [\Box_{<20}(C_1 + C_2 + C_3 \geq 2) \Rightarrow \Diamond_{\leq 1}\, \Box_{<20}(R = open)] \\ \wedge\, [\Box_{<2}(C_1 + C_2 + C_3 \leq 1) \Rightarrow \Diamond_{\leq 1}\, (R = closed)] \end{array}\right] \qquad \text{(Eq. 13)}$$

The proof of (Eq. 12) is as follows:

1.	$majorVote \models \Box_{<20}(C_1 + C_2 + C_3 \geq 2) \Rightarrow \Diamond_{\leq 1}\, \Box_{<20}(C = 1)$	by modular-validity of (Eq. 11)
2.	$relay \models \Box_{<20}(C = 1) \Rightarrow \Diamond_0 \Box_{<20}(R = open)$	by modular-validity of (Eq. 10)
3.	$\Diamond_{\leq 1}\, \Box_{<20}(C = 1) \Rightarrow \Diamond_{\leq 1}\, \Diamond_0 \Box_{<20}(R = open)$	(2) and RTTL
4.	$\Diamond_{\leq 1}\, \Box_{<20}(C = 1) \Rightarrow \Diamond_{\leq 1}\, \Box_{<20}(R = open)$	(3) and RTTL
5.	$[majorVote \,\|\, relay] \models \Box_{<20}(C_1 + C_2 + C_3 \geq 2) \Rightarrow \Diamond_{\leq 1}\, \Box_{<20}(R = open)$	RTTL on (1), (4) and the Composition Theorem

The temporal logic reasoning is performed in the RTTL proof system of [24]. For example, the RTTL theorem used in step (3) is: $(p \Rightarrow q) \rightarrow (\Diamond_{\leq 1}\, p \Rightarrow \Diamond_{\leq 1}\, q)$.

The above proof can in fact be performed automatically because it uses the propositional fragment of RTTL [2]. A semi-automated deduction system will soon be available for quantified temporal logic [21].

The Composition Theorem provides a powerful technique for beating combinatorial explosion of states. To verify a global requirement p of a system composed of modules, it is not necessary to deal with the complete system (e.g. by generating its global reachability graph). Instead, we need only verify the specification of each of its components one at a time, provided we can show that the component specifications entail the global requirement.

7.0 Majority voting control of the DRT

The actual controller of the DRT involves the use of three independent microprocessors, with the final decision to shutdown based on majority voting. The single processor controller (Fig. 4) must be replaced by the new 3-version controller shown in Fig. 14 (the plant remains the same as before).

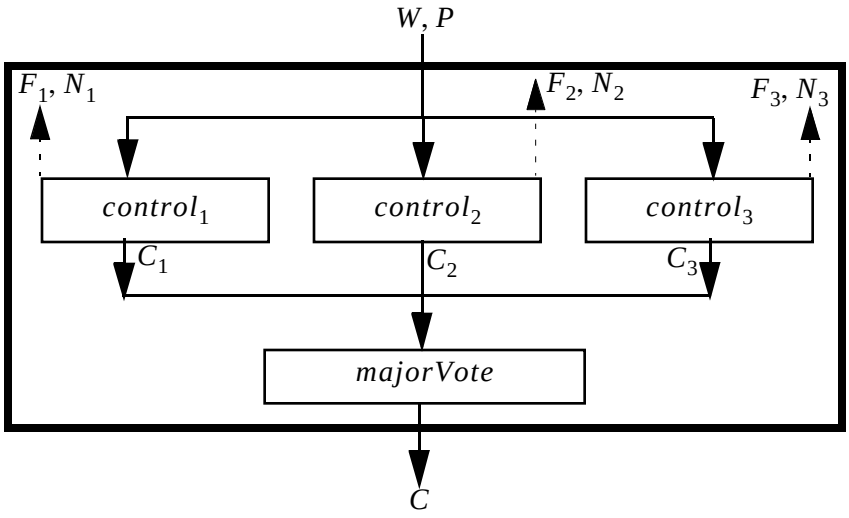
The new controller itself consists of various modules. The module *majorVote* (Fig. 13) was discussed in the previous section. The module $control_1$ (and by analogy $control_2, control_3$) is shown in Fig. 15. The abstract version of the controller is used, thus simplifying the design, because the congruence transformations may be used to refine the abstract controller into pseudocode (Sect. 5.3).

The global requirements (Sect. 3.2) for the new system under design involving the majority voting logic is given by:

$$\textbf{R:}\ \forall i, j{:}\{1, 2, 3\} \left[\begin{array}{l} (i \neq j) \wedge \Box\neg fail_i \wedge \Box\neg fail_j \rightarrow \\ \qquad [cont0(i, j) \wedge bothHi \wedge \Diamond_{30} powerHi \Rightarrow \Diamond_{[30,\,32]} \Box_{<20}(R = open)] \\ \quad \wedge\, [cont0(i, j) \wedge powerLo \Rightarrow \Diamond_{\leq 2}(R = closed)] \\ \quad \wedge\, [\neg cont0(i, j) \Rightarrow \Diamond_{\leq 52} cont0(i, j)] \end{array}\right] \qquad \text{(Eq. 16)}$$

FIGURE 14. Majority voting control module

module control
/* $control = (control_1 \| control_2 \| control_3) \| majorVote$ */
INTERFACE
external in *W, P* /* power and pressure */
out *C* /* command to relay based on majority voting */
out $\bar{F}$ /* where $\bar{F} = F_1, F_2, F_3$ are the failure variables needed for specifying failed behaviour */
out $\bar{N}$ /* where $\bar{N} = N_1, N_2, N_3$ specify normal controller behaviour */
BODY



end module

where i and j range over the three controllers, i.e. $type(i) = type(j) \stackrel{\text{def}}{=} \{1, 2, 3\}$. Control failure is $fail_i \stackrel{\text{def}}{=} (F_i = fail)$ and controller initialization is defined as:

$$cont0(i, j) \stackrel{\text{def}}{=} init(control_i) \wedge init(control_j)$$

where $init(control_i) \stackrel{\text{def}}{=} (F_i = normal \wedge N_i = 0)$. The predicates *bothHi*, *powerHi* and *powerLo* are defined in Fig. 15.

7.1 Modular verification of the system

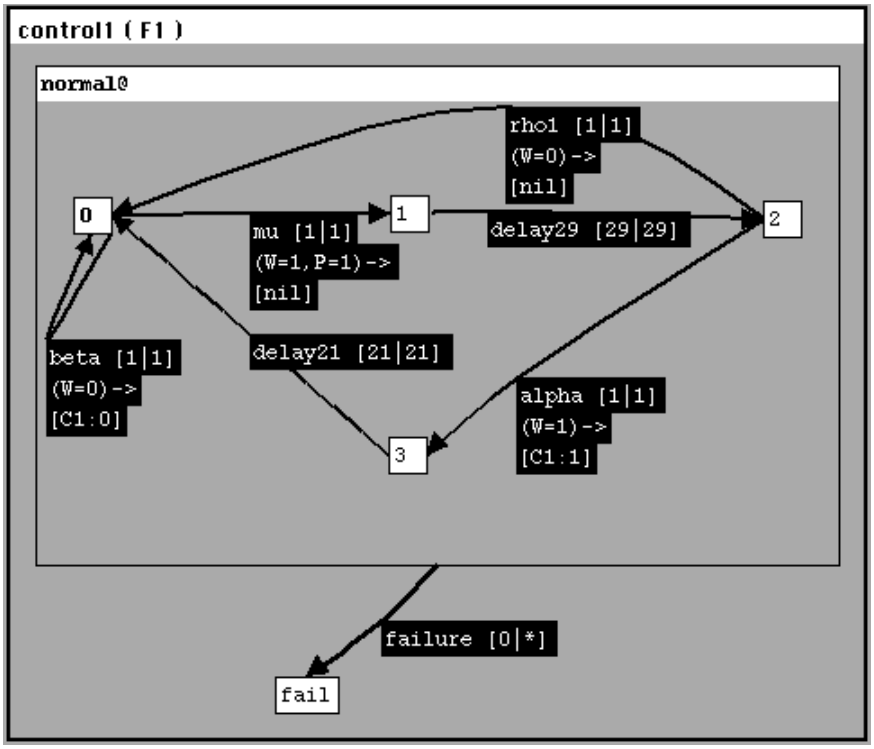
Given that the system under design *sud* is the composition of the plant and the controller with majority voting logic (Fig. 14), we must now show the truth of $sud \models \mathbf{R}$.

The first step is to make sure that the temporal logic module specifications are modularly-valid. (Eq. 10) and (Eq. 11) were already treated in a previous section. What remains is to prove that (Eq. 14) is modularly-valid for $control_1$ (and by symmetrical reasoning the corresponding analogous specifications are modularly-valid for the other two controllers $control_2, control_3$).

(Eq. 14) is defined as a conditional specification $\mathbf{A}(output) \rightarrow p_1$ which states that if the environment of $control_1$ behaves like *output*, then the controller satisfies p_1. By (Th. 3), its is sufficient to check the modular-validity of $[output \| control_1] \models p_1$. In fact $output \| control_1$ is a closed system and has no input environment (although its outputs are the variables F_1, N_1, C_1). The validity of p_1 is easily checked by using an observer similar to that of Sect. 3.2.1.

FIGURE 15. Control module for processor 1

module control1
INTERFACE
external in *W,P* /* power and pressure */
out F_1 where initially $F_1 = fail$ /* the fail variable */
out C_1 /* command to open or close relay */
out N_1 where initially $N_1 = 0$ /* activity variable for normal behaviour */
BODY
/* The body is similar to that of Fig. 12, except that the event *delay20* in the *normal* chart has been replaced by *delay21*. This ensures that plant updates occur at even clock times, and controller responses at odd clock times — which is closer to how the actual system will work. */



SPECIFICATION

$$\mathbf{A}(output(;P, W, D)) \rightarrow p_1, \qquad \text{(Eq. 14)}$$

where p_1 is the specification for processor 1 given by:

$$p_1 : \left[\begin{array}{l} \Box\neg fail_1 \rightarrow [init(control_1) \wedge bothHi \wedge \Diamond_{30} powerHi \Rightarrow \Diamond_{31}\Box_{<20}(C_1 = 1)] \\ \qquad \wedge [init(control_1) \wedge powerLo \Rightarrow \Diamond_{\leq 1}\Box_{<2}(C_1 = 0)] \\ \qquad \wedge [\neg init(control_1) \Rightarrow \Diamond_{\leq 52} init(control_1)] \end{array} \right] \qquad \text{(Eq. 15)}$$

where $init(control_1) \stackrel{\text{def}}{=} (F_1 = normal \wedge N_1 = 0)$, $fail_1 \stackrel{\text{def}}{=} (F_1 = fail)$,
$bothHi \stackrel{\text{def}}{=} (D = 0 \wedge P = 1 \wedge W = 1)$, $powerHi \stackrel{\text{def}}{=} (D = 0 \wedge W = 1)$, and $powerLo \stackrel{\text{def}}{=} (D = 0 \wedge W = 0)$.

/* (Eq. 14) asserts that if the power and pressure variables (the environment of the module) behave as defined by the module *output* (Fig. 6), then p_1 must hold. The *output* module specifies that the values of power and pressure are updated at most every two clock ticks. The specification p_1 requires that if the controller does not fail, then properties similar to the three requirements for a single processor must hold. In an environment in which the power and pressure are allowed to vary arbitrarily, the consequent p_1 will be invalid. */
end module

Having proved that all the module specifications are modularly-valid, we may now directly prove the global requirement $sud \models \mathbf{R}$. Let i, j be arbitrary variables that range over the three controllers. Then

1.	$(i \neq j) \wedge \Box\neg fail_i \wedge \Box\neg fail_j$	**Assume**
2.	$[output \parallel control_i] \models p_i$	Modular validity of (Eq. 14) for *i*-th controller
3.	$[output \parallel control_j] \models p_j$	Modular-validity of (Eq. 14) for *j*-th controller
4.	$[output \parallel control_i \parallel control_j] \models p_i \wedge p_j$	(2), (3) and Composition Theorem
5.	$\mathbf{A}(output) \wedge \mathbf{A}(control_i) \wedge \mathbf{A}(control_j)$	**Assume**
6.	$p_i \wedge p_j$	(Th. 3) and (4), (5)
7.	$p_i \wedge p_j \rightarrow [cont0(i, j) \wedge bothHi \wedge \Diamond_{30} powerHi \Rightarrow \Diamond_{31}\Box_{<20}(C_i = 1 \wedge C_j = 1)]$	(1), RTTL and the definition of $cont0(i, j)$.
8.	$[(i, j{:}\{1, 2, 3\}) \wedge (i \neq j) \wedge (C_i = 1 \wedge C_j = 1)] \Rightarrow (C_1 + C_2 + C_3 \geq 2)$	integer reasoning
9.	$[cont0(i, j) \wedge bothHi \wedge \Diamond_{30} powerHi \Rightarrow \Diamond_{31}\Box_{<20}(C_1 + C_2 + C_3 \geq 2)]$	(1), (6), (7), (8) and RTTL
10.	$[majorVote \parallel relay] \models [\Box_{<20}(C_1 + C_2 + C_3 \geq 2) \Rightarrow \Diamond_{\leq 1} \Box_{<20}(R = open)]$	from (Eq. 12)
11.	$\mathbf{A}(majorVote) \wedge \mathbf{A}(relay)$	**Assume**
12.	$\Box_{<20}(C_1 + C_2 + C_3 \geq 2) \Rightarrow \Diamond_{\leq 1} \Box_{<20}(R = open)$	(Th. 3) and (10), (11)
13.	$cont0(i, j) \wedge bothHi \wedge \Diamond_{30} powerHi \Rightarrow \Diamond_{31}\Diamond_{\leq 1} \Box_{<20}(R = open)$	(9), (12) and RTTL
14.	$[cont0(i, j) \wedge bothHi \wedge \Diamond_{30} powerHi \Rightarrow \Diamond_{[30, 32]}\Box_{<20}(R = open)]$	(13) and RTTL

Line (14) of the above proof produces requirement 1 (first conjunct in the consequent of (Eq. 16)). The other two conjuncts are obtained by similar (and much simpler) reasoning. We thus have:

15. $$\left[\begin{array}{l}(i \neq j) \wedge \Box\neg fail_i \wedge \Box\neg fail_j \rightarrow \\ \qquad [cont0(i, j) \wedge bothHi \wedge \Diamond_{30} powerHi \Rightarrow \Diamond_{[30, 32]}\Box_{<20}(R = open)] \\ \wedge\ [cont0(i, j) \wedge powerLo \Rightarrow \Diamond_{\leq 2}(R = closed)] \\ \wedge\ [\neg cont0(i, j) \Rightarrow \Diamond_{\leq 52} cont0(i, j)]\end{array}\right]$$ **discharging (1)**

16.	$\mathbf{R}$	(15) and *i* and *j* were arbitrary
17.	$\mathbf{A}(sud) \rightarrow \mathbf{R}$	**discharging assumptions (5) and (11)**
18.	$sud \models \mathbf{R}$	(17) and (Th. 3)

8.0 Conclusions

This paper has presented a modular design method that scales up to larger real-time reactive systems than the non-modular approach. In the DRT example, abstraction speeded up automatic verification of individual module specifications, and composition was used to deduce global requirements from the module specifications, without having to produce the global reachability graph. The same techniques can be applied to infinite state systems, except that the StateTime semi-automated deduction tool “Develop” would have to be used to verify the module specifications.

The DRT example was a case of "reverse" engineering. A fully specified design down to pseudocode was already present, and module abstraction was then just a means of reducing the combinatorial explosion of states so that the verifier could automatically check the module specifications. If the methods of this paper are used from scratch, then the abstract controller modules can be developed first; the abstract version of the microprocessor controller (Fig. 15) is close conceptually to the original analog design (Fig. 3), and hence relatively easy to obtain. Then, the equivalence transformations can be used to refine the abstract module down to pseudocode.

If the StateTime tool can verify the complete "closed" system, then such a non-modular approach is preferable; global requirements are usually simple, and the system can be verified automatically. However, more often than not, a modular approach will be needed for realistic systems due to combinatorial explosion of states, and also to keep the system structured and manageable. But, the modular approach requires much more effort on the part of the designer. Module specifications tend to be more complex than global requirements because they must take into account the behaviour of the environment. Also, the composition theorem requires the additional step of applying RTTL compositional reasoning to obtain the global requirements from module specifications.

Much of the RTTL reasoning can be automated. The STeP temporal logic theorem prover and model checker [21] is one possibility (a propositional fragment called PTL was used to check part of the DRT example). Real-time extensions are currently being implemented.

The Build tool was originally designed for closed systems. In future work, it is proposed to automate the facility for real-time reactive modules. Once modules are available, then the refinement checking algorithms of [17], and the equivalence transformations of [16] can also be implemented.

The Verify tool took just under two hours to generate and check a reachability graph of 73,000 states and edges (Table 1). More efficient model-checkers [4,11] can check (untimed) systems of a similar size in a few minutes, but cannot directly verify real-time systems. A current research project has shown that there is a way to use these "smart" model-checkers on real-time systems using a special purpose conversion algorithm. Tentative results indicate that the large system of Table 1 can be checked in a few minutes (an improvement of two orders of magnitude). The intention is therefore to extend the ability of the StateTime tool to export data to these more efficient model-checkers. That would result in a substantial improvement in checking the validity of module specifications.

The addition of the abovementioned features will certainly enhance the effectiveness of the StateTime tool. However, specifying and verifying real systems will still rely on the creativity and hard work of the designer. Although the visual specification language (and/or table specification methods) will allow the software designer to communicate more effectively with the plant engineers, there will still be a need for trained verifiers to deal with the theorem proving aspects of the design.

9.0 References

[1] Abadi, M. and L. Lamport. "Conjoining Specifications." *ACM Trans. on Programming Languages and Systems*, 17(3): 507-534, 1995.

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

[3] Bowen, J. and M.G. Hinchley. “Formal Methods and Safety Critical Standards.” *Computer*, 27(8): 68-71, 1994.

[4] Burch, J.R., E.M. Clark, 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.

[5] Butler, R.W. and G.B. Finelli. “The Infeasibility of Quantifying the Reliability of Life Critical Real-Time Software.” *IEEE Transactions on Software Engineering*, 19(1): 3-12, 1993.

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

[7] Clarke, E.M., E.A. Emerson, and A.P. Sistla. “ Automatic Verification of Finite State Concurrent Systems Using Temporal Logic.” *ACM Transactions on Programming Languages and Systems*, 8:244-263, 1986.

[8] Craigen, D., S. Gerhart, and T. Ralston. *An International Survey of Industrial Applications of Formal Methods*. U.S. National Institute of Standards. NIST GCR 93/626 (Vol. 1 and 2), 1993.

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

[10] 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.

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

[12] Hooman, J. “Correctness of Real-Time Systems by Construction.” In *Proc. Symposium on Formal techniques in Real-Time and Fault-Tolerant Systems*, 19-40. LNCS 863. Springer-Verlag, 1994.

[13] Hooman, J. and W.-P.d. Roever. “Design and Verification in Real-time Distributed Computing: an Introduction to Compositional Methods.” In *Proc. of the 9th International Symposium on Protocol Specification, Testing and Verification*, North-Holland, 1989.

[14] Kurshan, R.P. and L. Lamport. “Verification of a Multiplier: 64 bits and Beyond.” In *Proc. of 5th Computer-Aided Verification Workshop CAV'93*, edited by C. Courcebetis, Springer Verlag, 166-179, 1993.

[15] 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.

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

[17] 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.

[18] 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.

[19] Leveson, N.G. and C.S. Turner. “An Investigation of the Therac-25 Accidents.” *Computer*, 26(7): 18-41, 1993.

[20] Littlewood, B. and L. Strigini. “Validation of Ultrahigh Dependability for Software-Based Systems.” *Communications of the ACM*, 36(11): 69-80, 1993.

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

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

[23] Milner, R. *Communication and Concurrency.* Prentice Hall, 1989.

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

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

[26] 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.

[27] Ostroff, J.S. *StateTime — a Diagrammatic Toolset for the Design and Verification of Real-Time Systems.* Department of Computer Science, York University. TR CS-92-07, 1992.

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

[29] Ostroff, J.S. “Visual Tools for Verifying Real-Time Systems.” In *Theories and Experiences for Real-Time System Development*, 83-101. AMAST Series in Computing: Vol. 2. World Scientific Publishing Company, 1994.

[30] 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.

[31] 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.

[32] Owre, S., J. Rushby, N. Shanka, and F.v. Henke. “Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS.” *IEEE Trans. on Software Engineering*, 21(2): 107-125, 1995.

[33] 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.

[34] Rajan, S., N. Shankar, and M.K. Srivas. “An Integration of Model-Checking with Automated Proof Checking.” In *Workshop on Computer-Aided Verification CAV '95*, Liege, Belgium, Springer Verlag, 84-97, 1995.