Composition and Refinement 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.

Abstract: Reactive systems exhibit ongoing, possibly non-terminating, interaction with the environment. Real-time systems are reactive systems that must satisfy quantitative timing constraints. This paper presents a structured compositional design method for discrete real-time systems that can be used to combat the combinatorial explosion of states in the verification of large systems. A *composition rule* describes how the correctness of the system can be determined from the correctness of its modules, without knowledge of their internal structure. The advantage of compositional verification is clear. Each module is both simpler and smaller than the system itself. Composition requires the use of both model-checking and deductive techniques. A *refinement rule* guarantees that specifications of high-level modules are preserved by their implementations. The *StateTime* toolset is used to automate parts of compositional designs using a combination of model-checking and simulation. The design method is illustrated using a reactor shutdown system that cannot be verified using the StateTime toolset (due to the combinatorial explosion of states) without compositional reasoning. The reactor example also illustrates the use of the refinement rule.

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).

Table of Contents

Figure

1.0 Introduction

Reactive systems exhibit ongoing, possibly non-terminating, interaction with the environment. *Real-time* systems are reactive systems that must satisfy quantitative timing constraints. This paper presents a structured compositional design method for discrete real-time systems that can be used to combat the combinatorial explosion of states in the verification of large systems.

A system is decomposed into parallel components called *modules*. A *composition rule* describes how the correctness of the system can be determined from the correctness of its modules, without knowledge of their interior structure. The advantage of compositional verification is clear. Each module is both simpler and smaller than the system itself.

In addition to system decomposition, an abstract specification of a module may need to be refined into implementations closer to code. A *refinement rule* guarantees that specifications of abstract modules are preserved by their implementations.

The *StateTime* toolset is used to automate parts of compositional designs using a combination of model-checking and simulation. The design method is illustrated using a reactor shutdown system that involves the use of three microprocessors, each independently checking sensor readings, with the final decision to shut down based on a majority vote. The single microprocessor version can be checked in the StateTime toolset without compositional reasoning. However, the three-microprocessor system suffers from a combinatorial explosion of states and a compositional approach is thus needed. The reactor example also illustrates the use of the refinement rule.

The compositional design method is based on the TTM/RTTL framework [36,37,40] which consists of the following:

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

- A *declarative specification language* called real-time temporal logic (RTTL) for describing the requirements that a TTM should satisfy without discussing how the TTM is constructed. RTTL is a timed extension of linear temporal logic augmented with a transition variable for describing TTM events.

- *Analysis techniques* for demonstrating that a TTM conforms to its specification. Model-checking and a proof system for theorem proving are the main analysis techniques. Model-checking is a method for automatically verifying concurrent systems in which a finite-state model of the system (TTM) is compared with a correctness requirement (RTTL). Since time is a monotonically increasing variable, the state-space of naive timed systems is automatically infinite state. Hence, special care is taken in the model-checking algorithms to keep the state space finite provided the data types are finite.

- A toolset called *StateTime* [38] which has a *visual* statechart-like *executable* language for representing TTMs hierarchically. An automatic translator to the model-checker and theorem prover STeP [31] allows for analysis. Although STeP is designed for untimed systems, the automatic translation is done in such a way so as to allow for the use of STeP's model-checking facilities. The STeP theorem prover can also be used for simplifying properties.

The TTM/RTTL framework was initially conceived for the analysis of *closed* systems whose behaviour is completely determined by the state of the system itself [17]. In a closed system, we assume that the environment may set the initial values of input variables, but, once the system starts running the environment cannot modify any of the system variables. Thus, all changes to the system variables are accounted for by the transitions of the program. By contrast, reactive systems are best thought of as *open* systems whose behaviour depends on continuous interaction with the environment. We provide below an informal sketch of how the framework is extended to the open setting. The concepts will be made precise in the sequel.

This paper defines the notion of an open real-time reactive *module* $m = [i, b, s]$ where i is the module *interface stub* (e.g. variables or channels shared with the environment), b its *body* (a TTM) and s the module *specification* (an RTTL formula in the interface variables). The module specification s must hold for all module computations including arbitrary changes that the environment might make at any time to the interface variables. The composition of two modules $m_1 \| m_2$ is also a module.

Not all parts of a module are always determined. For example, the interface stub and specification may be given, but not the body. We denote a module with an unspecified body by $[i, \bullet, s]$. A Composition Rule (justified in the sequel) given by

Composition Rule: $$\left.\begin{array}{c} m_1 \vDash s_1 \\ m_2 \vDash s_2 \\ (s_1 \wedge s_2) \rightarrow r \end{array}\right\} m_1 \| m_2 \vDash r$$

states that if each of the modules satisfy their respective specifications, then the system satisfies its global requirement r provided the requirement can be derived from the conjunction of the module specifications. The composition rule allows for both bottom-up and top-down design. In the bottom-up method, the independently designed and implemented modules (with respective specifications s_1, s_2) when brought together exhibit the emergent property r provided $(s_1 \wedge s_2) \rightarrow r$.

In top-down development, the system under design (*sud*) that is required to conform to a global system requirement r can be decomposed into modules $m_1 = [i_1, \bullet, s_1]$ and $m_2 = [i_2, \bullet, s_2]$ provided $(s_1 \wedge s_2) \rightarrow r$. At this stage, we have not yet committed to module implementations. Each of these modules can then be given to a programmer whose job it is to develop a body that satisfies the module specification.

The body of module m_1, whose variables can be reduced to finite ranges, can be shown to satisfy its module specification (i.e. $m_1 \vDash s_1$) by model-checking provided the effects of the environment are taken into account. The proof of $(s_1 \wedge s_2) \rightarrow r$, except in the simplest of cases, requires the use of deductive techniques (RTTL theorem proving). Thus the composition rule usually involves a combination of algorithmic and deductive techniques.

It is advisable that the programmer design and code the body of a module at as high a level as possible (using TTMs). This keeps the body simple and small which makes it understandable and prevents state explosion. There is then a need to *refine* the high-level module body into a TTM that is closer to implementation. For example, an abstract TTM may directly specify a delay of 50 ticks, but the implementation on a microprocessor might be a loop construct that increments a counter every traversal of the loop. The internal loop and counter are unobservable to an external agent interacting with the module as the agent can only observe changes in the interface variables.

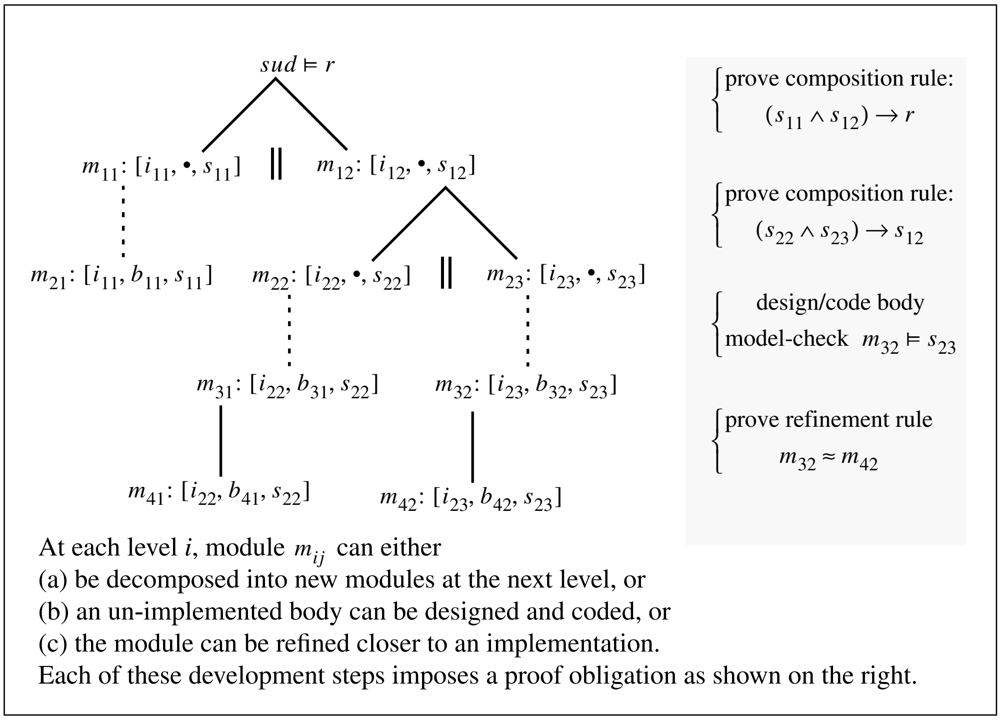
Two modules with the same interface are observationally equivalent (written: $m_1 \approx m_2$) if they agree on timed observations of their interface variables. Under suitable conditions (presented in the sequel) a Refinement Rule states that:

$$\textbf{Refinement Rule}: \quad \left.\begin{array}{c} m_1\colon [i, b_1, \bullet] \\ m_2\colon [i, b_2, \bullet] \\ m_1 \approx m_2 \end{array}\right\} (m_1 \vDash s) \equiv (m_2 \vDash s) \text{ for any module specification } s.$$

Hence, if m_2 is observationally equivalent to m_1, then m_2 can replace m_1 wherever it occurs with a guarantee that any module specification s will be preserved. There are efficient polynomial algorithms for checking observational equivalence of finite state systems, and equivalence preserving transformations are available for refining infinite state systems.

Given a requirement r that a system *sud* must satisfy, the composition and refinement rules allow for a systematic modular development method represented by the tree in Fig. 1. Each step imposes a proof obligation as shown in the right hand column of the figure. The process continues until all the modules have bodies that can be directly coded into the given program language. We need not adhere to the ordering suggested by the figure. For example, the complete implementation of m_{11} can take place before the other modules are designed. It is also possible to reverse-engineer already implemented code and move bottom-up.

FIGURE 1. Structure diagram for compositional design method


$sud \vDash r$
m_{11}: $[i_{11}, \bullet, s_{11}]$
$\|$
m_{12}: $[i_{12}, \bullet, s_{12}]$
m_{21}: $[i_{11}, b_{11}, s_{11}]$
m_{22}: $[i_{22}, \bullet, s_{22}]$
$\|$
m_{23}: $[i_{23}, \bullet, s_{23}]$
m_{31}: $[i_{22}, b_{31}, s_{22}]$
m_{32}: $[i_{23}, b_{32}, s_{23}]$
m_{41}: $[i_{22}, b_{41}, s_{22}]$
m_{42}: $[i_{23}, b_{42}, s_{23}]$
prove composition rule: $(s_{11} \wedge s_{12}) \rightarrow r$
prove composition rule: $(s_{22} \wedge s_{23}) \rightarrow s_{12}$
design/code body
model-check $m_{32} \vDash s_{23}$
prove refinement rule
$m_{32} \approx m_{42}$

At each level i, module m_{ij} can either
(a) be decomposed into new modules at the next level, or
(b) an un-implemented body can be designed and coded, or
(c) the module can be refined closer to an implementation.
Each of these development steps imposes a proof obligation as shown on the right.

We proceed as follows in the rest of this paper. In section 2 we provide background information needed to understand the TTM/RTTL framework and the StateTime toolset. Section 3 defines the notion of a module, modular validity and the composition rule. It also describes how conditional specifications can be used to constrain module environments. Section 4 presents the refinement rule for modules based on the notion of observational equivalence of TTMs developed in [26]. Observational equivalence of TTMs will be defined precisely in the sequel, but the reader is referred to [27] for a set of TTM equivalence preserving transformations and to [28] for an efficient polynomial time algorithm to check TTM observational equivalence. Module observational equivalence is defined in such a way that the TTM results can be applied directly to module equivalence as well. In Section 5, we use the composition and refinement rules for the structured design of a reactor shutdown system. The design method is also discussed in some detail (Sect. 5.6). Comparisons to other approaches and concluding remarks are presented in Section 6.

2.0 Background

In the sequel, we use relative quantification $(Qx{:}T|R{:}P)$ where Q is a quantifier ($\forall$ or $\exists$), T is the type of the dummy variable x, R is the range of the dummy variable and P a predicate [14]. For example, $(\forall i{:}int|3 \le i{:}P)$ means “for all values of an integer variable i, if i is at least as large as 3 then i has property P”. If no range is supplied then it is *true*. The notation $d{:}D$ generally means that $d \overset{\text{def}}{=} D$. For example, $f{:}(\exists d{:}int|d > 0{:}x + d \le 4)$ means that we are defining f by $f \overset{\text{def}}{=} (\exists d{:}int|d > 0{:}x + d \le 4)$. In TTM update functions (see sequel), $d{:}D$ denotes assignment, i.e. $d{:=}D$.

2.1 Real Time Temporal Logic (RTTL)

Linear time temporal logic [32] uses temporal connectives such as $\Box$ (henceforth), $\bigcirc$ (next), $\Diamond$ (eventually), $\mathcal{U}$ (until) and past operators such as $\ominus$ (previous state) to represent qualitative temporal properties. The standard connectives are applied to *state-formulas* (which are the atomic predicates) to obtain temporal logic formulas.

Real-time temporal logic (RTTL) is obtained by adding a fair *tick* transition and the ability to refer to system transitions via a distinguished transition variable. We refer the reader to [32] for a precise discussion of standard temporal logic and to [37,40] for real-time temporal logic. We now provide a brief review of some of the basic concepts.

Let x and y be the system variables where the type of x is the integers and y has a set type. An example of a state-formula f is $(\exists d{:}int|(x + d \le 4) \wedge (7 \in y))$. In this formula, the bound variable d is just a dummy variable and is not considered a system variable. A state is a mapping from the system variables to values in their relevant types. Since f evaluates to true in the state given by $s = \langle x{:}2, y{:}\{7, 9\}\rangle$, we write $s \vDash f$ (state s *satisfies* f), and we call s an f-state.

A temporal logic formula such as $\Diamond f$ (“eventually f is true”) cannot be interpreted in a single state; rather it is evaluated in an infinite sequence of states σ given by $\sigma = s_0 s_1 s_2 \ldots s_i \ldots$ where $\sigma \vDash \Diamond f$ (“σ satisfies $\Diamond f$“) will mean that there is at least one state subsequent to the initial state that is an f-state. An inductive definition of the satisfaction relation $\vDash$ can then be given. Let $(\sigma, i) \vDash f$ denote the satisfaction of temporal formula f at a position $i \ge 0$ of the sequence σ. For a state-formula f, $[(\sigma, i) \vDash f] \overset{\text{def}}{=} [s_i \vDash f]$.

We can then give the appropriate inductive definitions for the propositional connectives (e.g. negation, conjunction, implication) followed by the usual definition of the temporal operators. For example, for temporal logic formulas g and h, the *until* operator is defined by $[(\sigma, i) \vDash g\mathcal{U}h] \overset{\text{def}}{=} (\exists j | j \geq i{:}(\sigma, j) \vDash h \wedge (\forall k | i \leq k < j{:}(\sigma, k) \vDash g))$. For an arbitrary temporal logic formula f, $\sigma \vDash f$ is an abbreviation for $(\sigma, 0) \vDash f$. A formula f is *generally-valid* iff $(\forall \sigma | \sigma \vDash f)$.

The implication $(f \rightarrow \Diamond g)$ states only that "f implies eventually g" at the initial position of the computation, i.e. if f holds at the initial position then there is a subsequent position where g holds. As a notational convenience, we will write $f \Rightarrow \Diamond g$ for $\Box(f \rightarrow \Diamond g)$ which states that the implication holds at *all* positions of the sequence. In general, the double arrow *entails* operator is defined by $[p \Rightarrow q] \overset{\text{def}}{=} \Box[p \rightarrow q]$ for any temporal logic formulas p and q.

We need the notion of *timed transition sequences* for the description of real-time systems. Since we envisage that a *transition* τ_i causes a transfer from state s_{i-1} to state s_i, we may rewrite the infinite sequence of states $\sigma = s_0 s_1 s_2 \ldots$ as:

$$\sigma = (\tau_0, s_0)(\tau_1, s_1)(\tau_2, s_2)\ldots \qquad \text{(Eq. 1)}$$

The start transition τ_0 (e.g. a computer reboot) puts the system in state s_0. The transition τ_1 takes the system from state s_0 to s_1 and so on. We give the initial transition τ_0 the special name *start*. The distinguished variable ε (the *transition variable*) is always part of the state. The transition variable is used to record the last event taken, i.e. for the sequence $(start, s_0)(\tau_1, s_1)(\tau_2, s_2)\ldots$ we have that $s_0(\varepsilon) = start$ and $(\forall i | i > 0{:}s_i(\varepsilon) = \tau_i)$. The reason we need a *start* transition is so that ε, like all other state variables, has an initial value.

The transition variable can be used to refer directly to event occurrences. For example, for a traffic system, the temporal logic formula $(\varepsilon = turn\_red) \Rightarrow \Diamond(\varepsilon = turn\_green)$ asserts that anytime the light turns red, it must eventually turn green.

In order to represent time, we introduce the special transition *tick*. A timed sequence σ must satisfy the *ticking constraint* which asserts that there are an infinite number of ticks occurring in the sequence, i.e. $\sigma \vDash \Box\Diamond(\varepsilon = tick)$. Thus, time must progress irrespective of what happens in a system or its environment. It is possible for any finite number of transitions to occur between two ticks of the clock.

We may use quantified Manna-Pnueli temporal logic to define the bounded real-time until operator, $p\mathcal{U}_{[l, u]}q$, which in turn can be used to express a variety of important real-time properties. Informally[2], the meaning of the bounded until operator is that eventually q will occur at a time between l and u ticks from now; until then q must hold. Other bounded operators can then be defined as follows:

$\Diamond_{[l, u]}\, p$	$\overset{\text{def}}{=}$	$(true\mathcal{U}_{[l, u]}\, p)$	*Bounded response*: p must hold after the l-th tick but before the $(u + 1)$-th tick.
$\Diamond_{\leq u} p$	$\overset{\text{def}}{=}$	$\Diamond_{[0, u]}\, p$	p must hold before the $(u + 1)$-th tick.

[2] Formally, the *bounded until* operator is defined using a *flexible* clock variable t (that is incremented by one every time the clock ticks), and a *rigid* time variable t_0 (that retains the same value over all states) as follows: $p\mathcal{U}_{[l, u]}q \overset{\text{def}}{=} (\forall t_0{:}type(t) | (t = t_0) \rightarrow p\mathcal{U}(q \wedge (t_0 + l \leq t \leq t_0 + u)))$. Please refer to [36,40] for the precise details. Since the bounded time operators are defined using ordinary quantified temporal logic, the untimed temporal theorem prover STeP [31] can be used to show the validity of theorems such as $\Diamond_0\Diamond_2 p \equiv \Diamond_2 p$, which can, in principal, be used for the deductive reasoning in the sequel.

$\Box_{<l}p \quad \overset{\text{def}}{=} \quad (p\,\mathcal{U}_{[l,\,\infty]}\,true)$ *Bounded invariance*: p must hold until the l-th tick.

$\Diamond_{d}p \quad \overset{\text{def}}{=} \quad (true\,\mathcal{U}_{[d,\,d]}\,p)$ *Exact time*: p is true in exactly d ticks.

The formula $\Diamond_0 p$ asserts that p will hold before the next tick of the clock. Several state changes can occur before p occurs without the clock advancing. The $\Diamond_0$ operator can often be used in place of the *next* operator where there is a need for stuttering-invariant formulas, i.e. formulas that are "robust" with respect to unobservable moves of the environment. Some further examples of clocked properties are:

- $p \to (q\,\mathcal{U}_{[3,\,7]}\,r)$: If p holds initially, then eventually between 3 and 7 ticks r holds, and q must hold continuously until then. This property is asserted only at the initial position.
- $\Box(p \to (q\,\mathcal{U}_{\leq 4}\,r))$: Every position satisfying p is followed within 4 ticks by r, and q holds continuously until then.
- $p \Rightarrow \Diamond_0 \Box_{<2} q$: If p holds at a position, then at some subsequent position before the next clock tick there should be the start of an interval of duration 2 ticks during which q holds continuously.
- $\Box(p \to \Box_{<3} \neg q)$: The property q cannot become true sooner than 3 ticks after any occurrence of the property p.

We often need to compare expressions in consecutive states. We therefore introduce an abbreviation for the next value of a variable v, written v'. For example, the formula $\Box(v' > v)$ asserts that the value of v is greater in every successor state that it is in its immediate predecessor (see [32] for the precise details).

2.2 Timed Transition Models (TTMs)

TTMs are timed extensions of the fair transition systems of Manna and Pnueli [32]. The extension involves lower and upper time bound constraints on transitions, that refer to the number of occurrences of the special transition *tick*. A TTM M is defined as a 4-tuple $M = (V, I, T, F)$ as follows:

- V: a finite set of typed *system variables*. The distinguished transition variable ε is always in V, where $type(\varepsilon) = T$. The variables set also include control and data variables that are used to describe the various parts of M. Each state of M is a map from V to its types; the set of all states is denoted by Σ_M (or just Σ when it is clear what the TTM is).
- I: the *initial condition*. This is a satisfiable boolean valued expression in the system variables that characterizes the states at which the execution of the TTM can begin. A state s satisfying I is called an *initial state*.
- T: a finite *set of transitions* which includes the distinguished transitions *start* and *tick*. Each transition $\tau \in T$ is a function $\tau{:}\Sigma \to powerset(\Sigma)$ that maps a *prestate* s in Σ to a (possibly empty) set of τ-*successor* states $\tau(s) \subseteq \Sigma$. An empty successor set means that the transition is disabled (i.e. cannot be taken from the prestate). A successor state s' is also called a *poststate* of τ from s. If the set of successor states consists of a single poststate, then the transition is deterministic. If there is more than one poststate, then the transition is nondeterministic.

- F: a *fairness set* where $F \subset T$. Informally, the fairness constraint for each transition $\tau \in F$ disallows computations in which τ is enabled infinitely often but is taken only finitely many times[3].

Since, in general, we do not need nondeterministic transitions[4], we can also describe a transition τ by its *enabling condition* $enb(\tau)$ (the condition under which the transition becomes eligible to be taken), and a simultaneous *update function*

$$upd(\tau) = \{v_1 \colon e_1, v_2 \colon e_2\}$$

where e_1 and e_2 are expressions in the system variables, which indicates that the values of v_1, v_2 in the poststate s' are $s(e_1), s(e_2)$ respectively, where s is the prestate. No other system variables (e.g. v_3) are changed. The transition τ is *enabled* in a state s (written: $s \vDash enb(\tau)$) if $\tau(s) \neq \varnothing$ — otherwise τ is said to be *disabled*.

The transition τ can be fully characterized by a *transition relation* ρ_τ given by $\rho_\tau \colon enb(\tau) \wedge (v_1' = e_1) \wedge (v_2' = e_2) \wedge (v_3' = v_3)$ which is a predicate in the primed and unprimed system variables. Primed variables refer to the value of the variables in the poststate, and unprimed variables refer to values in the prestate (see [32] for precise details). By convention, we leave out conjuncts such as $(v_3' = v_3)$ for which there is no change.

In addition to the enabling condition and update function, we associate with each non-tick transition τ a lower time bound $low(\tau)$ and an upper time bound $hi(\tau)$, where $0 \le low(\tau) \le hi(\tau) \le \infty$. We allow bounds $\tau[0, 0]$ and $\tau[0, \infty]$ but not $\tau[\infty, \infty]$. The meaning of these bounds will be defined formally in the sequel, but we first provide an informal overview.

A timed transition $\tau[l, u]$ with lower time bound l ticks and upper time bound u ticks, must delay l ticks before being taken, but must be taken by u ticks of the clock, provided it remains continuously enabled, and is not disabled by the occurrence of another transition that might have the effect of disabling τ.

The operational semantics of TTMs will be described by the set of all its behaviours called *trajectories*. Informally, a trajectory is a timed sequence of states that starts in an initial state satisfying the initial condition of the TTM. From any state of the computation, any enabled transition is taken *in one atomic step*. Either a *tick* transition is taken at each step, in which case time advances, or a non-tick transition is taken, in which case time stays the same. The resulting interleaving of enabled transitions allows us to model concurrent processes[5]. When the transitions are taken, they update the variables according to the transition update function. The clock must tick infinitely often in any computation, and

[3] Fairness is defined more formally in the sequel. A weaker notion of fairness than the one defined in this paper is called *justice* [32]. Fairness ensures that in interleaved parallel processes, the processes progress independently (fairness distinguishes concurrency from nondeterminism). The stronger notion of fairness defined in this paper is needed for the *tick* transition.

[4] There is one exception to the rule. When $choose(v_1, v_2)$ is used in transition updates it assigns arbitrary values to v_1 and v_2 in their appropriate types. No assumptions are made about the probabilistic distributions of the values assigned. This is a purely nondeterministic update that says any value in the type is possible in the successor state. This notion will used to construct environments of modules (Sect. 3.0).

[5] Actual systems may have *overlapped* rather than interleaved execution. However, provided an appropriate fair set of transitions with the right level of atomicity is chosen, the interleaving model can accurately describe overlapped execution (see [32, p103] for further discussion).

an arbitrary but finite number of (non-tick) transitions can be taken between any two ticks of the clock. The lower and upper time bounds of transitions must be respected.

A *computation* $\langle \tau_0, s_0\rangle \langle \tau_1, s_1\rangle \langle \tau_2, s_2\rangle \ldots$ of a TTM $M = (V, I, T, F)$, where $\tau_i \in T$ for $i \geq 1$ and $\tau_0 \overset{\text{def}}{=} start$, is a timed sequence satisfying the three constraints below. In each case, we show how to write the constraint as a temporal logic formula.

1. *Initialization constraint*: The first state of the computation satisfies the initial condition, i.e. $s_0 \vDash (I \wedge (\varepsilon = start))$. The initialization constraint is thus represented by the temporal logic formula $init(M) \overset{\text{def}}{=} I \wedge (\varepsilon = start) \wedge \bigcirc\Box(\varepsilon \neq start)$. The transition *start* occurs once at the beginning of the computation and never again.
2. *Succession constraint*: $(\forall i | i \geq 0 \colon s_{i+1} \in \tau_{i+1}(s_i))$, i.e. every prestate at position *i* must have as its successor a poststate according to the update function of τ_{i+1} (the transition taken at position *i*). The succession constraint can be expressed in RTTL as $succession(M) \overset{\text{def}}{=} \Box(\exists\tau{:}T | \rho_\tau)$, where ρ_τ is the transition relation for τ.
3. *Fairness constraint*[6]: For each transition τ in the fairness set, it is not the case that τ is infinitely often enabled beyond some position in the trajectory, but taken at only finitely many positions in the trajectory.The fairness constraint can be written in temporal logic as $fair(M) \overset{\text{def}}{=} (\forall\tau{:}F | \Box\Diamond enb(\tau) \rightarrow \Box\Diamond(\varepsilon = \tau))$.

A *timed sequence* that satisfies the above three constraints is called a *computation* of M. A computation describes the behaviour of a Manna-Pnueli fair transition system (enhanced with the *tick* of timed sequences). To describe the behaviour of timed transition models, we further constrain computations by lower and upper time bound constraints and call the resulting computations *trajectories*.

4. *Lower bound constraint:* for every transition τ with lower bound $l > 0$, if τ is taken at position *j* of the computation, then there must exist a prior position $i < j$ so that there are at least l ticks of the clock between i and j, and $(\forall k | i \leq k < j \colon s_k \vDash enb(\tau) \wedge \varepsilon \neq \tau)$, i.e. τ is enabled but not taken in the states $s_i \ldots s_j$.
5. *Upper bound constraint*: for every transition τ with upper bound $u \neq \infty$, if τ is enabled at position *j* of the computation, then there must exist a subsequent position $k \geq j$ with no more than u ticks of the clock between j and k, such that either τ is taken or disabled at position *k*.

As with the initialization, succession, and fairness constraints, both the bound constraints can also be described in RTTL. For a non-tick transition τ with lower time bound l (where $l > 0$) and upper time bound u, the bound constraint is:

$$bound(\tau) \overset{\text{def}}{=} moe(\tau) \Rightarrow [\Box_{<l}\bigcirc(\varepsilon \neq \tau)] \wedge [enb(\tau)\,\mathcal{U}_{\leq u}(\varepsilon = \tau \vee \neg\, enb(\tau))] \qquad \text{(Eq. 2)}$$

[6] The fairness constraint is included for generality but is not necessary for the example developed in the sequel. However, real-time systems may have requirements where fairness is useful. For example, there may be a requirement to log every error to a file or printer; this does not have to happen within a precise time as the requirement may merely be that the error is eventually logged. In the Fischer protocol, mutual exclusion is ensured if certain precise timing constraints are satisfied; however, for response it may be sufficient to specify that each process eventually returns to it’s non-critical region. For TTMs, we allow in increasing stringency: spontaneous $\tau[0, \infty]$ transitions, fair transitions and timed transitions. This allows us to describe systems to the appropriate precision.

where $moe(\tau) \overset{\text{def}}{=} enb(\tau) \wedge [\varepsilon \in \{start, \tau\} \vee \ominus\neg enb(\tau)]$, and where $\ominus p$ (the *previous* temporal operator) holds at a position j of a trajectory provided j is not the first position of the trajectory and p holds at position $j-1$. If $l = 0$, then the left conjunct $\Box_{<l}[\bigcirc(\varepsilon \neq \tau)]$ is replaced by *true*. If $u = \infty$, then the right conjunct of the consequent in (Eq. 2) is replaced by *true*. The bound constraint (for both lower and upper bounds) can be written in temporal logic as:

$$bound(M) \overset{\text{def}}{=} (\forall \tau{:}T | bound(\tau)). \qquad \text{(Eq. 3)}$$

The moment of enablement $moe(\tau)$ describes the relevant positions of a computation at which the bound constraint for a transition τ (that is enabled at that position) must be asserted. A relevant position is either the initial position $(\varepsilon = start)$, or a position at which the transition has just been taken $(\varepsilon = \tau)$ and is re-enabled, or a position where τ has just become enabled $(enb(\tau) \wedge \ominus\neg enb(\tau))$.

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

As noted above, the initialization, succession, fairness and bound constraints can be expressed in RTTL. The formula $des(M)$ defined by

$$des(M) \overset{\text{def}}{=} init(M) \wedge succession(M) \wedge fair(M) \wedge bound(M) \qquad \text{(Eq. 4)}$$

fully describes the set of all trajectories of the TTM *M*.

Since a trajectory of a TTM M is a timed sequence, the trajectory must also satisfy the ticking constraint $ticking(M)$: $\Box\Diamond(\varepsilon = tick)$. However, there is the possibility of a conflict between the upper bound and the ticking constraint (in which case no timed sequence will satisfy $des(M)$ and the ticking constraint simultaneously). This happens in the presence of *immediate* transitions of the type $\tau[0, 0]$ that are self-loops — such a τ is taken repeatedly yet the tick transition is delayed indefinitely[7]. This is called a Zeno computation and the TTM is said to exhibit *Zeno behaviour.* Any cycle of transitions whose elements are all immediate may also exhibit Zeno behaviour. A TTM that exhibits Zeno behaviour cannot be implemented, and hence we must find ways to ensure that our systems are non-Zeno.

The problem of Zeno computations can be avoided by disallowing self-looping immediate transitions. However, immediate transitions are useful for modelling “instantaneous” (i.e. before the clock ticks) reactions. If immediate transitions are used in a TTM *M*, then we must check for the validity of $\Box\Diamond(\varepsilon = tick)$ in every single computation that satisfies the bound constraints. Fortunately, for those systems where model-checking can be used, the ticking property can be verified automatically (e.g. see Table 1 in Sect. 5.5). *In the*

[7] The StateTime tool automatically converts TTMs to fair transition systems [39] that can then be analyzed using STeP (see Sect. 2.4). In this conversion, additional conjuncts are added to the enabling condition of the *tick* transition that disables the tick transition when an urgent timed transition must be taken. In a system with a selfloop $\tau[0, 0]$ transition, the *tick* transition is disabled indefinitely. This reflects the conflict between the ticking constraint and the upper time bound constraint. The conversion procedure does declare tick to be fair. However, since *tick* is disabled until the urgent transition is taken, the fairness constraint is satisfied despite the fact that *tick* is not taken.

sequel, we assume that all TTMs are non-Zeno. This is not restrictive at all for the examples of this paper because all TTMs can be model-checked to ensure that they are non-Zeno.

The set of all trajectories of a TTM M is denoted by $traj(M)$. If a trajectory σ satisfies a temporal logic formula p, then we write $\sigma \vDash p$. If an RTTL formula p is satisfied in all trajectories of M (i.e. $(\forall\sigma{:}traj(M)|\sigma \vDash p)$), then we write $M \vDash p$, and the formula p is said to be *M-valid*. Any generally-valid formula is also *M*-valid. Any trajectory in $traj(M)$ always satisfies $des(M)$; hence, the transition system M and the temporal logic formula $des(M)$ are two equivalent ways of describing $traj(M)$.

Theorem 1: For any (non-Zeno) TTM M and RTTL formula p in the variables of M: (a) $[M \vDash p] \equiv [\,\vDash des(M) \to p]$, and (b) $M \vDash des(M)$.

If we treat $des(M)$ as an axiom of the RTTL logic, then (Th. 1)(a) describes the relative completeness of the logic for proving *M*-validities. An oracle is a device that is guaranteed to provide a proof of any generally-valid RTTL formula. Hence to prove the *M*-validity of p it is sufficient to submit to the oracle the formula $des(M) \to p$. While the axiom $des(M)$ is theoretically adequate it is not very practical. In practice the special proof rules in [36] and model-checking (Sect. 2.4) are the preferred methods for proving *M*-validities.

2.3 Parallel composition of TTMs

The parallel composition $M_1 \| M_2 = (V, I, T, F)$ of two TTMs $M_1 = (V_1, I_1, T_1, F_1)$ and $M_2 = (V_2, I_2, T_2, F_2)$ is defined in [40] by:

- $V = V_1 \cup V_2$,
- $I = I_1 \wedge I_2$ provided $I_1 \wedge I_2$ is satisfiable,
- $T = T_1 \cup T_2$ where $\{start, tick\} \subset T_1 \cap T_2$ and hence $\{start, tick\} \subset T$, and
- $F = F_1 \cup F_2$ where $tick \in F_1 \cap F_2$. We call $M_1 \| M_2$ the *composite* TTM.

The above definition holds for shared variables but must be slightly modified for synchronized transitions or channels as described in [40]. Both M_1 and M_2 synchronize with respect to the *start* and *tick* transitions. The *tick* transition thus provides the composed system with a uniform notion of time.

2.4 Overview of the StateTime toolset

The StateTime toolset assists the user (a) to describe devices and systems using a graphical structured language, (b) to execute the description so as to validate that the description is a reasonable model of the actual system, and (c) to check that the description conforms to its requirements using model-checking. We give a brief description below of the main features of the toolset needed for the sequel. The reader is referred to [38] for a more complete description.

The main parts of the toolset of interest to us are the *Build* tool and its translator to the theorem prover and model-checker STeP [31]. The Build tool is a window-based front end for constructing compact visual models of real-time systems called TTMcharts. TTMcharts resemble statecharts, but with a simpler semantics and with the additional feature that transitions may have time bounds. We often use the terms TTMcharts, charts and TTMs interchangeably as the semantics of TTMcharts is based on TTMs.

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

Charts may have *data variables* which are tested and set by events. Each non-primitive serial (XOR) object has an *object variable* which is used to indicate which of its children is currently operating. As an example, consider the *plant* chart (Fig. 2) which will be described in more detail in Sect. 5.2. The plant is the parallel composition of two children called *relay* and *output* which we write as $plant = relay \| output$. The serial object *relay* has two children *closed* and *open* which are primitive. Zooming in to the *output* object indicates that it is the serial composition of the primitive object *wait* and the sub-object *update*. The *update* object is the parallel composition of the *pressure* and *power* sub-objects which is where the pressure and power sensor values are updated.

The top-level objects *relay* and *output* have object variables R and D respectively where $type(R) = \{closed, open\}$ and $type(D) = \{wait, update\}$. The state-formula defined by $(R = closed \wedge D = wait)$ describes a state in which the relay is closed and the next sensor update is two ticks away. The pressure P and power W are examples of data variables.

A serial object begins execution at its *default* indicated in bold; e.g. the default for the *output* object is ***wait*** (Fig. 2). Once a *cycle[0,0]* event is taken in the *output* object, nothing else can happen until two ticks of the clock are taken. After two but before the third clock tick, the *endupdate[2,2]* event *must* occur (in this case, there are no other events to preempt its occurrence). Before *endupdate* occurs, the pressure and power, or just one of them, or no update at all *may* occur. The source of the *endupdate* event is the structured object *update*; hence *endupdate* can be taken, no matter where execution in *update* currently resides, and preempts the internal events of *update*.

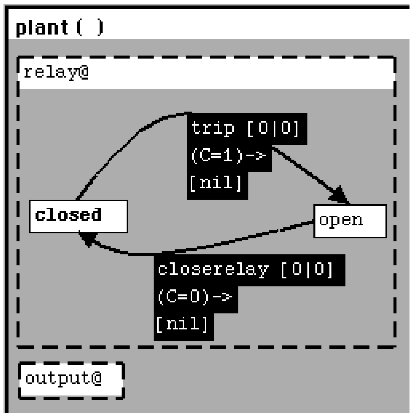
A user can describe systems incrementally by composing sub-objects together to form a super-object (bottom-up), or by decomposing a object into further sub-objects (top-down). A chart can be executed at any point in the development cycle even before it is finally fixed using the interactive simulation tool. The simulation tool displays chart trajectories, and requires user interaction to select the transition to be taken at nondeterministic selection points. The Build tool automatically translates TTMcharts into fair transition systems according to the algorithm presented in [39]; STeP [31] can then be used to model-check the chart for conformance to its specification.

The current StateTime toolset was not meant for modular systems. It suffers from various deficiencies including the fact that it does not support interface stubs, automatic generation of module environments (Sect. 3.0) and refinement. It is easy to verify standard temporal properties, but an observer must be constructed for real-time properties. How-

FIGURE 2. Plant module

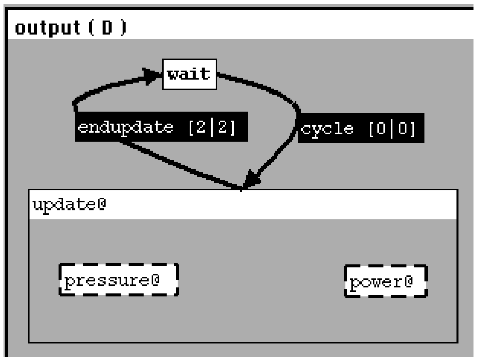
module plant(C;P,W)
in *C*: {0,1} **where** $(C = 0)$ /* relay activation. $(C = 1)$ causes the relay to open */
out *R*: {closed, open} **where** $(R = closed)$ /* Relay position variable */
out *P*: {0,1} **where** $(P = 0)$ /* pressure variable where $(P = 1)$ is high pressure*/
out *W*: {0,1} **where** $(W = 0)$ /* power variable where $(W = 1)$ is high power*/

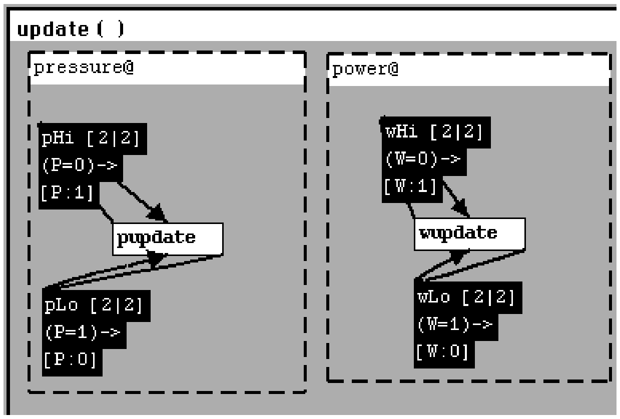
Body
private D: {wait,update}



The dotted lines in the TTM to the left indicate parallel composition, i.e. $plant = relay \| output$. The "@" symbol in *output*@ indicates that it has further internal structure. Zooming into *output*@ produces the TTM bottom left. Zooming a further level down into *update*@ produces the TTM to the bottom right, which is where pressure and power updates take place.

The lower time bound of 2 in events *pH*i, *pLo* and *endupdate* enforce the filtering assumption viz. pressure and power updates are no more frequent than once every two ticks of the clock. Before the third tick of the clock, the upper bound of *endupdate* causes a move back to *wait*, at which point there is another delay of two clock ticks before further sensor updates can occur.





Specification

$$sp(plant):\begin{bmatrix}\Box_{<20}(C = 1) \Rightarrow \Diamond_0\Box_{<20}(R = open)\\ \Box_{<2}(C = 0) \Rightarrow \Diamond_0\Box_{<2}(R = closed)\end{bmatrix} \qquad \text{(Eq. 5)}$$

The above module specification, inherited from Fig. 3 for the relay, is modularly-valid.
end module plant

ever, the tool is used in this paper for the construction of modules, their environments (done manually) and model-checking module properties. In principal, a chart when loaded into STeP can also be verified using theorem proving — however, theorem proving real-time properties proved tedious (especially on account of the need to use quantifiers). We are currently updating StateTime to fully support real-time modules and real-time formulas for both model-checking and theorem proving in a seamless fashion, based on the results of this paper.

3.0 Modules and module composition

Our notion of a module is based on the untimed reactive modules of Manna and Pnueli [32]. Although the Manna Pnueli framework has been used for real-time systems [23], the extension to their system for modules as delineated by Chang [8] is different to ours. The main differences are: (a) our modules are supported by a model-checker, (b) we provide a state-event refinement relation for modules, and (c) the reactive modules of [32] are not fully compositional as their parallel composition yields a transition system, not another module (composition of our modules yields another module). We now explain these differences in more detail.

Chang [8] advocates a restricted assumption/guarantee style, wherein the environmental assumption is stated as a restriction on the environment's next-state relation. He also presents a decision procedure in the propositional case and a deductive system for the discrete time metric temporal logic used for transition modules. Although Chang provides a deductive framework for real-time modules, he does not present model-checking algorithms and tools (which are crucial for the needs of this paper).

Chang's temporal operators are new; they are not expressed in ordinary untimed temporal logic. The transition modules of [8] must be self-disabling, i.e. once a transition is taken it cannot be again enabled (as in a self-loop). The TTM semantics of modules in this paper does not impose this restriction on module descriptions.

The untimed refinement relation of [32] will not work for real-time modules (as will be explained in Sect. 4.0). Hence, in Sect. 4.0, we introduce the necessary framework needed for real-time module refinement.

The reactive modules of [32] are not fully compositional as their parallel composition yields a transition system, not another module. In this section, we provide the notion of a fully compositional discrete time transition module (like [8]). This requires a more complete treatment of the notion of the interface stub and modes of variables in a module. It also allows our treatment to deduce the trajectories of the composite module given its submodules (Lemma 1), from which we obtain the notion that a module specification must be satisfied independently of the behaviour of the environment (Lemma 2), and finally yields the Composition Rule (Th. 2). By contrast, [32] starts with the notion of a module as given in Lemma 2 and then proceeds from there to obtain the Composition Rule.

A module $m = [is(m), bd(m), sp(m)]$ is defined by its interface stub $is(m)$, body $bd(m)$ and RTTL specification $sp(m)$:

1. The *interface stub* consists of the declaration of all the variables that are shared between module m and other modules in its environment (defined more precisely in Sect. 3.2). The stub also declares the initial values of all the shared variables. We let $is(m)$ denote the set of shared variables.

2. The *body* $bd(m)$ is a program whose statements may refer only to variables declared *private* to the body, or to variables in the interface. The set of private variables is denoted $pr(m)$. In the sequel, the body is a TTM, in which case we let $bd(m)$ denote the TTM with variables set $is(m) \cup pr(m)$. The initial condition $init(m)$ is the conjunction of all the initial conditions declared on both the private and interface variables.

3. The *specification* $sp(m)$ of the module is an RTTL formula in the shared interface variables. The specification asserts the required visible behaviour of the module.

In order to describe the behaviour of a module in an environment that may arbitrarily modify the interface variables $is(m) = i_1, \ldots, i_n$, we adjoin to the module TTM a spontaneous environmental transition $\tau_E[0, \infty]$ defined by the update function $choose(i_1, \ldots, i_n)$ (i.e. the interface variables can take on arbitrary values) while all the private variables remain unchanged, i.e. $(\forall v{:}pr(m)|v' = v)$. Thus the environmental transition may exhibit arbitrary behaviour, except that it may not modify any private variables of the module. However, shared interface variables may be changed at any point to any value in their respective types.

> **Definition 1:** [*The TTM associated with a module*] The TTM $\hat{m}$ associated with the module m is defined as $\hat{m} = (V, init(m), T, F)$ where $V = is(m) \cup pr(m)$ and $T = T_{body} \cup \{\tau_E\}$ where T_{body} is the set of transitions of the body TTM, and $F \subset T$ is the set of fair transitions of the body (note that $\{start, tick\} \subset T$). Since $\hat{m}$ is a TTM, we define $traj(m) \overset{\text{def}}{=} traj(\hat{m})$ and $des(m) \overset{\text{def}}{=} (\exists \bar{p}|des(\hat{m}))$ where $\bar{p}$ is the set of all private variables, i.e variables in $pr(m)$. (As before, we require that the timed transition model $\hat{m}$ be non-Zeno).

The succession constraint of $\hat{m}$ allows the body transitions to be interleaved in an arbitrary fashion with the environmental transition. The environmental transition thus simulates the behaviour of the module in an arbitrary context and allows the module to take stuttering steps in which none of the module private variables change from the prestate to the poststate.

The existentially quantified formula $(\exists \bar{p}|des(\hat{m}))$ in (Dfn. 1) describes the same system as $des(\hat{m})$ except with the private variables $\bar{p}$ hidden, and thus this existential formula can be considered a description of m by abstract implementation [32, p.340]. In this style of description, we may choose the most straightforward implementation of the module m and describe its operational behaviour using a TTM (e.g. if m is a buffer, then a private list variable may be used to remember sequences of messages). What makes the implementation abstract is the existential quantification of the private variables. This means that we do not require or imply in any way that the real implementation of the module should contain any of these private variables (e.g. the list variable in the case of a buffer need not be used).

> **Definition 2:** [*Modular-validity*] The RTTL formula p is modularly-valid for the module m (written $m \vDash p$) iff $(\forall \sigma{:}traj(m)|\sigma \vDash p)$.

3.1 Parallel composition of modules

Modules m_i (with variable sets V_i) for $i = 1, 2$ are said to be *compatible* with each other if:

- each module has private variables that are not variables of the other module, i.e. $pr(m_1) \cap V_2 = \emptyset$ and $pr(m_2) \cap V_1 = \emptyset$, and
- the conjunction of their initial conditions is satisfiable, i.e. $init(m_1) \wedge init(m_2)$ is satisfiable, and
- the conjunction $sp(m_1) \wedge sp(m_2)$ is satisfiable.

Compatible module composition, $m = m_1 \| m_2$, is defined by $m \overset{\text{def}}{=} [is(m), bd(m), sp(m)]$ where $is(m) \subseteq is(m_1) \cup is(m_2)$, i.e. some of the interface variables of the sub-modules may be hidden at the parent level. $bd(m) = bd(m_1) \| bd(m_2)$ is ordinary TTM composition (Sect. 2.2). Finally $sp(m) = sp(m_1) \wedge sp(m_2)$.

The private variables of the composite is $pr(m) \overset{\text{def}}{=} pr(m_1) \cup pr(m_2)$, and the initial condition is defined by $init(m) \overset{\text{def}}{=} init(m_1) \wedge init(m_2)$. The super-module $m_1 \| m_2$ is itself a module; the TTM associated with this super-module is just the TTM obtained from $bd(m_1) \| bd(m_2)$ together with the environmental transition that may change only variables in $is(m)$ (i.e. it may not change any private variables).

In the next lemma, we assume that we have two modules m_1 and m_2. If an environmental transition in a trajectory of module m_1 has the same effect on its interface variables as a transition τ_2 of m_2, then we relabel the environmental transition in the trajectory to τ_2, and the set of all the relabelled trajectories of m_1 we call $traj(\bar{m}_1)$. A symmetric definition also provides us with the set $traj(\bar{m}_2)$ of relabelled trajectories of m_2.

Lemma 1: If $m = m_1 \| m_2$ then $traj(m) = traj(\bar{m}_1) \cap traj(\bar{m}_2)$.

Proof: Let $\sigma \in traj(m)$. Trivially $\sigma \vDash init(m_1)$ and hence the initialization constraint of m_1 is satisfied. For the succession constraint, consider any position i of σ. Either the environment transition is taken at position i or some transition of m is taken. The environment transition of m may not modify any private variables of m and hence may also not modify private variables of m_1, so any environment step of m is also an environment step of m_1. If some transition of m is taken at position i, then it is either a transition of m_1 or of m_2 that is taken. Since no transition of m_2 may modify private variables of m_1, a step taken by a transition of m_2 (say τ_2) is the same as an environment step relative to m_1 (the transition τ_2 must be renamed to an environmental transition). Thus at any position either a transition of m_1 is taken or an environment transition of m_1 is taken, and hence the succession constraint $\sigma \vDash succession(m_1)$ holds. The fairness constraint of m_1 is also satisfied, as any transition of m_1 that is enabled infinitely often but not taken would also violate the fairness constraint of m. The ticking constraint of m_1 is also satisfied, for suppose there is a position of σ beyond which there is no tick of the clock for m_1, then the ticking constraint for m would also be violated. If a transition of m_1 violates its bound constraint, then the bound constraint on transitions of m will also be violated. Hence σ must also satisfy the bound constraint of m. Since σ satisfies the initialization, succession, fairness, ticking and bound constraints of m_1, it follows that $\sigma \in traj(\bar{m}_1)$ holds. By symmetry it also follows that $\sigma \in traj(m_2)$ holds. Thus $\sigma \in traj(\bar{m}_1) \cap traj(\bar{m}_2)$.

For the converse, let $\sigma \in traj(\bar{m}_1) \cap traj(\bar{m}_2)$. At any position of σ either a transition of m_1 or of m_2 is taken, in which case the same transition belonging m is taken, or an environment transition that is an environment transition of both m_1 and m_2 is taken. This environment step must also be an environment step of m as no private variables of m_1 and m_2 could have been changed. We can make similar arguments as before for the other constraints but in the converse direction. Hence $\sigma \in traj(m)$. ■

Lemma 2: Let modules m_1 and m_2 be compatible. Then

(a) $[(m_1 \vDash sp(m_1)) \wedge (m_2 \vDash sp(m_2))] \rightarrow [(m_1 \| m_2) \vDash (sp(m_1) \wedge sp(m_2))]$, and

(b) For a module m, $[m \vDash p] \rightarrow (m \| m') \vDash p$ for any compatible module m' and RTTL property p.

Proof: Follows directly from Lemma 1. ■

Recall that a property is modularly-valid only if it is satisfied by all trajectories of the module. Lemma 1 tells us that the trajectories of the super-module are always a subset of those of its sub-modules. This means that a valid specification of a sub-module must also be valid for the super-module (Lemma 2a), and that a module specification remains valid no matter what the behaviour of its environment is, provided the environment respects the compatibility constraints (Lemma 2b).

Theorem 2: [*Composition Rule*].
Let m_1 and m_2 be any two compatible modules and let the general-validity given by $\models sp(m_1) \wedge sp(m_2) \rightarrow r$ hold. Then $[m_1 \models sp(m_1)] \wedge [m_2 \models sp(m_2)] \rightarrow [m_1 \parallel m_2 \models r]$.

Proof: Follows directly from Lemma 2 and temporal logic. ■

As mentioned in the introduction, the Composition Rule can be used bottom-up or top-down. In the bottom-up method, pre-existing implemented "off the shelf" modules can be combined into a super-module that satisfies a system requirement *r*. In the top-down method, we proceed as follows:

1. The system architect decomposes the system under design (sud) into modules m_1 and m_2 by:
   (a) designing compatible interface stubs $is(m_1)$ and $is(m_2)$, and
   (b) designing module specifications such that $sp(m_1) \wedge sp(m_2) \rightarrow r$.
2. The architect gives each module interface and specification to a programmer. It is the job of the programmer to develop the module body so that the specification is modularly-valid. For example, if the programmer is given $is(m_1)$ and $sp(m_1)$ for the first module, he must design a body $bd(m_1)$ so that $m_1 \models sp(m_1)$ where the module m_1 is fully described by $m_1 = [is(m_1), bd(m_1), sp(m_1)]$.
3. The required system is then $sud = m_1 \parallel m_2$ which is guaranteed by the Composition Rule to conform to the requirement r.

Parts of the development method can be automated by using a combination of model-checking for proving modular-validity (step 2), and deductive theorem proving techniques can be used for proving that the system requirement is a consequence of the module specifications (step 1b).

A compositional proof has the following outline:

1.	$m_1 \models p_1$	p_1 is modularly-valid for m_1 (by model-checking)
2.	$m_2 \models p_2$	p_2 is modularly-valid for m_2 (by model-checking)
3.	$\models (p_1 \wedge p_2) \rightarrow r$	general-validity (deductive theorem proving)
4.	$m \models r$	1, 2, 3 and the Composition Rule where $m = m_1 \parallel m_2$

In the sequel, we will leave out the module satisfaction symbol (except for its appearance in the last line) and write the above proof as:

1.	p_1	p_1 is modularly-valid for m_1
2.	p_2	p_2 is modularly-valid for m_2

3. $(p_1 \wedge p_2) \rightarrow r$ general-validity

4. $m \models r$ 1, 2, 3 and the Composition Rule where $m = m_1 \| m_2$

By Lemma 2 (b), once we know that the context of the proof is the module m, then any specification of a sub-module of m will also hold for m, and hence there is no need to indicate which sub-module specification we are dealing with.

3.2 Modes of interface variables

The interface stub of a module defined in the previous subsection consists of a set of typed shared variables with their initial conditions. We can provide more structure and flexibility to the interface specification which will enhance the user's ability to understand a module.

The additional structuring mechanism is provided by describing the *modes* of the shared variables. A variable in the interface stub is either **in** (the module body can read the variable but not write to it), **out** (the environment can read the variable but not write to it), or **share** (both the body and the environment have write access):

interface_stub ::= {*mode* {*variables*}$^+$ **:** *type* [**where** *init*]}*
mode ::= {**in** | **out** | **share**}

If a module m has a declaration "**out** y_1", then no other module in the environment of m may have a writing reference to the variable y_1. If two (or more) modules each write to y, then they must each have the declaration "**share** y_1", thus indicating that the external environment may also change y_1.

Let the variables in the interface stub be $\bar{y} = y_1, \ldots, y_j, y_{j+1}, \ldots, y_k$, where $y_1, \ldots, y_j$ are the "**in**" and "**share**" variables (i.e. all variables whose value may be changed by the environment), and where $y_{j+1}, \ldots y_k$ are the remaining interface variables (the "**out**" variables that the environment does not change). We often refer to the module by $m(y_1, \ldots, y_j; y_{j+1}, \ldots y_k)$, where the semicolon separates the **out** variables from those that the environment can read and modify (the **in** and **share** variables).

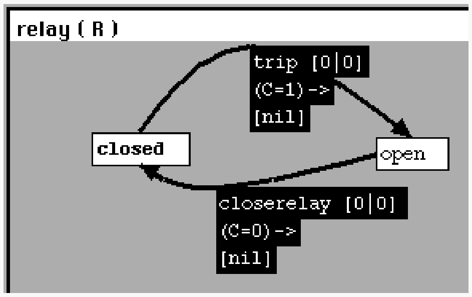
> **Definition 5:** Two modules m_1 and m_2 are *interface compatible*, provided each variable $v \in is(m_1) \cap is(m_2)$ satisfies the following constraints: the types declared for v in both interfaces match, the conjunction of their **where** clauses (supposed *true* when not specified) is satisfiable, and if one of the declarations specifies an **out** mode, then the other specifies an **in** mode.

The reactor trip relay module *relay* (taken from the example in Sect. 5.2) is shown in Fig. 3. When the command to open the relay ($C = 1$) comes from the environment, then the relay is immediately opened ($R = open$) before the next clock tick, thus shutting down the reactor. The specification of the relay (see (Eq. 6) in Fig. 3) does not contain the next operator $\bigcirc$ in the consequent; instead, the operator $\Diamond_0$ is used. This is because the trajectories of a module may have environmental steps that leave the state unchanged. Specifications must therefore allow such "stuttering" steps otherwise the specification will not be modularly-valid.

FIGURE 3. The relay module

module relay(C;R)
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 object variable that is exported as readonly output */

Body TTMchart (using the StateTime Build tool)



Note: The transitions *trip[0,0]* and *closerelay[0,0]* are immediate transitions, i.e. their time bounds force them to occur before the next clock tick once they become enabled. The *guard* of the *trip* transition is *(C=1)* and its enabling condition is *(C=1,R=closed)*. In Build expressions, the comma is used for conjunction and the semicolon for disjunction. The update function *nil* in the *trip* transition indicates that no *data variable* (e.g. *C*) is changed; however, when the *trip* transition is taken the relay *R* is changed to *open*. This module has no private variables.

Specification:

$$sp(relay): \begin{bmatrix} \Box_{<20}(C = 1) \Rightarrow \Diamond_0 \Box_{<20}(R = open) \\ \Box_{<2}(C = 0) \Rightarrow \Diamond_0 \Box_{<2}(R = closed) \end{bmatrix} \quad \text{(Eq. 6)}$$

/* **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). The above specification is modularly-valid */
end module relay.

3.3 A small example of compositional reasoning

The module $majorVote(C_1, C_2, C_3;C)$ (Fig. 4) is part of the DRT controller which will be discussed in the sequel. The controller consists of three independent microprocessors, each one with independent sensors of reactor power and pressure. Each microprocessor controller $micro_i$ 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 1 when the majority of the microprocessor vote for opening the relay (i.e. when $C_1 + C_2 + C_3 \geq 2$). The specification $sp(majorVote)$ can be shown to be modularly-valid by model-checking.

The relay module (Fig. 3) and the voting module (Fig. 4) are interface compatible. We may therefore use the modularly-valid module specifications (Eq. 6) and (Eq. 7), and the Composition Rule to prove the validity of

$$[majorVote \parallel relay] \vDash p \quad \text{(Eq. 8)}$$

where *p* is defined by:

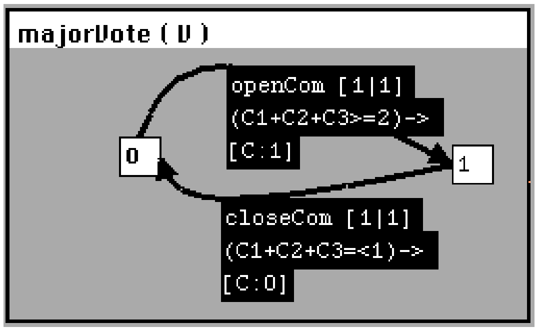
$$p: \begin{bmatrix} [\Box_{<20}(C_1 + C_2 + C_3 \geq 2) \Rightarrow \Diamond_{\leq 1} \Box_{<20}(R = open)] \\ \wedge [\Box_{<2}(C_1 + C_2 + C_3 \leq 1) \Rightarrow \Diamond_{\leq 1} (R = closed)] \end{bmatrix} \quad \text{(Eq. 9)}$$

The proof of the first conjunct of (Eq. 9) is as follows:

FIGURE 4. Module for majority voting logic

module majorVote(C1,C2,C3;C)
with bitType={0,1}
in C_1, C_2, C_3 : bitType; /* 1 stands for a vote to open the relay, and 0 to close the relay. */
out C : bitType **where** initially $(C = 0)$ /* Only *majorVote* can write to C to set the relay*/

Body
private V : bitType **where** initially $(V = 0)$ /* majority vote object variable*/.



Specification

$$sp(majorVote) : \begin{bmatrix} \Box_{<20}(C_1 + C_2 + C_3 \geq 2) \Rightarrow \Diamond_{\leq 1} \Box_{<20}(C = 1) \\ \wedge \Box_{<2}(C_1 + C_2 + C_3 \leq 1) \Rightarrow \Diamond_{\leq 1} \Box_{<2}(C = 0) \end{bmatrix} \qquad \text{(EQ 7)}$$

/*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 20 ticks, 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 20 ticks of the clock. The second line states a similar specification for the command to close the relay. */

end module majorVote

1. $\Box_{<20}(C_1 + C_2 + C_3 \geq 2) \Rightarrow \Diamond_{\leq 1} \Box_{<20}(C = 1)$ by modular-validity of (Eq. 7)
2. $\Box_{<20}(C = 1) \Rightarrow \Diamond_0 \Box_{<20}(R = open)$ by modular-validity of (Eq. 6)
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. $majorVote \parallel relay \models \Box_{<20}(C_1 + C_2 + C_3 \geq 2) \Rightarrow \Diamond_{\leq 1} \Box_{<20}(R = open)$ (1), (4) and Composition Rule

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

The Composition Rule provides a powerful technique for beating combinatorial explosion of states. To verify a global requirement *r* 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 objects one at a time, provided we can show that the object specifications entail the global requirement.

The modular-validity of module specifications for a module m can be determined by applying the model-checking and theorem proving tools of StateTime (Sect. 2.4) to the TTM $\hat{m}$ that corresponds to m. For example, the relay module specification $sp(relay)$ in Fig. 3 can be proved modularly-valid by model-checking the set of transitions associated with the body together with the nondeterministic environmental transition with update function *choose(C)*, which allows the input variable C to vary arbitrarily.

In the above relay example, an unrestricted environment was used to check the modular-validity of the module specification. This is not always possible as an unrestricted environment can sometimes generate larger intermediate reachability graphs than the reachability graph obtained when the environment is limited to a known set of fixed modules. This is because certain states of the module in an unrestricted environment may be unreachable in the composite. There are two ways to address this issue: either (a) decompose the module into smaller sub-modules where an unrestricted environment will not be problematic, or (b) restrict the environment of the module to the actual environment in which the module is expected to operate.

The easiest way to restrict the environment involves the use of *conditional specifications* for the module of the form $Env \to r$ which asserts that if the environment is assumed to behave according to the RTTL formula Env then the module is guaranteed to behave according to the RTTL formula r. In other frameworks, such conditional specifications are called assumption/guarantee properties [22], and special rules are provided for reasoning about them. In our framework, conditional specifications are no different from any other module specifications. Our purpose will be to show that $Env \to r$ is modularly-valid for the module m, i.e. $m \vDash Env \to r$. This does not contradict our definition that a module specification should hold independently of what the environment does. The property r will indeed hold true only if the module environment behaves according to Env. However, $Env \to r$ holds for the module in any environment; this is because if the environment does not satisfy Env, then r need not hold true [32, p.356].

In the sequel, we deal with modules that are intended to work in fixed environments. For example, the environment of the DRT *controller* module (Sect. 5.0) is the *plant* which will remain fixed throughout the design. Consider a conditional specification $des(plant) \to r$ for one of the controller sub-modules m which asserts that if the plant (which is the environment of m) behaves according to $des(plant)$ then m will behave according to r. To verify the modular-validity $m \vDash des(plant) \to r$ in an unrestricted environment in which the plant output variables can take on any value at any moment, will generate a larger reachability graph than necessary because there will be states that are not reachable in practice. The actual plant sensors are filtered and hence change only every two ticks of the clock. Thus we do not need to consider all the possibilities generated by continuously changing sensor values. Instead, we can verify $[plant \,\|\, m] \vDash r$ which will involve a smaller reachability graph in which plant changes occur only every two ticks. The following theorem justifies this procedure.

Theorem 3: Let m_1 and m_2 be two compatible modules and p an RTTL formula in the interface variables. Then $[m_1 \vDash des(m_2) \to p] \equiv [m_1 \,\|\, m_2] \vDash p$.

Proof:

$[m_1 \vDash des(m_2) \to p]$

$\equiv$ < (Th. 1)(a) >

$\vDash des(m_1) \to [des(m_2) \to p]$

$\equiv$ < propositional temporal logic >

$\vDash des(m_1) \wedge des(m_2) \to p$

$\equiv$ < Composition Rule and $m_i \vDash des(m_i)$ holds for $i = 1, 2$ by (Th. 1)(b) >
$[m_1 \parallel m_2] \vDash p$. ■

4.0 Module refinement

If a module m has been implemented with a given body, under what conditions can we replace the body with a new one while still retaining the same observed timed behaviour at the interface stub? One possibility is to use the notion of program equivalence of untimed concurrent programs developed in [32, p46]. However, this notion of equivalence will not work for our real-time reactive modules.

Consider a program with two variables x and y. In [32, p46], a sub-sequence $\langle \varepsilon{:}start, x{:}0, y{:}0 \rangle \langle \varepsilon{:}tick, x{:}0, y{:}0 \rangle \langle \varepsilon{:}tick, x{:}0, y{:}0 \rangle \langle \varepsilon{:}\tau, x{:}1, y{:}2 \rangle$ is reduced to $\langle \varepsilon{:}start, y{:}0 \rangle \langle \varepsilon{:}tick, y{:}0 \rangle \langle \varepsilon{:}\tau, y{:}2 \rangle$ if we want to observe variable y. We have thus lost a record of one of the clock ticks, because in the refinement relation of [32], program states that are identical to their predecessors are omitted from the sequence. But, in real time systems, it is essential that the reduced system show the same *timed* behaviour as the original system. We will thus need to define a notion of observational equivalence that takes into account state (data such as the value of y) as well as events (ticks of the clock). In this section, we adapt the *state-event* notion of *observational equivalence* developed in [26,27,28] to the needs of real-time reactive modules. Because we need to deal with both states and events, we also cannot just use the standard event-based notion of bisimulation [33], as will be explained in this section.

Consider two modules that have the same interface stub but different bodies. For such modules we will define a notion of module observational equivalence that is *compositionally consistent* and preserves *any* stuttering invariant RTTL module specification (detailed explanation follows below). Thus the first body can be replaced by the second with a guarantee that any module specification that holds for the first will also hold for the second, and vice versa. Observational equivalence will allow us to *refine* an abstract module into one closer to code implementation. The abstract module may have a substantially smaller state space than the refinement and hence will be more amenable to model-checking.

Informally, if a module m_1 is equivalent to a module m_2 having the same interface stub (written $m_1 \approx m_2$) then m_1 preserves the timed behavior of m_2 over the interface variables. We want a notion of observational equivalence that only distinguishes between the two modules if the distinction can be detected by an external agent interacting with each of them. The agent can observe any of the interface variables and the *start* transition and *tick* of the conceptual global clock, but not any of the private variables or internal transitions which are unobservable to the external agent. We call such internal unobservable actions λ-transitions. Although an external agent may not be able to observe an internal transition itself, it may be able to observe the effects of the internal transition (e.g. if the internal transition changes one of the interface variables).

4.1 Observation equivalence of TTMs

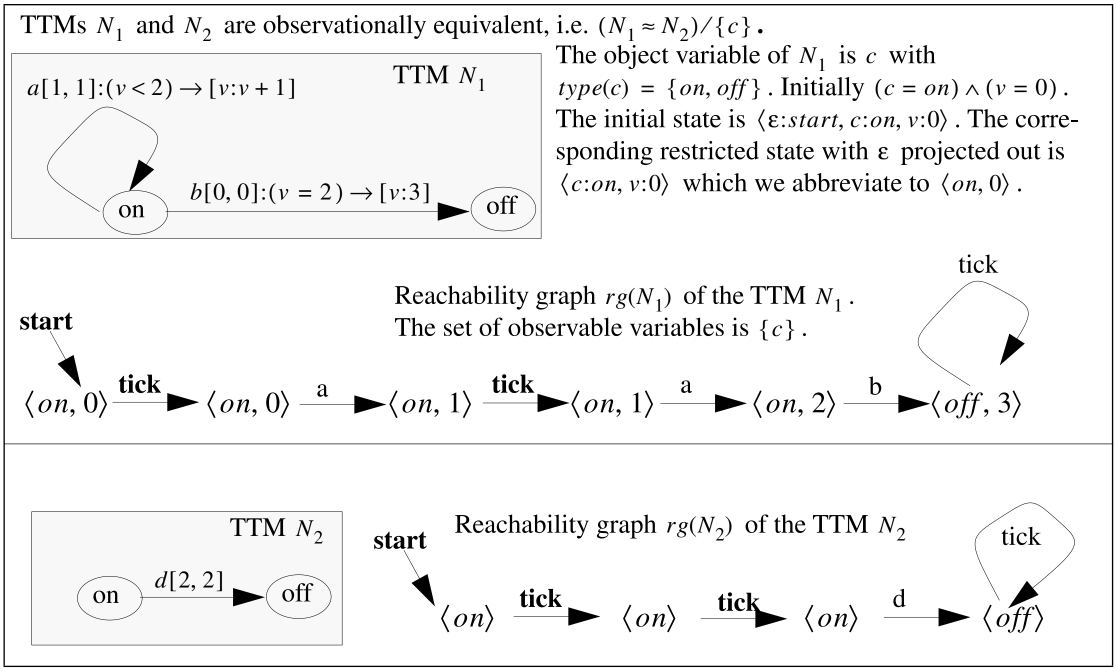
In [37], an algorithm is given for constructing the reachability graph of a TTM. The reachability graph is used as the basis for model-checking RTTL formulas, as maximal fair paths in the reachability graph correspond to TTM trajectories.

We illustrate the concept of a reachability graph by referring the reader to the sample TTM N_1 with variables set $V = \{\varepsilon, c, v\}$ as shown in Fig. 5. The reachability graph of N_1 $rg(N_1) = (Q, T, R, q_1)$ is also shown in Fig. 5. The reachability graph is a labelled transition system with state set Q, transition label set $T = \{start, tick, a, b\}$, R is a set $\{R_\tau | \tau \in T\}$ of binary relations on Q, and the initial state is $q_1 = \langle c{:}on, v{:}0\rangle$.

If $q, q' \in Q$ and $\tau \in T$ then $R_\tau(q, q')$ holds precisely when $s' \in \tau(s)$ (i.e. s' is a τ-successor of s) where q, q' are the restrictions of s, s' respectively, and s, s' both occur in trajectories of the TTM N_1. Let $\tau(q, q')$ be an abbreviation for $R_\tau(q, q')$ which is called a τ-*step* from q to q'. The notation $tick(q_1, q_2)a(q_2, q_3)tick(q_3, q_4)\ldots$ denotes a sequence of steps in the graph. Any maximal sequence of steps in the reachability graph corresponds to a trajectory of the TTM respecting the initialization, succession, fairness, and bound constraints (Sect. 2.2).

The timed behaviour of the TTM N_2 in Fig. 5 is equivalent to N_1 with respect to the observable variable c (in a sense to be made precise in the sequel). In this weakly observable setting, the $tick$ and $start$ transitions are observable but no other transitions are visible to an external agent. The observable variables set is $O = \{c\}$; we require that the variable c be in the variables set of both TTMs. The TTM N_2 is much simpler than N_1 and has a smaller reachability graph (Fig. 5). We therefore call N_2 an *abstract* specification[8] of the concrete *refinement* N_1.

FIGURE 5. Observably equivalent TTMs



TTMs N_1 and N_2 are observationally equivalent, i.e. $(N_1 \approx N_2)/\{c\}$.

The object variable of N_1 is c with $type(c) = \{on, off\}$. Initially $(c = on) \wedge (v = 0)$. The initial state is $\langle \varepsilon{:}start, c{:}on, v{:}0\rangle$. The corresponding restricted state with ε projected out is $\langle c{:}on, v{:}0\rangle$ which we abbreviate to $\langle on, 0\rangle$.

Reachability graph $rg(N_1)$ of the TTM N_1. The set of observable variables is $\{c\}$.

Reachability graph $rg(N_2)$ of the TTM N_2

[8] In this section, we show that a TTM description of a concrete system N_1 meets its abstract TTM specification N_2 is by showing that N_1 is equivalent to N_2 on the observable variables. Although this approach is natural in many cases, we do not thereby imply that equivalence of TTMs is always the right way to express conformance. Temporal logic is often more convenient for expressing a *partial* specification, i.e. a property which should be satisfied by a system but which does not fully determine its observable behavior. An example of a partial specification is $\square_{<2}(c = off)$ (where c is the N_1 object variable in Fig. 5) which specifies that N_1 should not be turned on any sooner than two ticks of the clock.

For the precise definition of observation equivalence, we need the concepts of state projection operators and unobservable λ-transitions. For a set of observable variables O of a given TTM, the *observable state projection operator* $\tilde{O}$ tells us when states q_1 and q_2 agree when restricted to their observable variables. For example, if we are given the states $q_1 = \langle c{:}on, v{:}0\rangle$, $q_5 = \langle c{:}on, v{:}2\rangle$ and the observable variables set $O = \{c\}$, then

$$\tilde{O}(q_1) = \tilde{O}(q_5)$$

as they agree on the c component of the state.

An external agent interacting with the TTM N_1 can observe the *start* and *tick* transitions; but the other transitions a and b are unobservable. Similarly, the transition d in N_2 is unobservable. We will relabel the edges of reachability graphs so that all unobservable transitions are called λ. Although the λ-transition itself is unobservable to an external agent, its effect may be observable (e.g. when the transition is taken it may change an observable variable); however, the external agent is unable to tell which transition caused that effect.

Definition 6: [*State-event labelled transitions systems SELTS*] Let $M = (V, I, T, F)$ be a TTM, and let $O \subset V$ be a given set of observable variables. Let the reachability graph of M be $rg(M) = (Q, T, R, q_0)$ where Q is a countable set of states and $q_0 \in Q$ is the initial state. Then $G_M = (Q, T_\lambda, \bar{R}, q_0, O)$ is a labelled transition system, called a state-event labelled transition system (or SELTS), where $T_\lambda = \{start, tick, \lambda\}$ and $\bar{R} = \{R_\tau | \tau \in \{start, tick\}\} \cup \{R_\lambda\}$ where:

$$R_\lambda \overset{\text{def}}{=} \bigcup_{\tau \in T - \{start,\, tick\}} R_\tau \qquad \text{(Eq. 10)}$$

(Eq. 10) achieves the required relabelling, i.e. all transitions in the reachability graph other than *start* and *tick* are now relabelled to the unobservable λ-transition in the corresponding SELTS. The following definition is needed for the *weak state-event bisimulation*:

Definition 7: The unobservable move $\hat{\lambda}(q, q')$ is defined by:

$$\hat{\lambda}(q, q') \overset{\text{def}}{=} (q = q')$$
$$\text{or } \exists q_1 \ldots q_n \begin{bmatrix} \lambda(q, q_1)\lambda(q_1, q_2)\ldots\lambda(q_{n-1}, q_n)\lambda(q_n, q') \\ \text{and } \forall j | 1 \le j \le n\text{: } \tilde{O}(q_j) = \tilde{O}(q) = \tilde{O}(q') \end{bmatrix}$$

The action of taking an observable step $\alpha \neq \lambda$ (i.e. α is either *start* or *tick*) in a SELTS that has (possibly empty) sequences of unobservable steps on both sides is defined by:

$$\tilde{\alpha}(q, q') \overset{\text{def}}{=} \exists q_1, q_2 \begin{bmatrix} \hat{\lambda}(q, q_1)\alpha(q_1, q_2)\hat{\lambda}(q_2, q') \\ \text{and } \quad \tilde{O}(q_1) = \tilde{O}(q) \wedge \tilde{O}(q_2) = \tilde{O}(q') \end{bmatrix}$$

We also define a similar move for the unobservable λ-transition (which may or may not cause a change in the observable variables) by:

$$\tilde{\lambda}(q, q') \overset{\text{def}}{=} (q = q') \vee \exists q_1, q_2 \begin{bmatrix} \hat{\lambda}(q, q_1)\lambda(q_1, q_2)\hat{\lambda}(q_2, q') \\ \text{and } \quad \tilde{O}(q_1) = \tilde{O}(q) \wedge \tilde{O}(q_2) = \tilde{O}(q') \end{bmatrix}$$

We are now ready to define the notion of a weak *state-event* bisimulation relation. In the weakly observable setting with unobservable λ-steps, the steps $\tau(q, q')$ and $\tilde{\tau}(q, q')$ are indistinguishable, producing the same observations (or possibly lack of observation in the case of a λ move).

Definition 8: [*Weak state-event bisimulation*] Let $G_{M_i} = (Q_i, T_\lambda, \bar{R}_i, q_{i0}, O)$ be state event labelled transition systems for the TTMs M_i for $i = 1, 2$ with a common observable variables set O. Then the relation $S \subseteq Q_1 \times Q_2$ is a weak state-event bisimulation relation if $\forall (q_1, q_2) \in S$: $\tilde{O}(q_1) = \tilde{O}(q_2)$ and $\forall \tau \in T_\lambda$:

- $\tau(q_1, q_1')$ implies $(\exists q_2' | \tilde{\tau}(q_2, q_2') \wedge (q_1', q_2') \in S)$
- $\tau(q_2, q_2')$ implies $(\exists q_1' | \tilde{\tau}(q_1, q_1') \wedge (q_1', q_2') \in S)$

The above definition of bisimulation can be paraphrased by saying that two states are weakly bisimilar if any move from one of the states to a new state can be matched by the other state making a move, or sequence of moves, producing the same observations on both the observable variables and the observable transitions (*start* and *tick*) and reaching a state that is weakly bisimilar to the state reached from the first state.

The standard notion of bisimulation [33] is defined with respect to the events of a labelled transition system. While it is possible to describe systems using only state information or event information, there are many applications where the use of both state and event information is quite natural. The above notion of (weak) bisimulation is defined not only with respect to the observable events of the labelled transition system (needed to maintain a global notion of time via the clock *tick*), but also with respect to the states of the labelled transition system (needed for dealing with properties involving the observable variables). For TTMs that must synchronize with each other via shared events (in addition to *start* and *tick*), the set T_λ in (Dfn. 6) can be expanded quite naturally to include any such additional synchronized events without the need to change the definition of bisimulation.

Since weak bisimulations are closed under union, there is always a largest weak bisimulation relation (which we denote by the infix operator $\approx$) relating the states of M_1 to that of M_2 for an observable set of variables O. Thus if q_1 (respectively q_2) is a state of the reachability graph of M_1 (respectively M_2) then we can write $(q_1 \approx q_2)/O$ whenever $(q_1, q_2) \in S$. This leads to the notion of state-event equivalence of TTMs:

Definition 9: $[(M_1 \approx M_2)/O]$ Let M_1 (with initial state q_1) and M_2 (with initial state q_2) be two TTMs with variables sets V_1 and V_2 respectively. Let $O \subseteq V_1 \cap V_2$ be a given observable set of variables. Then M_1 and M_2 are called *state-event equivalent over O* (written: $(M_1 \approx M_2)/O$) provided $(q_1 \approx q_2)/O$.

Where the observable set of variables is fixed from the context to O, we write $M_1 \approx M_2$. For the example TTMs in Fig. 5 with observable variables set $O = \{c\}$, we have that $N_1 \approx N_2$.

For finite state TTMs, [28] provides an efficient polynomial time algorithm for checking the equivalence of two TTMs. For possibly infinite state TTMs, [27] presents equivalence preserving transformations. The following theorems indicate the usefulness of state-event equivalence [26].

Lemma 3: (corollary of Lemma 2 in [26])
Given TTMs M_1, M_2, P_1, P_2 all having the same observable variables set, then $(M_1 \approx P_1 \wedge M_2 \approx P_2) \rightarrow [M_1 \| M_2] \approx [P_1 \| P_2]$

Thus, state-event equivalence of TTMs is compositionally consistent, i.e. the designer can replace a TTM with an equivalent refinement with a guarantee that the observed time behavior will be unchanged.

The set of *SESI* (state-event stuttering invariant) temporal logic formulas is defined in [26]. We will only need a subset of SESI formulas for the sequel, which we now define. An atomic SESI formula *atomic_sesi* of a module m is any state-formula, having no occurrences of the transition variable ε, and whose free variables are the observable variables, i.e. the variables in $is(m)$. A SESI formula is defined by:

$$sesi ::= atomic\_sesi \mid sesi \vee sesi \mid \neg sesi \mid sesi\,\mathcal{U}\,sesi \mid sesi\,\mathcal{U}_{[l,u]}\,sesi \mid \Diamond(\varepsilon = tick) \qquad \text{(Eq. 11)}$$

The formula $\Diamond_0 p$ is SESI as it is derived from the bounded *until* operator which itself is SESI. Also $\Box\Diamond(\varepsilon = tick)$ is SESI because all the other temporal logic operators, except for *next*, can be obtained from the *until* operator. The $\Diamond_0$ operator can usually replace the *next* operator. It is shown in [26] that some formulas involving the *next* operator are also SESI, but we will not need these for the sequel.

Lemma 4: (corollary of Theorem 3 in [26]) Let s be a SESI formula with a given observable variables set O. If M_1 and M_2 are TTMs such that $(M_1 \approx M_2)/O$ then: $[M_1 \vDash (\Box\Diamond(\varepsilon = tick) \rightarrow s)] \equiv [M_2 \vDash (\Box\Diamond(\varepsilon = tick) \rightarrow s)]$.

The above lemma is significant for model-checking. We may check an abstraction M_2 for conformance to s rather than its more complex refinement M_1, with a guarantee that s will also hold for the refinement, provided the TTMs are non-Zeno.

4.2 Observation equivalence of modules

The behaviour of a module m was defined in Sect. 3.0 with the help of an associated TTM $\hat{m}$, which is the composition of the body TTM and an environment transition that arbitrarily changes interface variables $is(m)$.

Definition 10: [*state-event equivalence of modules*] Let m_1 and m_2 be two modules having precisely the same interface variables (i.e. $is(m_1) = is(m_2)$). The observable variables set O of these modules is defined as $O = is(m_1) = is(m_2)$. The corresponding reachability graph of each of these modules is $rg(\hat{m}_i) = (Q_i, T, R_i, q_i)$ for $i = 1, 2$ from which their corresponding SELTS can be obtained as in (Dfn. 6). The state event equivalence of these modules is then defined by: $[m_1 \approx m_2] \overset{\text{def}}{=} (\hat{m}_1 \approx \hat{m}_2)/O$.

As with TTMs, one may check the conformance of an abstract module for conformance to its specification with the guarantee that the refinement will also satisfy its specification, as stated in the following theorem.

Theorem 4: [*Refinement Rule*] Let s be an arbitrary SESI formula for non-Zeno modules m_1 and m_2 having the same interface variables such that $m_1 \approx m_2$. Then: $[m_1 \vDash s] \equiv [m_2 \vDash s]$.

Proof: Since $m_1 \approx m_2$ we have that $(\hat{m}_1 \approx \hat{m}_2)/O$ where $\hat{m}_i$ is the TTM corresponding to the module (Dfn. 1) for $i = 1, 2$ and $O = is(m_1) = is(m_2)$. By Lemma 4, it follows that $[\hat{m}_1 \vDash \Box\Diamond(\varepsilon = tick) \to s] \equiv [\hat{m}_2 \vDash \Box\Diamond(\varepsilon = tick) \to s]$. Since the modules are non-Zeno, $[\hat{m}_1 \vDash s] \equiv [\hat{m}_2 \vDash s]$ holds. Hence, by the definition of modular-validity (Dfn. 2) $[m_1 \vDash s] \equiv [m_2 \vDash s]$ holds as required. ■

5.0 Modular Design of the delay reactor trip (DRT)

Industrial reactive systems are often specified using a combination of timing diagrams, pseudocode and careful English narrative. This has the considerable advantage that it is accessible and intelligible to a wide community. It has the disadvantage that even the most lucid informal descriptions are prone to omissions and ambiguities. More importantly, conformance analysis can only be undertaken in a more precise setting.

In this section we describe an example taken from the actual requirements document for the shutdown system of an industrial nuclear reactor. We translate the informal descriptions and requirements into precise counterparts in the TTM/RTTL framework, and then use the modular development method developed in this paper to design the system and check its conformance to requirements. The abstract design so obtained can then be refined down to a format close to pseudocode suggested in the original requirements document. This is not the way the original problem was presented. Originally, the pseudocode was a given, and the engineers wanted to know if the pseudocode satisfied the informal requirements as presented in the timing diagram. This reverse engineering problem can be solved using the same compositional and abstraction techniques but working bottom-up (see [38] for the reverse engineering problem).

5.1 Informal description of the problem

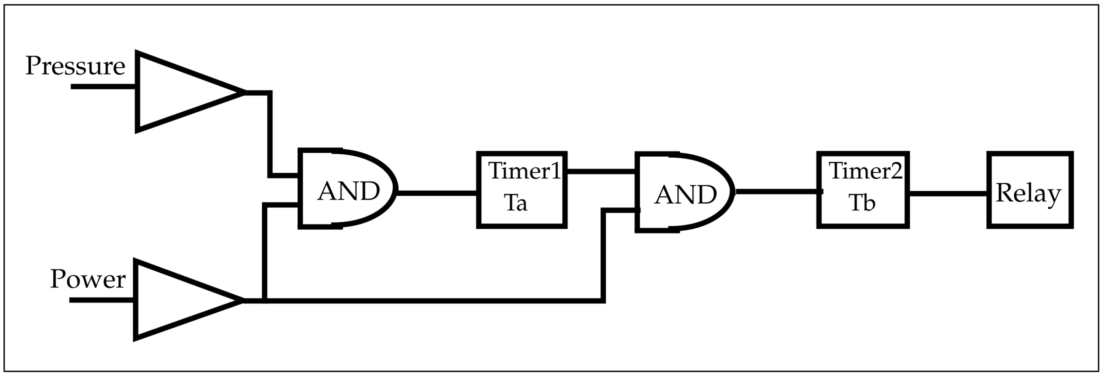
In early nuclear reactors, the shutdown systems were constructed of analog devices. The analog control had the virtue of being simple to understand but inflexible, unable to perform system checks and not always reliable. It was felt that the situation could be improved by installing computerized control with at least two independent shutdown systems, designed by different teams, each shutdown system itself having 3-version control and majority voting logic [43].

The delayed reactor trip (DRT) problem was first described by Lawford *et. al.* [27]. Lawford developed behaviour preserving transformations for timed transition models (TTMs) with which he was able to discover a flaw in the proposed design [25] involving a single controller. However, the transformational theory cannot be fully automated as no set of transformations is complete for proving observation equivalence between the actual implementation and its abstract specification. In [38], the StateTime toolset was used to verify the single controller case, where it also helped to find a bug in the original specification. A corrected version of the pseudocode was shown to conform to its requirements by model-checking.

In this paper we consider the case of 3-version control using a majority voting circuit to determine control actions. The StateTime toolset was not able to model-check the complete system due to a combinatorial explosion of states. However, using a combination of model-checking and deductive techniques in the modular framework, the conformance of the systems to its requirements can be demonstrated.

The DRT for nuclear reactors used to be implemented in hardware using timers, comparators and logic gates similar to the timing diagram shown in Fig. 6. The new DRT system is implemented on microprocessors. Digital control systems provide cost savings and flexibility over the hardware implementation. However, the question now is whether the new microprocessor based software controller satisfies the same specifications as the old hardware implementation.

FIGURE 6. Analog implementation of the delay relay trip timing.



The hardware version of the controller implements the following informal requirements[9]:

> **[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. After 3 seconds, Timer1 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. By the same token, each time the reactor is unnecessarily 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:

[9] In the sequel, we assume that we are to satisfy the original hardware requirements, because this is the way the original industrial requirements document posed the problem, and we wanted to show that formal methods could deal with the problem as posed. Using the original requirements also allows the design method of this paper to be directly compared to the reverse engineering problem of [38]. It could be argued that these original requirements are biased by the hardware implementation, and simpler less strict requirements can therefore be obtained.

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

In the actual DRT, there are three identical microprocessors that have independent sensors for power and pressure. The final decision on when to shut down the reactor is based on a majority vote of the three microprocessors.

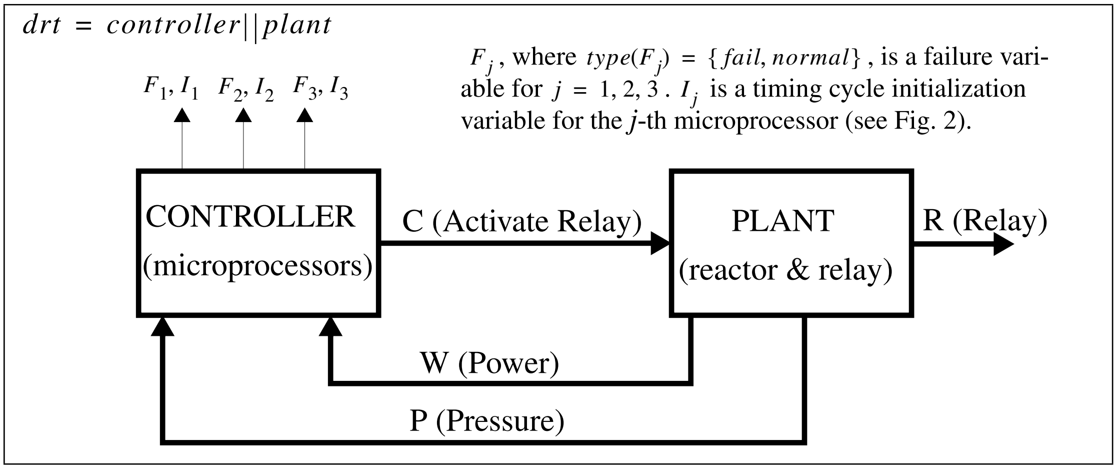
The code is to be implemented on a microprocessor with a cycle time of 100ms. The microprocessor samples the inputs (pressure P and power W) and passes through a block of code every 0.1 seconds. It is assumed that the input signals have been properly filtered and that the sampling rate is sufficient to ensure adequate control. In the formal model, one tick of the clock will represent 100ms.

5.2 Formal requirements

The first step is to decompose the *drt* into two parallel modules the *plant* and the *controller*, i.e. $drt = plant \| controller$. The plant corresponds to the part of the system that is fixed and known. The controller is the part of the system that must be designed.

The observable variables of the DRT are shown in the data flow diagram of Fig. 7. The plant outputs are the relay position (R), power (W) and pressure (P) variables. The input to the plant (C) is a relay activation variable that can be used to force the relay to open or close. In the absence of control, the plant can behave unsafely. For example, if pressure and power both go to unsafe levels, there is nothing to force the relay to trip.

FIGURE 7. The observable inputs and outputs of the DRT


$drt = controller||plant$
F_1, I_1 F_2, I_2 F_3, I_3
CONTROLLER (microprocessors)
C (Activate Relay)
PLANT (reactor & relay)
R (Relay)
W (Power)
P (Pressure)

F_j, where $type(F_j) = \{fail, normal\}$, is a failure variable for $j = 1, 2, 3$. I_j is a timing cycle initialization variable for the j-th microprocessor (see Fig. 2).

The *plant* (Fig. 2) was described previously in Sect. 2.4 in the discussion of the StateTime toolset and in the description of the *relay* module (Fig. 3). The *output* object of the plant updates the pressure and power readings at most every two ticks of the clock. If the *endupdate* event is deleted with only the *update* object remaining, then pressure and power would be *forced* to change their values. With *endupdate* included, the sensor updates can be preempted thus leaving open the possibility that pressure or power (or both) remain unchanged for an additional two ticks[10].

The *output* object for power and pressure updates could have been included in the controller as it represents the *filtered* sensor readings not the generation of power and pressure in the plant itself which are continuously changing. Since the *output* object behaviour is fixed and known *a priori*, it is more convenient to include it with the plant.

In contrast to the plant, parts of the controller are initially unknown. It is known that there will be 3 microprocessors together with a majority voting circuit, i.e. the controller can be decomposed into sub-modules (Fig. 8) described by:

$$\begin{aligned} controller &= cont \,\|\, majorVote \\ cont &= micro_1 \,\|\, micro_2 \,\|\, micro_3 \end{aligned} \qquad \text{(Eq. 14)}$$

The microprocessors can either be in a normal or failed mode. The *j*-th microprocessor thus has an observable **out** variable F_j with $type(F_j) = \{fail, normal\}$ (Fig. 7). However, the precise nature of the *normal* behaviour is initially unknown, although the informal timing diagram (Fig. 6) does provide some guidance.

It is necessary to be able to tell when a microprocessor is at the initial point of a timing cycle where it checks for unsafe pressure and power levels (before invoking the two timers described in Fig. 6). Once a timing cycle is initiated in response to unsafe power or pressure levels, a new timing cycle cannot be initiated until the controller returns to its initial point. Hence, the *j*-th microprocessor also has an observable **out** variable I_j with $type(I_j) = \{0, 1\}$ where $(I_j = 0)$ means that the microprocessor is at its initial point. We require that a microprocessor timing cycle take no longer than the combination of the two timers which is 50 ticks with an additional two ticks to cover controller reaction times, i.e. $(I_j \neq 0) \Rightarrow \Diamond_{\leq 52}(I_j = 0)$.

We are now in a position to state the DRT requirements for 3-version control. The informal requirements [R1] and [R2] can be stated in temporal logic for any two functioning microprocessors i and j as:

R1: $[(I_i = 0) \wedge (I_j = 0) \wedge bothHi \wedge \Diamond_{30} powerHi] \Rightarrow \Diamond_{[30, 32]} \Box_{<20}(R = open)$ (Eq. 15)

R2: $[(I_i = 0) \wedge (I_j = 0) \wedge powerLo] \Rightarrow \Diamond_{\leq 2}(R = closed)$ (Eq. 16)

where the predicates *bothHi*, *powerHi* and *powerLo* are defined as:

$$\begin{aligned} bothHi &\overset{\text{def}}{=} (R = closed) \wedge \Box_{<2}(P = 1 \wedge W = 1) \\ powerHi &\overset{\text{def}}{=} \Box_{<2}(W = 1) \\ powerLo &\overset{\text{def}}{=} \Box_{<2}(W = 0) \end{aligned} \qquad \text{(Eq. 17)}$$

The controller can only react to changes in the pressure and power that persist long enough for the controller to be guaranteed to detect them (2 ticks of the clock). The controller microprocessors can sample pressure and power only once every tick of the clock. Hence, we require that the pressure and power both remain high for at least two ticks of the clock for the relay to open [R1]. Similar considerations apply when closing the relay [R2].

[10] The *pHi*, *pLo*,*wHi*, and *wLo* events could have been given bounds [2,∞] which would not force these events to occur. But then the pressure and power updates could drift apart. In the current model, $(D = wait) \rightarrow \Box_{<2}(P' = P \wedge W' = W)$ so that the sensor readings remain constant for a period sufficient to ensure that the microprocessor controllers can react to their inputs. We could have changed the definition of *bothHi* in (Eq. 17) to $(R = closed \wedge D = wait)$ except for the fact that D is a private variable (Fig. 2).

FIGURE 8. Architecture of the controller based on majority voting control

module controller

in W, P /* power and pressure variables from the plant */

out C : {0,1} **where** $(C = 0)$ /* relay activation variable based on majority vote. $(C = 1)$ means send a signal to the relay to request it to open */

out F_1, F_2, F_3 : {fail, normal} **where** $(F_j = normal)$ for $j = 1, 2, 3$ /* failure variables needed for specifying failed behaviour */

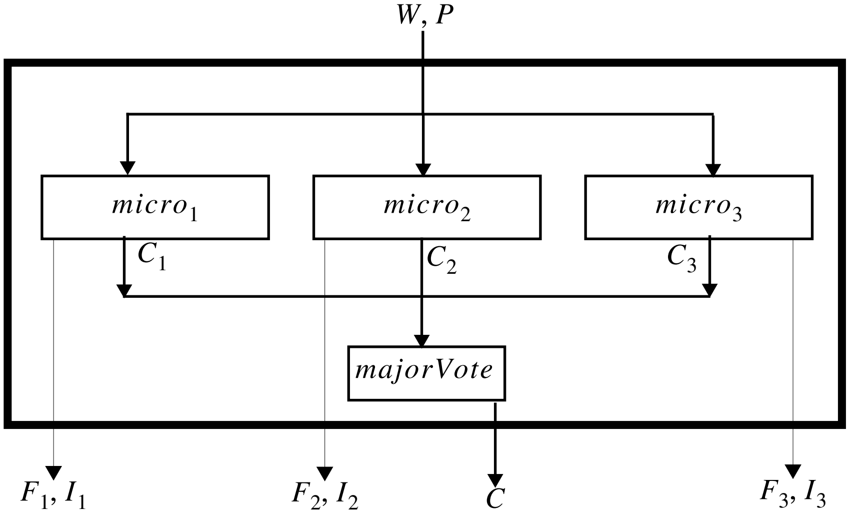
out I_1, I_2, I_3 : {0,1} **where** $(I_j = 0)$ for $j = 1, 2, 3$ /* $(I_j = 0)$ means the *j*-th microprocessor is at or has returned to the beginning of a timing cycle where it waits for unsafe power or pressure signals*/

Body

private C_1, C_2, C_3 **:** {0,1} **where** $(C_j = 0)$ for $j = 1, 2, 3$

/* The *j*-th microprocessor outputs a relay activation variable C_j as input to the majority voting circuit. The majority voter must decide, based on the microprocessor relay activation variables, whether to send an actual command to the relay of the plant via C. The interconnection diagram between the modules of the controller is shown below: */

$$controller = (micro_1 \parallel micro_2 \parallel micro_3) \parallel majorVote$$



Specification $sp(controller)$ **:**

$$\forall i, j{:}\{1, 2, 3\} \left[\begin{array}{l} (i \neq j) \wedge \Box(F_i \neq fail) \wedge \Box(F_j \neq fail) \rightarrow \\ \qquad [(I_i = 0) \wedge (I_j = 0) \wedge bothHi \wedge \Diamond_{30} powerHi \Rightarrow \Diamond_{[31, 32]} \Box_{<20}(C = 1)] \\ \qquad \wedge [(I_i = 0) \wedge (I_j = 0) \wedge powerLo \Rightarrow \Diamond_{\leq 1} \Box_{<2}(C = 0)] \\ \qquad \wedge [(I_i \neq 0) \Rightarrow \Diamond_{\leq 52}(I_i = 0)] \wedge [(I_j \neq 0) \Rightarrow \Diamond_{\leq 52}(I_j = 0)] \end{array} \right] \quad \text{(Eq. 12)}$$

The formulas *bothHi*, *powerHi* and *powerLo* are defined in (Eq. 17). The module specification (Eq. 12) is similar to the DRT requirement **R** but with the controller output variable C playing the same role for the controller specification that the relay variable plays in **R**. The specification is stated under the proviso that at least two of the microprocessors work normally, as the majority voting logic is only robust with respect to a single failure. The last conjunct of the consequent asserts that the controller cycle is at most 52 ticks of the clock, after which it is guaranteed to be back at its initial position (it is not 50 ticks as it may take up to two ticks to detect a change in the plant). Since the environment of the controller is the (fixed) plant, the controller specification can be weakened to:

$$des(plant) \rightarrow \text{(Eq. 12)} \qquad \text{(Eq. 13)}$$

end controller

The requirements as stated above do not take into account the possibility of microprocessor failures. **R1** and **R2** can only be required to hold if at least two of the microprocessors are functioning normally. The final requirement **R** is therefore:

$$\mathbf{R}\colon\ \forall i, j{:}\{1, 2, 3\}\,|\,(i \neq j){:}\ \Box(F_i \neq fail) \wedge \Box(F_j \neq fail) \rightarrow \mathbf{R1} \wedge \mathbf{R2} \qquad \text{(Eq. 18)}$$

where the integer variables i and j range over the three microprocessor controllers, i.e. $type(i) = type(j) \overset{\text{def}}{=} \{1, 2, 3\}$.

5.3 Problem to be solved

We must prove that the DRT conforms to its requirements. Formally, this means we must prove that $drt \models \mathbf{R}$ holds where $drt = plant \,\|\, controller$ and **R** is the formula given in (Eq. 18). Using the Composition Rule, a proof outline is:

1.	$sp(plant)$	modular-validity of (Eq. 5) in Fig. 2 for the *plant* by model-checking
2.	$sp(controller)$	modular-validity of (Eq. 12) in Fig. 8 for the *controller* by model-checking
3.	$sp(controller) \wedge sp(plant) \rightarrow \mathbf{R}$	general-validity (similar to the proof of (Eq. 8))
4.	$\mathbf{R}$	1, 2, 3 and the Composition Rule
5.	$drt \models \mathbf{R}$	$drt \overset{\text{def}}{=} controller \,\|\, plant$

The body of the plant module is given in Fig. 2. The only input variable to the plant is the relay activation variable C, which can be altered arbitrarily by the environment transition without generating too large a reachability graph. Hence step 1 in the above proof outline was verified using StateTime model-checking.

The only part of the above proof that cannot be verified is step 2, as the controller body is only partially defined at this point in the development. *Thus we must complete the design of the controller by designing its body, and demonstrate the modular validity of the controller specification.* Then the above proof outline guarantees that the DRT conforms to its requirements.

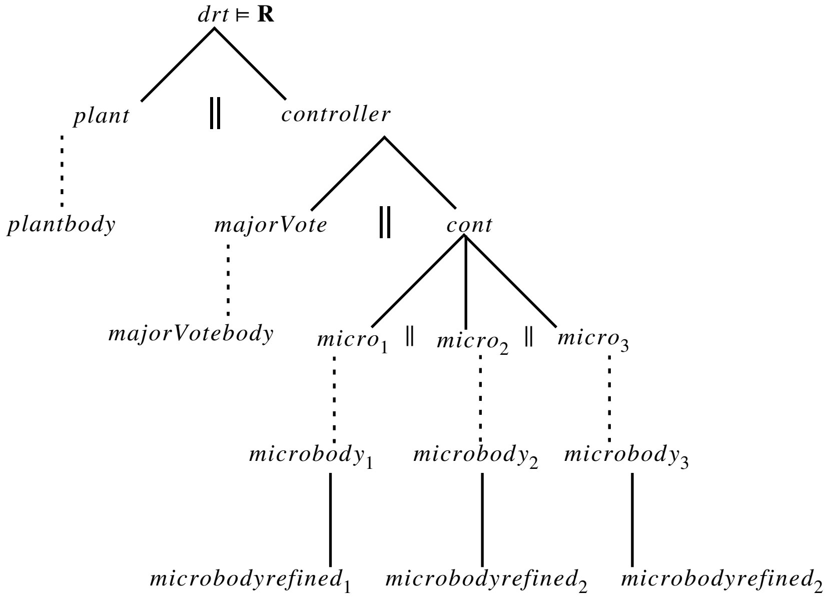
In checking the modular-validity of the controller specification (Eq. 12), it is sufficient replace step 2 above with the weaker specification (Eq. 13). Instead of using an unrestricted environment transition, (Th. 3) allows us to check sub-modules of the controller in the environment $plant$. The resultant reachability graphs of the sub-modules are much smaller than if an unrestricted environment transition is used. The above proof that the DRT conforms to its requirements then becomes:

1.	$sp(plant)$	modular-validity of the *plant* specification
2.	$des(plant) \rightarrow sp(controller)$	modular-validity of (Eq. 13) in Fig. 8 for the *controller*
3.	$des(plant)$	(Th. 1)(b)
4.	$sp(controller)$	2,3 and temporal logic
5.	$sp(controller) \wedge sp(plant) \rightarrow \mathbf{R}$	general-validity via deductive theorem proving
6.	$drt \models \mathbf{R}$	1, 4, 5 and the Composition Rule

The design of the DRT controller will be performed using the structured compositional approach described by the structure diagram (Fig. 1) as outlined in the introduction. The structure diagram for the DRT is given in Fig. 9.

FIGURE 9. Structure diagram for the DRT

See Fig. 1 in the introduction for the interpretation of the structure diagram



5.4 Controller design

A partial description of the controller was provided in Fig. 8. The *majorityVote* sub-module of the controller was described in Sect. 3.3 (Fig. 4). We must now design the microprocessor sub-modules. The body of the module $micro_1$ is shown in Fig. 10, with the other two microprocessors having symmetric descriptions.

The *normal* object of the controller (Fig. 10) is a more thorough description of the informal timing diagram of the analog controller (Fig. 6). The lower and upper time bounds of 1 in the transitions of *normal* indicate that the microprocessor samples the sensor inputs and passes through a block of control code every tick of the clock (0.1 seconds). Once unsafe power and pressure levels are detected by the transition *mu*, the *normal* object waits in activity *n1* for 29 clock ticks (2.9 seconds) before proceeding to activity *n2*. If the power is still high then the relay activity variable C_1 is set via transition *alpha*, else the system resets via transition *rho1*. The second timer Timer2 of the analog controller is described by the *delay20* transition. The *beta* transition resets the control activation variable when power returns to normal levels.

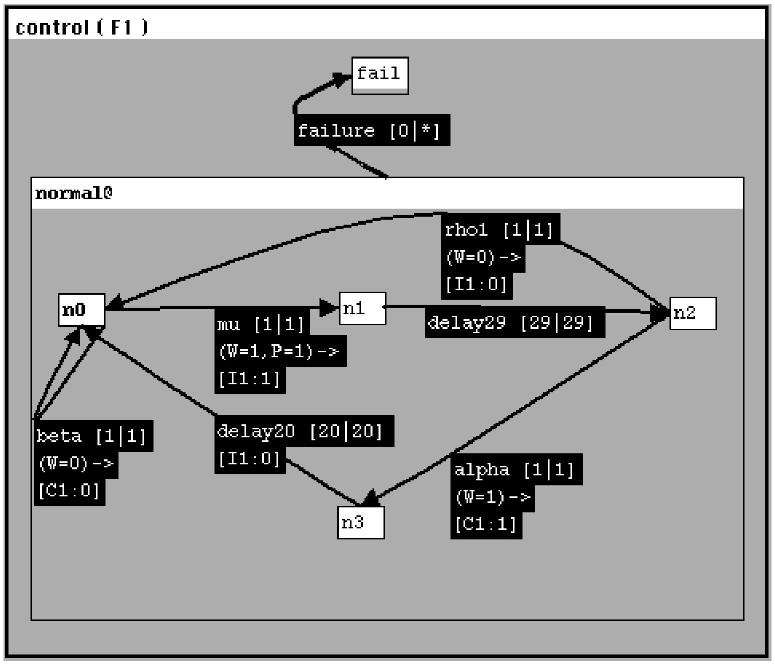
It is obvious from the foregoing that TTMs can provide precise convenient descriptions of timing information. The *normal* object can be seen as a high level specification of the microprocessor. The microprocessors do not have delay and time-out constructs; rather, timing variables must be incremented every pass through the block of code to keep track of the passage of time. In Sect. 5.5, *normal* will be refined closer to code that can be implemented on the microprocessors.

Once the body of the microprocessor module is known, the modular-validity of $sp(micro_1)$ in Fig. 10 can then be verified via StateTime model-checking. As explained at the end of Sect. 5.3, the controller will be used in the constrained environment of the *plant*. Hence we need not consider an environment transition that can arbitrarily modify power and pressure. The *output* object of the plant (Fig. 2) allows updates of power and pressure

FIGURE 10. Control module $micro_1$

module micro1
in *W,P* /* power and pressure from the plant */
out F_1 : {0,1} **where** $F_1 = fail$ /* the fail variable */
out C_1 : {0,1} **where** $(C_1 = 0)$ /* relay activation variable*/
out I_1 : {0,1} **where** $(I_1 = 0)$ /* initial condition variable for start of timing cycle */

Body
private N_1 **where** $N_1 = n0$ /* object variable of *normal* */



Specification:

$$sp(micro_1):\quad \Box(F_1 \neq fail) \to (r_1 \wedge s_1 \wedge t_1)$$

$$\text{where: } \begin{bmatrix} r_1 \overset{\text{def}}{=} [(I_1 = 0) \wedge bothHi \wedge \Diamond_{30} powerHi] \Rightarrow \Diamond_{31}\Box_{<20}(C_1 = 1) \\ s_1 \overset{\text{def}}{=} [I_1 = 0 \wedge powerLo] \Rightarrow \Diamond_{\leq 1}\Box_{<2}(C_1 = 0) \\ t_1 \overset{\text{def}}{=} [(I_1 \neq 0) \Rightarrow \Diamond_{\leq 52}(I_1 = 0)] \end{bmatrix}$$

end module

at most once every two ticks of the clock; this constrained environment will produce a smaller reachability graph. Hence, instead of showing the modular-validity of $sp(micro_1)$ (Eq. 12), we can verify the weaker validity (Eq. 13) given by

$$micro_1 \vDash des(output) \to sp(micro_1)$$

by model checking $(output \parallel micro_1) \vDash sp(micro_1)$.

Since the microprocessor and majority vote modules satisfy their module specifications, we can now show that $sp(controller)$ is modularly-valid. Let i, j be integer variables that range over the three microprocessors ($type(i) = type(j) \overset{\text{def}}{=} \{1, 2, 3\}$). Then

1. $(i \neq j) \wedge \Box(F_i \neq fail) \wedge \Box(F_j \neq fail)$ **Assume**
2. $sp(micro_i)$ modular validity $micro_i$

3. $sp(micro_j)$ modular validity of $micro_j$

4. $sp(micro_i) \wedge sp(micro_j)$ 2,3 and the Composition Rule

5. $r_i \wedge r_j$ 1, 4 and temporal logic (see Fig. 10 for the *micro* specifications r_i, r_j)

6. $(r_i \wedge r_j) \rightarrow [(I_i = 0 \wedge I_j = 0 \wedge bothHi \wedge \Diamond_{30} powerHi) \Rightarrow \Diamond_{31} \Box_{<20}(C_i = 1 \wedge C_j = 1)]$ general-validity

7. $[(i, j{:}\{1, 2, 3\}) \wedge (i \neq j) \wedge (C_i = 1 \wedge C_j = 1)] \Rightarrow (C_1 + C_2 + C_3 \geq 2)$ integer reasoning

8. $(I_i = 0 \wedge I_j = 0 \wedge bothHi \wedge \Diamond_{30} powerHi) \Rightarrow \Diamond_{31} \Box_{<20}(C_1 + C_2 + C_3 \geq 2)$ 1,5,6,7 and temporal logic

9. $[\Box_{<20}(C_1 + C_2 + C_3 \geq 2) \Rightarrow \Diamond_{\leq 1} \Box_{<20}(C = 1)]$ modular-validity of *majorityVote* module

10. $(I_i = 0 \wedge I_j = 0 \wedge bothHi \wedge \Diamond_{30} powerHi) \Rightarrow \Diamond_{[31,\, 32]} \Box_{<20}(C = 1)$ 8,9 and the Composition Rule

Line (10) of the above proof produces the first conjunct in the consequent of the controller specification (Eq. 12). The other conjuncts are obtained by similar (and much simpler) reasoning. We thus have:

$$11. \quad \left[\begin{array}{l} (i \neq j) \wedge \Box(F_i \neq fail) \wedge \Box(F_j \neq fail) \rightarrow \\ \qquad [(I_i = 0) \wedge (I_j = 0) \wedge bothHi \wedge \Diamond_{30} powerHi \Rightarrow \Diamond_{[31,\, 32]} \Box_{<20}(C = 1)] \\ \quad \wedge [(I_i = 0) \wedge (I_j = 0) \wedge powerLo \Rightarrow \Diamond_{\leq 1} \Box_{<2}(C = 0)] \\ \quad \wedge [(I_j \neq 0) \Rightarrow \Diamond_{\leq 50}(I_j = 0)] \wedge [(I_j \neq 0) \Rightarrow \Diamond_{\leq 50}(I_j = 0)] \end{array}\right] \qquad \textbf{discharging 1.}$$

12. $controller \vDash des(plant) \rightarrow sp(controller)$ *i* and *j* were arbitrary; a constrained environment was used

As shown in Sect. 5.3, the above result implies that the DRT conforms to its requirements. The proof of conformance used a combination of model checking (for verifying modular-validity) and deduction (e.g. for proving the general validity in step 6).

5.5 Refining the controller

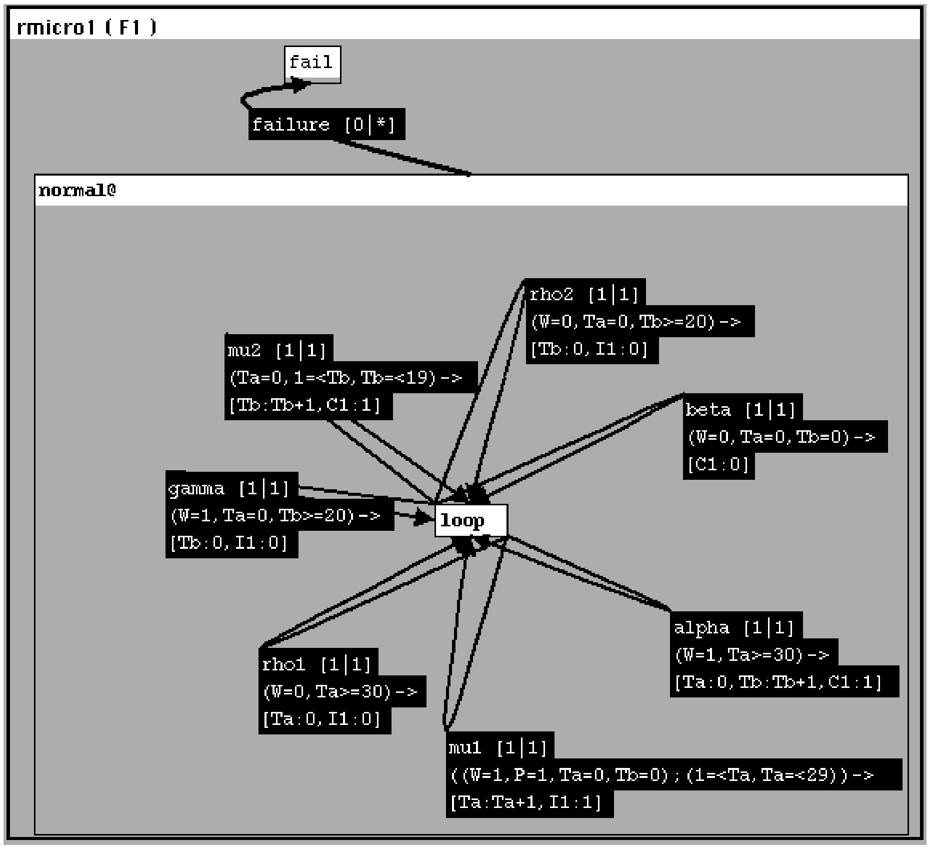
The abstract module $micro_1$ (Fig. 10) is observationally equivalent to its refinement $rmicro_1$ (Fig. 11), i.e. $rmicro_1 \approx micro_1$. The refinement $rmicro_1$ is closer to the final pseudocode [38]. As mentioned in Sect. 4.0, two methods have been developed for showing observational equivalence:

- The designer can interactively apply equivalence preserving transformations to derive $rmicro_1$ from $micro_1$. The reader may consult [25] where this transformation is done for a TTM body the same as that of $micro_1$ but without the additional failure transition and the initial condition variable I_1. The proof used in [25] can be used as is for $rmicro_1 \approx micro_1$. The transformation rules can be applied to infinite state systems, but it can be shown that there is no complete set of transformations, i.e. there is no finite set of transformations such that it is always possible to prove TTM equivalence by using that set of transformations [27].
- For TTMs that can be reduced to finite state reachability graphs, there is an efficient polynomial time algorithm for showing observational equivalence [28]. The equivalence of $rmicro_1$ and $micro_1$ can be shown with this algorithm as the data types are finite.

The abstract module $micro_1$ satisfies the non-Zeno condition (Table 1). Since $sp(micro_1)$ is SESI (state event stuttering invariant) over the interface variables, (Th. 4) guarantees that $sp(micro_1)$ also holds for the refinement $rmicro_1$. Thus there is no need to

FIGURE 11. Refinement of microprocessor control module

/* Body of $rmicro_1$ with same interface stub and module specification as $micro_1$ (Fig. 10) •/
private Ta: {0 ... 30} **where** (Ta = 0)   /* Timer1 variable in timing diagram */
private Tb: {0 ... 20} **where** (Tb = 0)   /* Timer2 variable in timing diagram */



redo the proofs of controller module specifications, and we remain with the guarantee that the DRT conforms to its requirements.

The module $micro_1$ is a high level description of a microprocessor controller. It is easier to understand than $rmicro_1$ because it is close to the informal timing diagram of the analog controller (Fig. 6). It does not have the two timer variables that $rmicro_1$ has, and as a result the guards on its transitions are simplified relative to those of $rmicro_1$. Its reachability graph is smaller (Table 1).

TABLE 1. Improved model checking times for the module $micro_1$ compared to $rmicro_1$ [a]

Modularly valid specifications	Abstraction $micro_1$	Refinement $rmicro_1$
$\Box(F_1 \neq fail) \to r_1$ (Fig. 10)	13785 states in 26 seconds	59452 states in 297 seconds
$\Box\Diamond(\varepsilon = tick)$ (non-Zeno constraint)	15248 states in 61 seconds	69059 states in 261 seconds

[a] Above checks used the StateTime toolset and STeP on an Sparc Ultra1 with 160MB RAM.

Table 1 shows the result for checking the most complex module. However, all the module specifications were verified using the model-checker. The deductive parts of the proof were done by hand. In principal the deductive part could have been done using the theorem prover, but it proved too tedious as explained at the end of Sect. 2.4.

We refer the reader to [38] for a discussion of the reverse engineering problem, i.e. how one goes from the pseudocode described in the original requirements document to the refinement presented in Fig. 11.

5.6 The design method

Although top-down design by stepwise refinement was *de rigueur* until the 1980's, it has subsequently come under attack. As Jackson has written [20]: "It was one thing to impose a single hierarchical structure on a *sequential* program of the programmer's own devising; it was quite another to impose it on a given, inconveniently ill-structured, real world domain". In fact, real-systems such as the DRT often have no single "top" function.

Our design method uses both top-down as well as bottom-up techniques. We have stressed in previous sections that the Composition and Refinement Rules can be used both ways. Our top-down methodology differs from the classic notion of stepwise refinement. In classic top-down design, a program is a single sequential process; concurrency and parallelism was "exotic" or unknown [20]. By contrast, TTM modules allows for nondeterminism, and serial as well as parallel constructs in any mixture and to any depth. This allows for adequate descriptions of real systems that have no "top" in the functional sense. Furthermore, at the top level, we do have requirements describing the safety and correctness of the overall system consisting of different parts (such as the plant and the controller). Such system requirements (e.g. the DRT requirements R1 and R2) are often *emergent* properties, i.e. they arise out of the combined interaction of the system modules taken together. There is thus still an urgent need to describe systems in a layered modular fashion, but without the sequential restrictions of the earlier methods.

We now describe in outline the basic design method. The notions of a module, composition and refinement developed in this paper, provide the precise theoretical underpinnings for the method which was originally sketched in [36, pages 4-6]. We also borrow concepts from the insightful description of requirements in [20, p169].

The basic design procedure starts with *requirements R*. Requirements are about the phenomena of the application domain (the relay, pressure and power of the DRT *plant*), not about the machine (the *controller*). Our first step in requirements is to divide the system into the two parallel objects: (a) the plant which can be described as it already exists and (b) the controller which must be designed. This division proceeds by describing their relevant interfaces and connections, as well as some of the internal phenomena and entities of the plant — this is the body of the plant which is a model of plant behaviour. The plant model cannot be too abstract because then it is not about the real problem anymore. It is a mistake to rush to the solution (by coding the controller) before delineating the problem to be solved (the plant requirements). The requirements are temporal logic formulas in *plant* entities such as pressure, power and the state of the relay. Therefore, the requirements do not describe the internal phenomena of the controller, although they might describe entities at the boundary of the controller and the plant (the *shared phenomena*).

It is the job of the controller to ensure that the requirements are satisfied, which it can do due to fact that it shares *some* phenomena with the plant (as described by the plant-controller interface). The controller might not be able to react to a shared phenomenon immediately (e.g. a change in reactor pressure), but the shared phenomenon happens in both the plant and controller simultaneously.

Because the controller does not always know all the plant phenomena (or at least cannot react to them immediately), there is always the possibility of a gap between the requirements and what the controller can achieve (as described in the controller specification). The progression from requirements to controller implementation is a way of bridging the gap between them. From the requirements expressed in terms of the plant, you derive a *specification S* of the controller in terms of the shared phenomena of the plant and controller. Then you derive the body of the controller from the controller specification. The Composition Rule justifies the eventual claim that the controller implementation satisfies the requirements by reasoning as follows: (a) the body of the controller satisfies the specification *S* and (b) the specification *S* together with the description of the plant entails the truth of requirements *R*.

In the case of the DRT controller, once the top-level interface stub was described, the parts of the controller were developed bottom-up component by component. A generic microprocessor controller was designed which was then instantiated three times to obtain 3-version control. Then the majority voting logic was designed. Bottom level modules were developed, simulated and verified to conform to their local specifications long before the modules were combined together. The plant description was quite simple and could be encapsulated in a single module. In more complicated application domains, the plant might also benefit from a bottom-up development.

6.0 Conclusions and related work

This paper has presented a structured compositional method for the deliberate design of real-time systems, and applied the method to an industrial example with partial support provided by the StateTime toolset. The main novelty of the approach is to provide a fully compositional definition of real-time reactive modules compatible with existing model-checking tools (Sect. 3.0) and a refinement relation (Sect. 4.0). This allows for the systematic development and verification of real-time systems.

There are four main areas where mechanical support is needed: (1) system simulation for validating models, (2) model-checking for modular-validity, (2) deductive theorem proving for the composition rule, and (3) proving observational equivalence for the refinement rule.

StateTime was used for *simulation* and *model-checking* all module specifications of the DRT example. Although, in principal, we could have used the toolset for the deductive part, it proved too unwieldy due to the proliferation of quantifiers. The toolset has no support for refinement, and this had to be done by hand using behaviour preserving transformations. So too, there is a need to directly supports modules with interface stubs and automatic generation of environments.

This leads us to the main conclusion of this paper. StateTime, as it currently stands, needs to be enhanced in a variety of ways if it is to support more seamless compositional verification. I am therefore currently working, together with Lewis Lo, on a new version of StateTime that builds on the reactive modules introduced in this paper, and allows for the complete spectrum of automated simulation, model-checking and theorem proving tools.

The enhanced StateTime will also support modules (interface stubs and automatic generation of environments). We use count-up and count-down clock variables with ordinary temporal logic (rather than the bounded operators of RTTL) for specification, but it is yet to early to tell to what extent this will simplify deductive reasoning.

The proof of observation equivalence (both algorithmically for finite state modules and via equivalence preserving transformations) for use in the refinement rule needs to be implemented and incorporated into the StateTime toolset, but we have not yet decided how to implement these techniques. Because our bisimulation relation involves both states and events (Sect. 4.0), we may not be able to directly use existing tools such as Concurrency Workbench [9]. The Concurrency Workbench allows for the testing of equivalences and preorders and the verification of systems in the modal mu-calculus; real-time CCS style front-ends are available.

Other tools such as Modechart [21], Statemate [16] and ObjectTime [45] also use statecharts for visual system descriptions. Modechart allows for both simulation and algorithmic analysis techniques for a subset of properties expressed in a predicate real-time logic [34]. Statemate can be used to do reachability analysis and ObjectTime is object-oriented which is useful in design, but it cannot deal with hard real-time systems. None of these tools have theorem provers, nor do they allow for modular verification.

RTTL is based on the linear time temporal logic LTL rather than on branching time logics such as CTL. It is commonly accepted that while specifying is easier in LTL, model-checking is more efficient in CTL. Both linear and branching time languages now have efficient model-checkers using either partial orders or BDD methods: SPIN [18] is one of the few LTL based model-checkers. SMV is a good example of a CTL based model-checker [5], with an extension to real-time systems called Verus which uses a branching time real-time language called RTCTL [6]. Verus also has the facility for specifying task priorities. The hybrid tool HyTech [2] extends branching time model-checking to continuous real-time systems using stop watches and symbolic fixpoint computation (the current version of the tool supports reachability analysis via monitor automatons and not directly the full set of CTL formulas). HyTech and Verus both allow for parametric analysis (e.g. determining the latest possible moment a controller can wait before issuing a command).

The STeP [31] model-checker and theorem prover was chosen as the back-end to StateTime rather than tools such as SPIN, SMV and HyTech for a number of reasons. Tools that use a non-interleaving synchronous execution step algorithm (e.g. SMV, the PVS model-checker [42] and COSPAN [15]) are efficient for dealing with hardware designs, but do not seem to be as efficient as SPIN when it comes to dealing with interleaved sequential code and integer variables. There is also another problem associated with modularity when it comes to branching time model-checkers. Although branching time is usually more efficient than linear time logics, the branching time algorithms become EXPTIME-complete for *module checking* which is worse than the PSPACE complexity of linear time logics [24]. This analysis seems to suggest that the accepted trade-off between LTL and CTL for *modules* is not as simple as it is for *closed* systems. We were not able to use SPIN because it only supports justice (weak fairness) not compassion (strong fairness) needed for the *tick* transition. More importantly, we hope to use the theorem proving components of STeP in future versions of our tool. None of the aforementioned tools (except PVS) have theorem provers.

Hooman [19] extends Hoare logic to real-time programs by freely mixing programs and assumption/guarantee assertions leading to a top-down derivation method. The theory is implemented using the interactive proof checker PVS [42]. The embedding of the proof system in PVS provides powerful mechanical support for compositional reasoning (but not model checking). The only timing construct is the precise delay; there are no time bounds on transitions as in TTMs.

There is a growing interest in compositional and refinement methods for reactive systems [1,7,22,35,41,46,48]. The field is somewhat less developed in the case of real-time systems especially in methods that also have tool support.

ASTRAL [10] is based on the framework of [11] that uses Petri Nets for system descriptions and a timed temporal logic called TRIO for specifications. ASTRAL provides structuring mechanisms that allow the designer to build modularized specifications that are translated into TRIO. Proofs in ASTRAL are either *interlevel* or *intralevel*. The former deals with proving that the specification at a higher level is consistent with a specification at a lower level. The latter deals with proving that a description at a level satisfies its specification. A tool is currently under development.

The frameworks mentioned thus far have specification languages that are based on logic, usually modal logic. Other approaches are based on algebra or automata. Discrete real-time process algebras [4,44] can describe systems compositionally at different levels of abstraction. The semantics of process algebras is usually defined in terms of labelled transition systems. An algorithm based on observation (bisimulation) equivalence is used to show that an implementation satisfies its specification. These bisimulation relations are usually event-based [33], whereas the bisimulation relation used in this paper is both event and state-based (Sect. 4.1). It is event-based in order to ensure a global notion of time via the *tick* transition. It is state-based so that module specifications can be written as temporal logic properties in the observable variables. Continuous time extensions to process algebras [47] lack the abstracting power of a congruence relation of the discrete event case, due to technical difficulties associated with their infinite branching continuous time semantics.

The real-time CSR language [13] provides a layered approach to dealing with shared resources. [12] presents hierarchical multistate machines for multilevel specifications. The automata based tool COSPAN has recently been extended to deal with real-time [3]. COSPAN supports top-down development through successive refinements and homomorphic reduction [15]. Timed automata [30] (see also the input/output automata described in [29]) have visible actions, a time passage action (analogous to our clock tick) and a special internal action. Dense upper bounds can be imposed between actions, but not lower time bounds. A refinement from one timed automaton to another is a time-preserving function similar to the classical notion of a homomorphism between automata.

In single language frameworks (e.g. automata based COSPAN or the logic based TLA [1]), both the implementation and specification are expressed in the same formalism (automata or logic). Conformance is proved by demonstrating that each fair trace of the implementation is also a fair trace of the specification. There is a certain elegance and simplicity associated with using a single language for both specifications and implementations. We have pursued the dual TTM/RTTL framework in this paper as it provides us with the flexibility of using the most appropriate analysis technique in each case. For TTM refinement, we use the algebraic notion of observation equivalence, and for TTM composition the logical conjunction of RTTL specifications.

Acknowledgments

I would like to thank Mark Lawford for his help with all aspects of this paper. I also thank the anonymous referees for their helpful criticisms, comments and suggestions.

7.0 References

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

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

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

[4] Baeten, J.C.M. and J.A. Bergstra. “Discrete Time Process Algebra.” *Formal Aspects of Computing*, 8(2): 188-208, 1996.

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

[6] Campos, S.V. and E.M. Clark. “ Real-Time Symbolic Model Checking for Discrete Time Models.” In *Theories and Experiences for Real-Time System Development*, ed. T. Rus and C. Rattray. AMAST Series in Computing, Vol. 2. World Scientific Press, 1994.

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

[8] Chang, E. “Compositional Verification of Reactive and Real-Time Systems.” Ph.D, Stanford University, 1995.

[9] Cleaveland, R. and S. Sims. “The NCSU Concurrency Workbench.” In *Computer-Aided* Verification (CAV '96), New Brunswick, NJ, edited by R. Alur and T. Henzinger, Springer-Verlag, LNCS 1102, 394-397, 1996.

[10] Coen-Porisini, A., R. Kemmerer, and D. Mandrioli. “A formal framework for ASTRAL intralevel proof obligations.” *IEEE Transactions on Software Engineering*, 20(8): 548-560, 1994.

[11] Felder, M., D. Mandrioli, and A. Morzenti. “Proving properties of real-time systems through logical specifications and Petri Net models.” *IEEE Transactions on Software Engineering*, 20(2): 127-141, 1994.

[12] Gabrielian, A. and M. Franklin. “Multilevel specifications of real-time systems.” *Communications of the ACM*, 34(5): 51-60, 1991.

[13] Gerber, R. and I. Lee. “A layered approach to automating the verification of real-time systems.” *IEEE Transactions on Software Engineering*, 18(9): 768-784, 1992.

[14] Gries, D. and F.B. Schneider. *A Logical Approach to Discrete Math*. Springer Verlag, 1993.

[15] Hardin, R.H., Z. Harel, and R.P. Kurshan. “COSPAN.” In *8th International Conference on Computer Aided Verification CAV'96*, LNCS 1102 Springer-Verlag, 421-427, 1996.

[16] 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.

[17] Harel, D. and A. Pnueli. “On the Development of Reactive Systems.” In *Logics and Models of Concurrent Systems*, ed. K. Apt. 477-498. F-13 of NATO Advanced Summer Institutes. Springer-Verlag, 1985.

[18] Holzmann, G. “The Model Checker Spin.” *IEEE Trans. on Software Engineering*, 23(5): 279-295, 1997.

[19] 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.

[20] Jackson, M. *Software Requirements & Specifications*. Addison-Wesley, 1995.

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

[22] Jones, C.B. “Specification and design of parallel programs.” In *IFIP 9th World Congress*, 321-323, 1983.

[23] Kesten, Y., Z. Manna, and A. Pnueli. “Verifying Clocked Transition Systems.” In *Hybrid Systems III*, Springer-Verlag, LNCS, 1996.

[24] Kupferman, O. and M.Y. Vardi. “Module Checking.” In *8th International Conference on Computer Aided Verification CAV'96*, LNCS 1102 Springer-Verlag, 75-86, 1996.

[25] 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.

[26] Lawford, M., J.S. Ostroff, and W.M. Wonham. “Model Reduction of Modules for State-Event Temporal Logics.” In *IFIP Joint International Conference on Formal Description Techniques (FORTE-PSTV'96)*, Chapman & Hall, 1996.

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

[28] 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.

[29] Lynch, N. and R. Segala. “A Comparison of Simulation Techniques and Algebraic Techniques for Verifying Concurrent Systems.” *Formal Aspects of Computing*, 7(3): 231-265, 1995.

[30] Lynch, N. and F. Vaandrager. “Forward and Backward Simulations for Timing-Based Systems.” In *REX Workshop — Real-Time: Theory in Practice*, 397-446. LNCS 600 Springer-Verlag, 1992.

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

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

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

[34] Mok, A. and D. Stuart. “Simulation vs. Verification: Getting the Best of Both Worlds.” In *11th Annual IEEE Conference on Computer Assurance (COMPASS)*, Washington D.C, 1995.

[35] Mokkedem, A. and D. Mery. “On Using Temporal Logic for Refinement and Compositional Verification of Concurrent Systems.” *Theoretical Computer Science*, 140:95-138, 1995.

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

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

[38] Ostroff, J.S. “A Visual Toolset for the Design of Real-Time Discrete Event Systems.” *IEEE Trans. on Control Systems Technology*, 5(3): 320-337, 1997.

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

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

[41] Owicki, S. and D. Gries. “Verifying properties of parallel programs: an axiomatic approach.” *Communications of the ACM*, 19(5): 279-285, 1976.

[42] Owre, S., J. Rushby, N. Shankar, 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.

[43] 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.

[44] Schneider, S., J. Davies, D.M. Jackson, G.M. Reed, J.N. Reed, and A.W. Roscoe. “Timed CSP: Theory and Practice.” In *REX Workshop --- Real-Time: Theory in Practice*, 640-675. LNCS 600, Springer-Verlag, 1992.

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

[46] Stolen, K., F. Dederichs, and R. Weber. “Specification and refinmenent of networks of asynchronously communicating agents using the assumption/commitment paradigm.” *Formal Aspects of Computing*, 8(2): 127-161, 1996.

[47] Yi, W. “CCS + Time = an Interleaving Model for Real Time Systems.” In *Proceedings of ICALP'91*, 217-228. LNCS 510 Springer-Verlag, 1991.

[48] Zwiers, J. and W.P.d. Roever. “Compositionality and modularity in process specification and design.” In *Temporal logic in specification*, ed. B. Banieqbal, H. Barringer, and A. Pnueli. 351-374. LNCS 398 Springer-Verlag, 1989.