Specifying and Verifying Real-Time Reactive Systems in TTM/RTTL

Jonathan S. Ostroff
Department Of Computer Science, York University[1],
4700 Keele Street, North York Ontario, Canada, M3J 1P3.
Email: jonathan@cs.yorku.ca Tel: 416-736-5053 Fax: 416-736-5872
Technical Report Number: CS-ETR-94-08
from: ftp.cs.yorku.ca:/pub/TECH-REPORTS

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

The original version of RTTL was based on the floating version of temporal logic for fair transition systems. In this paper, we recast RTTL in the anchored framework of Manna and Pnueli [21], which is simpler and more concise than the floating version. To this we add the real-time semantics and proof rules for dealing with hard time systems. A decomposition theorem for modular reasoning is provided. Hierarchical modular development of systems using the heuristics and the toolset is illustrated with examples, including a mutual exclusion algorithm based on time bounds and real-time resource allocation. Timing constraints need not be known a priori. The heuristics are used to derive the timing constraints that will guarantee various properties such as mutual exclusion and real-time response. The development paradigms proposed in this paper including compositional reasoning, heuristics for verifying modules or decision procedures, and the support of a graphical toolset provides a feasible approach to the systematic development of real-time reactive systems.

Key Words: Real-Time Temporal Logic, Timed Transition Systems, Concurrent Programs, Specification, Verification, Proof System, CASE tools, Modules, Modular Reasoning.

1. This work is supported in part by the National Science and Engineering Research Council of Canada.

Table of Contents

1.0 Introduction

Software that controls safety critical systems can endanger the environment and life itself. In a recent case for example, errors in the code of computerized radiation therapy machines resulted in serious injuries and deaths [19]. Coding errors are one source of concern, but omissions in the software requirements or mishandled environmental conditions may also contribute to dangerous behaviour of critical equipment (e.g. nuclear reactors, robots, planes, and traffic systems). Extensive testing and software quality assurance is obviously crucial if systems are to be deployed safely.

One of the best methods we know of for software quality assurance, is the use of formal methods for the specification and verification at the module and systems level. *Real-time reactive modules* are the crucial elements in safety critical, embedded and concurrent systems. Transformational programs produce a final result on termination. In contrast, *reactive* programs maintain ongoing non-terminating interaction with their environments. Embedded software must interact with an environment of machinery, sensors and actuators that are not completely under the programmer's control. The difficulty of specifying such systems is compounded when *hard real-time* constraints on the system behaviour must be satisfied.

Logical formalisms and techniques have been invaluable for proving properties of complex computing systems. The idea of representing concurrent programs and their specifications as formulas in a temporal logic was first proposed by Pnueli [33]. However, untimed temporal logic admits only the treatment of qualitative timing properties such as the demand that an event occur "eventually". Hard real-time systems depend crucially on the actual times at which events occur.

Semantic extensions to temporal logic for hard real-time systems were originally proposed in [4,17]. The TTM/RTTL framework was introduced a few years later, and included a proof methodology and the first decision procedures for fragments of the real-time logic [22,23,32]. Real-time extensions to Petri nets, process algebras and various real-time logics have been proposed over the last decade. A detailed discussion and comparison of other real-time formalisms and TTM/RTTL is presented in [26].

The TTM/RTTL framework provides a formal methodology for the specification, development and verification of real-time reactive systems. This methodology consists of several elements listed below:

- A *generic computational model* called timed transition systems (TTMs) provides a uniform representation for various real-time programming languages (e.g. Ada, Occam and timed Petri nets) and diverse mechanisms for timing, synchronization and communication constructs [21,22]. A TTM is a guarded transition system with lower and upper bounds on the transitions that relate to the occurrence of a special transition *tick*. The computational model assigns a *semantics* to each real-time reactive system represented as a TTM. The semantics associates with each TTM a *behaviour*, i.e. a set of computations of the system called *legal trajectories*. A legal trajectory is an infinite sequence of states

and events that represents a single execution of the program or system. An interleaving representation of concurrency is used, which is adequate for representing true concurrency if the system is modelled at a sufficiently fine grain of atomicity, and with the appropriate real-time and fairness requirements on legal trajectories.

- An *abstract specification language* called real-time temporal logic (RTTL), which is a timed extension of temporal logic, augmented with some program specific predicates and verification conditions needed to describe the state in a trajectory of a reactive program. The legal trajectories of TTMs serve as *models* (in the logical sense) for the formulas of RTTL. The linear semantics of RTTL allows us to associate with a formula a set of trajectories that are models of the formula. A program *satisfies* an RTTL formula if all the legal trajectories of the program are models of p. A TTM and an associated RTTL formula are thus two ways of looking at a reactive system; in each case they are associated with a set of legal trajectories that characterize the behaviour of the program. TTMs constitute our link to low-level "real world" procedural implementations, while an RTTL formula is a declarative higher-level abstract specification of the system behaviour. This is in contrast to single language frameworks such as the temporal logic of actions [1] in which both low level programs and their high-level requirements are specified in the same temporal language.
- A *proof system* which allows us to prove that a program (i.e. a TTM) satisfies a given RTTL specification. The proof system is partitioned into two parts. The state reasoning part is for proving validities of *state-formulas* (predicates in the program variables with no temporal operators occurring in them that can be evaluated in a single state). State-formulas make assertions about relationships between the program variables which range over various domains such as integers, rationals, arrays, sets and lists. The temporal logic part of the proof system is itself divided into parts. A general part allows for the derivation of generally valid theorems (satisfied in any program). A program part allows for the derivation of those properties that are specific to a particular program (but are not necessarily generally valid). The separation between state reasoning and temporal logic reasoning is advantageous. Most of the work in proving important properties involves mainly state reasoning with very little actual temporal logic needed. Hence, off the shelf theorem provers can go a long way towards mechanizing proofs.
- *Decision procedures* and *visual tools* for automated specification and verification take the burden off the human verifier and allow for larger systems to be treated than can be hand verified. Tools that have graphical specification methods [12,16,35] have been developed because engineers like to work with pictures. StateTime [27] is a prototype toolset for the TTM/RTTL framework. The BUILD tool allows the designer to enter systems as TTMcharts (similar to statecharts). Any partial or complete model is immediately executable, which allows for rapid prototyping and validation. The VERIFY tool is used to model-check finite state TTMs. The DEVELOP tool is used for verifying infinite state systems using the proof system. The DEVELOP tool uses constraint logic pro-

gramming for state reasoning, and proof diagram heuristics for temporal reasoning. StateTime was used for checking part of the shutdown procedure for the Candu reactor [30]. The tool automatically discovered that the pseudocode for the trip delay did not satisfy its requirements.

Model-checking was first introduced by the authors of [8], and extended to real-time systems in [22,23], which is the basis of the VERIFY tool. Time cannot just be modelled by a concurrent process that continuously increments some time variable, for then the reachability graph would be infinite state. A more sophisticated approach must be used to keep the reachability graph finite state, but this results in an additional complexity over untimed systems that depends on the product of the upper time bounds of the timed transitions or clocks [2,5].

Techniques for efficient model-checking for real-time temporal logics have steadily improved since the earlier work, especially with the use of binary decision diagrams and efficient state space exploration [3,6,13,34]. Not all of these techniques are suitable for the general systems that VERIFY must deal with. For example, binary decision diagrams do not always deal with data variables efficiently. The intention is to incorporate some of these techniques into the VERIFY tool.

A proper method for constructing real-time reactive systems cannot wait until a full implementation is constructed followed by *a posteriori* hand verification of the complete system. Real systems are just too big and complex for such an approach. A more constructive approach is needed for gradually transforming a specification into a correct implementation. Two major transformations of importance are *decomposition* and *refinement*.

In refinement, we replace a moderately sized action that is atomic at one level of abstraction, by a more detailed implementation that necessarily makes the original action non-atomic at this lower level. At the module level, we would like to replace a high-level module M_h by M_l (with lower level constructs that are directly available on the considered machine). To do this, we define the *observable subset* of the module variables (usually the external input and output variables), and consider the corresponding reduced behaviour projected onto the observable set, taking due care that the modules agree on when to take clock ticks. M_l refines (or implements) M_h if the reduced behaviour of M_l is a subset of the reduced behaviour of M_h. Such a refinement relation can be checked efficiently in polynomial time for finite state TTMs [18]. We may reverse the process and consider M_h an *abstraction* of M_l. Instead of verifying the detailed module M_l, we may instead verify the simpler abstraction M_h. At the logical level, refinement is just entailment, i.e. $R_l \rightarrow R_h$ where R_h, R_l are the RTTL reactive behaviours of modules M_h, M_l respectively.

Decomposition involves breaking a process M into modules M_1, M_2 so that $M = M_1 \parallel M_2$ (parallel composition). To properly handle decomposition, we need a compositional proof system, in which the temporal properties of a composite process can be deduced from the properties of its components.

Contribution and outline of this paper

The original version of RTTL was based on the floating version of temporal logic for fair transition systems. In this paper, we recast RTTL in the anchored framework of Manna and Pnueli [21], in which the validity of a temporal formula is anchored to the initial state of a trajectory (computation). The anchored version is simpler and more concise than the floating version, and past temporal operators are introduced without increasing the complexity of decision procedures for the propositional fragment of the logic [20].

Since RTTL is a conservative extension of the logic of [21], any theorem or rule in [21] holds in RTTL, and we may immediately use its compositional apparatus. RTTL extends the untimed logic with various axioms and rules specific to timing properties. A decomposition theorem for real-time modular reasoning is provided in Section 5. The theorem is applicable to all real-time reactive systems that map into TTMs and not just to a specific language. This may be contrasted to the compositional proof systems (one in temporal logic and one in an extended Hoare logic) provided in [15] for an Occam-like language. In our proof system, Hoare-like constructs are subsumed in the temporal proof system.

The DEVELOP tool uses the constraint logic programming language PrologIII [9] for automating state reasoning about infinite state modules. Using proof diagram heuristics in the tool, modular development and verification of systems can be performed. Weakest preconditions are used to characterize the effects of taking a transition rather than the transition relation of [21]. The weakest precondition characterization is useful for developing heuristics for proof strategies akin to the sequential program derivations of [10,11]. Timing constraints need not be known a priori. The heuristics are used to derive the timing constraints that will guarantee various properties such as mutual exclusion and response.

The development paradigms proposed in this paper include compositional reasoning, heuristics or decision procedures for verifying modules, and the support of a graphical toolset. These methods together with the decision procedures of the VERIFY tool, and the theory of refinement proposed in [18], provide a feasible approach to the systematic hierarchical development of real-time reactive systems.

An outline of the rest of this paper is as follows. In Section 2, the computational model of timed transition systems is defined. In Section 3, RTTL is presented in the anchored framework together with its proof system. The proof heuristics using proof diagrams and weakest preconditions are discussed in Section 4. The heuristics are applied to a mutual exclusion algorithm that depends upon timing constraints. In Section 5, modules and modular reasoning are discussed. A comprehensive example of modular reasoning is presented using a real-time resource allocator. In Section 6 the various tools for computer aided specification and verification in the framework are discussed.

2.0 TTMs — Timed Transition Models

State transitions graphs are often used by computing scientists, communication engineers, control theorists and circuit designers to describe physical systems. However, the basic transition as an event leading from one state to the next is much too "low-level" for the adequate representation of programming constructs. Therefore, we generalize transitions by allowing multiple simultaneous assignments to several variables where each variable may itself be a complex structure. The transitions of TTMs are timed versions of the fair transition systems of Manna and Pnueli [21], and related to the UNITY notion of a transition in the work of Chandy and Misra [7].

A TTM M is defined as a 5-tuple $M = (\mathcal{V}, I, \mathcal{T}, \mathcal{J}, \mathcal{F})$ consisting of a finite set of typed variables $\mathcal{V}$, an initial condition I, a finite set of transitions $\mathcal{T}$, and justice and fairness sets $\mathcal{J}, \mathcal{F}$ (subsets of $\mathcal{T}$). The variables set is subscripted with the TTM name when more than one TTM is involved, e.g. $\mathcal{V}_M$ refers to the variables set of M (likewise the other component of a TTM will be subscripted where necessary).

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

- *Spontaneous transitions* may occur at any point in time that they are enabled, or they may never occur. An example is the event of a device failure. In the sequel, spontaneous transitions are indicated by the fact that their upper time bound is infinity (∞).
- *Just transitions* must eventually occur if they are continually enabled. Fairness is a stronger version of justice which requires a transition to be taken if it becomes eligible infinitely often. Fairness too is a qualitative property.
- *Timed transitions* must occur within an exact interval specified by a lower and an upper time bound. For example, it may take between 3 and 6 ticks of the clock for a message to reach the recipient.

The TTM/RTTL framework can treat all the different type of transitions [21,22]. This paper focuses on real-time properties, and we therefore restrict our attention to the 3-tuple $M = (\mathcal{V}, I, \mathcal{T})$ without a justice of fairness constraints, and our examples will involve spontaneous and timed transitions. For completeness, the semantic definition of justice and its resulting proof rule are provided.

2.1 Definition of a Timed Transition Model

The basic components of a TTM $M = (\mathcal{V}, I, \mathcal{T})$ are defined below:

- The *variable set* $\mathcal{V}$ is a finite set of typed variables that always has two distinguished variables: t (the current clock time) and ε (the event variable). The event variable is useful for specifying the occurrence of a transition from a particular state. Besides the two distinguished variables, there may also be *data variables* that range over data domains used in the TTM, such as integers, lists

or sets. *Activity variables* (also called control variables) may be used to indicate progress in the execution of the various concurrent threads or processes of the TTM. Activity variables range over locations in the TTM called *activities*.

Each variable $v \in \mathcal{V}$ has an associated range of values $type(v)$. For example, for the clock variable t we usually set $type(v) = \{0, 1, 2, \ldots\}$. The type of the event variable ε is just the set of transitions (as will be illustrated later), i.e. $type(\varepsilon) = \mathcal{T}$.

A *state* s of the TTM is a mapping that assigns to each variable $v \in \mathcal{V}$ a value $s(v)$ in $type(v)$. The set of all states is denoted by $\mathcal{S}$.

The *restricted state* $\hat{s}$ is the restriction of the state s to all variables except for the event variable ε (an example of a restricted state is given later in (EQ 3)). The set of all restricted states is denoted by $\hat{\mathcal{S}}$.

- The *initial condition* I is a boolean valued expression in the variables set that characterizes the states at which the execution of the TTM can begin. A state satisfying I is called an initial state.
- The *transition set* $\mathcal{T}$ is a finite set of transitions $\tau = (e_\tau, h_\tau, l_\tau, u_\tau)$ where e_τ is the enabling condition of the transition, h_τ is the transformation function, l_τ is the lower time bound and u_τ is the upper time bound, where $l_\tau, u_\tau \in type(t) \cup \{\infty\}$. Every transition set has a distinguished transitions $tick$ (the clock tick). In the sequel, a precise definition of a transition will be given. A transition is guarded by its enabling transition, its lower and upper time bounds determine its moment of occurrence with relation to the number of ticks from its moment of enablement, and its transformation function determines the successor state.

  The *enabling condition* is a boolean valued expression in the variables set that asserts under what condition a state s may have successor states via the transition τ. The *transformation function* h_τ is a partial function $h_\tau : \hat{S} \to 2^{\hat{S}}$ that maps the restricted state $\hat{s}$ in $\hat{\mathcal{S}}$ to a set of successor states that are obtained when τ is taken. The transformation function is defined for all restricted states $\hat{s} \in \hat{\mathcal{S}}$ in which the enabling condition e_τ evaluates to true.

An *expression* of a TTM is constructed from the variables in $\mathcal{V}$, from constants such as 0 (the number zero) and $\varnothing$ (the empty set), to which functions such as + (integer addition) or $\cup$ (set union), and relations such as $\geq$ (greater than) or $\supset$ (subset) are applied. For example, $v_1 + 5v_2$ and $v_3 \cup v_4$ are expressions. Expressions also have a type depending on the types of the constants, variables, functions and relations. For example, the type of $v_1 + 5v_2$ is the integers.

A *boolean expression* has a type $\{true, false\}$. For example

$$\neg((v_1 = 0) \vee (v_2 \subseteq \{1, 2, 3\}))$$

is a boolean expression.

A *state-formula* (or predicate) of a TTM is constructed out of boolean expressions, boolean connectives and quantification over dummy variables. For example

$$\forall d | d{:}integer \cdot (v_1 \geq d) \rightarrow (v_2 = 0) \qquad \text{(EQ 1)}$$

is a state-formula that is universally quantified over the dummy integer variable d. Type declarations such as $d{:}integer$ are left out if its is clear from the context what they the type of the dummy is. The free variables in the formula are usually variables in the variable set $\mathcal{V}$ (which are called *flexible variables* as they assume different values in different states). By contrast, we may add to the variables set a set of *rigid variables*, which have the same value in all states of a sequence (computation).

An example of a state s of a TTM with $\mathcal{V} = \{\varepsilon, t, v_1, v_2\}$ is given by

$$s = \langle \varepsilon{:}push, t{:}10, v_1{:}5, v_2{:}7 \rangle \qquad \text{(EQ 2)}$$

i.e. s maps the event variable ε to the transition *push*, the clock variable t to 10, and the variables v_1 and v_2 are mapped to 5 and 7 respectively. The restricted state $\hat{s}$ corresponding to s is

$$\hat{s} = \langle t{:}10, v_1{:}5, v_2{:}7 \rangle \qquad \text{(EQ 3)}$$

(i.e. $\hat{s}$ is the same as s except that the event variable ε is left out).

The assignment of values to variables, by a state, can be uniquely extended in the usual way to expressions and state-formulas. For example, $s(v_1 + 5v_2) = 40$ where $s(v_1) = 5$ and $s(v_2) = 7$. The state-formula in (EQ 1) evaluates to *false* in the state s *given above*. If a state-formula p evaluates to *true* in a state s, then we write

$$s \Vdash p \qquad (s \textit{ state-satisfies} \text{ the formula } p)$$

A state provides values for all the free variables in a state-formula and hence also provides sufficient information to determine whether the satisfaction relation holds. Quantification over dummy variables does not effect the determination of satisfaction because it is irrelevant how the state interprets the dummy variables. Thus a state-formula is fully interpreted by a state. This is in contrast to formulas of temporal logic (see later) which cannot be interpreted in a single state but must be interpreted over an infinite sequence of states.

Consider a state s over the variables set $\mathcal{V} = \{\varepsilon, t, x, y, z\}$ given by

$$s = \langle \varepsilon{:}push, t{:}3, x{:}0, y{:}1, z{:}2 \rangle .$$

Then the satisfaction of the state-formula

$$((x + z = 2y) \rightarrow (y > z)) \wedge (\varepsilon = push)$$

in the state s is computed as follows:

$$\begin{aligned}
& s \Vdash (x + z = 2y \rightarrow y > z) \wedge (\varepsilon = push) \\
\equiv\ & (s(x) + s(z) = 2(y) \rightarrow s(y) > s(z)) \wedge (s(\varepsilon) = push) \\
\equiv\ & ((0 + 2) = 2 \times 1) \rightarrow (1 > 2)) \wedge (push = push) \\
\equiv\ & (true \rightarrow false) \wedge true \\
\equiv\ & false
\end{aligned}$$

i.e. the state does *not* satisfy the state-formula.

2.2 An example of a TTMchart

In this paper, TTMcharts are used as concrete representations of real-time reactive systems. Petri nets, programming languages or communicating processes could also have been used, and transformed into TTMs. However, the hierarchical visual constructs of TTMcharts are convenient in systems that are combinations of hardware devices (e.g. sensors, valves pumps) and software. TTMcharts have the visual hierarchical and compositional constructs of statecharts [12] such as nested boxes and arrows to arbitrary depth for state decomposition via products (i.e. AND-composition for concurrency) or disjoint union (i.e. XOR-composition for sequential operations). Communication is via the blocking rendezvous rather than the broadcast mechanism of statecharts, which is more convenient for representing real-time code in Ada or Occam.

A tool called *StateTime* automatically transforms TTMcharts into generic TTMs which can then be automatically verified [27]. The tool will be discussed in more detail in Section 6.0. We now provide an example of a TTMchart and its transformation into a TTM.

TABLE 1. TTM transitions corresponding to the TTMchart *InSensor* (see Fig. 1)

Name	Enabling Condition	Transformation Function	Lower	Upper
first	$(x = ok) \wedge (z = 0)$	$[z{:}1, c{:}1]$	0	∞
add	$(x = ok) \wedge (z = 1) \wedge (c \leq 10)$	$[c{:}c + 1]$	4	∞
reset	$(x = ok) \wedge (z = 1) \wedge (c = 0)$	$[z{:}0]$	1	2
fail	$(x = ok)$	$[x{:}bad]$	0	∞
fix	$(x = bad)$	$[x{:}ok, z{:}0, c{:}0]$	2	150
tick	$true$	$[t{:}t + 1]$	—	—

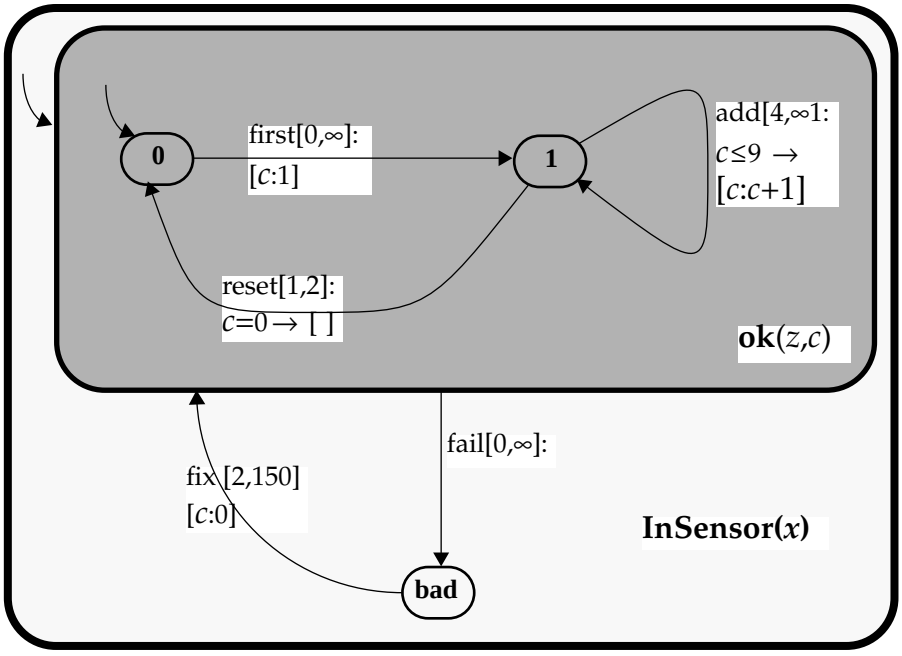
The TTMchart *InSensor* in Fig. 1 represents a sensor some distance from a level crossing gate that is situated at the point where a railway track crosses a road. The sensor detects trains that are about to enter the crossing area. The sensor information can be used to lower or raise the gate to ensure that the gate is down whenever there are trains in the area. The trains move only in one direction from *InSensor* at the entrance to the crossing to another sensor *OutSensor* at the exit to the crossing area.

Any number of trains can be in or near the crossing area (for example, a number of single engines without accompanying cars). A count *c* is kept of the number of trains in the area (the count is incremented by entering trains and decremented by exiting trains).

In Fig. 1, the TTMchart *InSensor* is the XOR-composition (XOR stands for exclusive or) of the subactivities *ok* and *bad*. To be in *InSensor* is to be in *ok* or *bad* but not both at the same time. The activity *ok* is annotated with a little arrow with no source to indicate that it is the *default* start activity of *InSensor*.

In Section 2.5, AND-composition of charts is used to model concurrency. A TTMchart which is AND-decomposed into subactivities must be in all of its sub-

FIGURE 1. The TTMchart *InSensor*— detects trains entering a crossing area



activities simultaneously. Thus, $Sensor = InSensor \parallel OutSensor$, and both sensors have the train count c as a shared variable.

We use the terms *activities* or *subactivities* for the states (substates) of statecharts. The word "state" is reserved in the sequel for references to the global assignment of values to all variables (e.g. see the state s in (EQ 2)). The top most activity *InSensor* together with all of its sub-activities is called a *TTMchart*.

The activity *bad* has no further internal structure, and is therefore called a *basic activity*, while *ok* is a *structured activity* as it is itself XOR-decomposed into the activities 0 and 1. The basic activity *bad* is called a *sibling* of *ok*. *InSensor* is the *root* activity, and the root activity together with all its siblings, descendants and transitions is called a TTMchart.

The *default* activity of *ok* is 0 as shown by the little arrow. Activity names may start with a letter (e.g *bad*, *ok*, *InSensor*) or a number (e.g. 0 and 1).

The structured activity *ok* is an abstraction of its subactivities 0 and 1. This abstraction allows for the expression of properties common to its components, thus facilitating hierarchical constructions. For example, an event originating at the boundary of a structured activity applies to all its subactivities. Thus the single event *fail* is used to describe the common property that no matter what the current subactivity in *ok* is, the sensor may fail at any point in time. Without the abstracting activity *ok*, two events (edges) would be needed to model sensor failure (one transition from 0 to *bad* and one from 1 to *bad*). So too, the event *fix* ends at the outer contour of *ok*, thus indicating that its destination is the default activity of *ok* (in this case 0).

The real-time behaviour of the TTMchart *InSensor* is described by the time bounds of its events. There is a conceptual global clock that ticks infinitely often.

An event waits the lower bound number of ticks before it becomes eligible to be taken. It must be taken or become disabled by the upper time bound number of ticks.

The event *fail[0,∞]* models a spontaneous event which can occur at any moment (or never), as indicated by the upper time bound of infinity. The time bounds of the transition *fix[2,150]* indicates that the transition *fix* must occur but only after being in the activity *bad* for between 2 and 150 ticks inclusive.

When a transition such as *first:[c:1]* occurs, it updates the data variable *c* to the expression on the right of *c* (i.e. c is assigned the value 1 indicating that the first train has arrived). So too, if *add:[c:c+1]* occurs, it increments the current value of *c* by one to indicate one more train has arrived.The interarrival rate of trains is constrained by the lower time bound 4 of *add*.

A guard may be placed on a transition to indicate that the transition may occur only if the guard is true. For example, since no more than 10 trains can ever be fitted into the crossing area, the guard on the transition *add* is $c \leq 9$. In general, guards may be any state-formula. If a transition is not annotated with a guard then the guard is assumed to be *true*.

The behaviour of the sensor is nondeterministic. For example, if the current subactivity is 1 (in *ok*), then it is possible that a new train may arrive (*add*), or the sensor may fail at any moment (*fail*). The TTMchart describes the possible events only not their probability of occurrence. If the sensor fails then *add* becomes disabled and *fix* becomes enabled, but not eligible to occur until the clock has ticked twice.

2.3 The TTM corresponding to a TTMchart

In order to define the precise behaviour of a TTMchart it is necessary to transform the chart into a TTM with its associated variables set, initial condition and set of transitions. Each structured activity has its own *activity variable* ranging over its subactivities. For the chart in Fig. 1, two activity variables x (for *InSensor*) and y (for *ok*) are required, with types:

$$\begin{aligned} type(x) &= \{ok, bad\} \\ type(z) &= \{0,1\} \end{aligned} \qquad \text{(EQ 4)}$$

In order to assert that the sensor is in (atomic) activity 0 of *ok*, we may write the state-formula $(x = ok) \wedge (z = 0)$. To assert that the current activity is *bad* write $(x = bad)$. A new activity variable is needed for each atomic activity that is refined with further internal structure. The StateTime tool supports such hierarchical top down development, as well as supporting bottom up construction of smaller parts into larger systems via OR and AND composition.

The sensor as a TTM is defined as $InSensor = (\mathcal{V}_{InSensor}, I_{InSensor}, \mathcal{T}_{InSensor})$. The variables set of the sensor with associated types is:

$$\mathcal{V}_{InSensor} = \{\varepsilon, t, x, z, c\}$$
$$type(\varepsilon) = \{init, tick, first, add, reset, fail, fix\}$$
$$type(t) = type(c) = \{0, 1, 2, \ldots\} \quad \text{(EQ 5)}$$

and the types of x and z are given in (EQ 4). The initial condition is given by the state-formula:

$$I_{InSensor} \stackrel{\text{def}}{=} (x = ok \wedge z = 0 \wedge c = 0) \quad \text{(EQ 6)}$$

The set of transitions $\mathcal{T}_{InSensor}$ is given in Table 3. Each event (edge) in the chart is mapped to a corresponding transition of the same name. The enabling condition of a transition is the conjunction of its guard and a state-formula describing the activity in which the chart must be for the transition to fire. For example, the guard of the transition *reset* is $(c = 0)$ and the state-formula describing the activity in which the transition is taken is $(x = ok \wedge z = 1)$. Thus the enabling condition of *reset* is

$$e_{reset} \stackrel{\text{def}}{=} (c = 0) \wedge (x = ok \wedge z = 1)$$

The transformation function of a transition is provided in simultaneous assignment format $[v_1{:}a_1, v_2{:}a_2, \ldots, v_i{:}a_i, \ldots, v_k{:}a_k]$ where each of the variables v_i is assigned the value of the expression a_i. The type of the expression a_i must be the same as the type of the variable v_i. When the transition is taken, the variables listed in assignment format are changed, but all other variables are left unchanged. For example, suppose the sensor is in the state

$$s_i = \langle \varepsilon{:}\, first, t{:}3, x{:}ok, z{:}0, c{:}0 \rangle$$

i.e. the sensor is in the sub-activity 0 of the *ok* activity, the train count is 0 and the clock indicates that 3 ticks have elapsed since initialization. In this state, the transition *first* is taken leading to a new state s_{i+1} where $s_{i+1} = \langle \varepsilon{:}?, t{:}3, x{:}ok, z{:}1, c{:}1 \rangle$, as determined by the transformation function of the transition $h_{first} = [z{:}1, c{:}1]$, i.e. the activity variable z and the data variable c change as shown in the successor state s_{i+1}, but all other variables remain the same. The value of the event variable ε remains unknown in this successor state until it becomes known what transition is taken (say *tick*). The complete successor state is then fully defined by

$$s_{i+1} = \langle \varepsilon{:}\text{tick}, t{:}3, x{:}ok, z{:}1, c{:}1 \rangle$$

The transformation function h_{first} does not include an assignment to the event variable as it is unknown at the time that *first* is taken that it will be followed by *tick*. Hence the effect of the transformation function of *first* must be stated in terms of restricted states, i.e. $\hat{s}_{i+1} = h_{first}(\hat{s}_i)$.

When a transformation function h is given in assignment format, then there is only one (restricted) successor state, i.e.

$$h{:}\hat{\mathcal{S}} \rightarrow \hat{\mathcal{S}} \text{ with } \hat{s}_{i+1} = h(\hat{s}_i).$$

The assignment format is convenient for developing proof heuristics using weakest preconditions, and is sufficient in most cases. In general, however, a state may

have many successor states, in which case $\hat{s}_{i+1} \in h(s_i)$. The transition *choose(x)* (Section 5.3.3.) allows for nondeterministic choice of the value of the variable x in the successor state, and hence cannot be given in assignment format. This presents no problem as its weakest precondition is well-defined.

An execution or computation performed by a TTM is called a *trajectory*. A trajectory σ of a TTM is an infinite sequence of states

$$\sigma = s_0 s_1 s_2 \ldots s_i s_{i+1} \ldots, \tag{EQ 7}$$

$$\text{or alternatively } \sigma = \hat{s}_0 \xrightarrow{\tau_0} \hat{s}_1 \xrightarrow{\tau_1} \ldots \rightarrow \hat{s}_i \xrightarrow{\tau_i} \hat{s}_{i+1} \xrightarrow{\tau_{i+1}} \ldots. \tag{EQ 8}$$

At each (restricted) state $\hat{s}_i$ there is an output transition τ_i to the successor $\hat{s}_{i+1}$. The full state consists of the restricted state and the transition taken at that restricted state, i.e. $s_i = (\hat{s}_i, \tau_i)$ where $s_i(\varepsilon) = \tau_i$. The tick transition is a diligent transition that is always ready to be taken even if no other transitions are enabled.

In process algebras transitions (events or actions) are preeminent while in extended state machine formalisms states are preeminent. In TTM/RTTL both states and events (using the distinguished event variable ε) are directly specified. For example, the RTTL formula $(x = bad \wedge c = 6) \rightarrow \Diamond(\varepsilon = \text{fix})$ specifies that if the TTM is in the "bad" *state* with the train count at 6, then eventually the "fix" *event* will occur (i.e. a state will be reached from which the fix event must be taken).

The first nine states of a possible trajectory $\sigma = s_0 s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 \ldots$ of *InSensor* are shown in Table 2. The first state s_0 must satisfy the initial condition, i.e. $s_0 \vDash I_{InSensor}$. The prefix shown in the table can be extended by showing what

TABLE 2. A trajectory σ of the sensor.

sensor variables	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	…
ε	tick	tick	first	fail	tick	tick	tick	fix	?	…
t	0	1	2	2	2	3	4	5	5	…
x	ok	ok	ok	ok	bad	bad	bad	bad	ok	…
z	0	0	0	1	1	1	1	1	0	…
c	0	0	0	1	1	1	1	1	0	…

transition (e.g. *tick* or *first*) occurs in s_8, and using the transformation function to compute the successor. The trajectory in Table 2 may be displayed graphically as:

$$s_0 \xrightarrow{tick} s_1 \xrightarrow{tick} s_2 \xrightarrow{first} s_3 \xrightarrow{fail} s_4 \xrightarrow{tick} s_5 \xrightarrow{tick} s_6 \xrightarrow{tick} s_7 \xrightarrow{fix} s_8 \rightarrow \cdots$$

2.4 Semantics of TTMs — legal trajectories

We saw in the sensor example that a trajectory is a sequence of states whose initial state satisfies the initial condition, whose successor states are obtained by applying enabled transitions according to the transformation functions, and

whose occurrence of transitions in any state must satisfy the lower and upper time bound requirements. Such a sequence of states will be called a *legal trajectory* of the TTM, and its precise definition is provided below. First a few definitions are required.

A transition τ is said to be *enabled* in a state s if $s \Vdash e_\tau$ (i.e. if s satisfies the enabling condition of τ). Similarly, a transition τ is *disabled* in a state s if $s \Vdash \neg e_\tau$. A transition τ is *taken* (or *occurs*) in a state s if $s \Vdash (\varepsilon = \tau)$.

Given a state s_i of a trajectory σ, we define the number of ticks until s_i as follows:

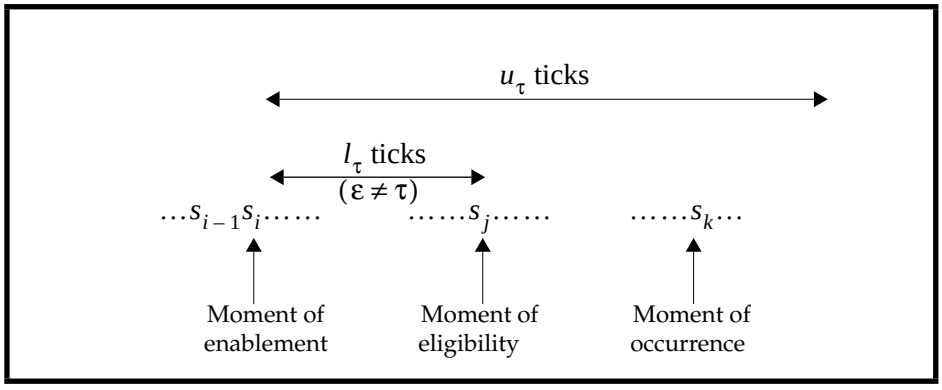
$$|s_i, \sigma| \stackrel{\text{def}}{=} \text{the } \textit{number of ticks} \text{ from the initial state of the trajectory } \sigma \text{ to the state } s_i$$

$$\equiv \#j \bullet 0 \le j < i \bullet s_j \Vdash (\varepsilon = tick) \qquad \text{(EQ 9)}$$

(where #j is the counting quantifier [Gries 1985, #59] that denotes the number of times that the expression $s_j \Vdash (\varepsilon = tick)$ is true in the range $0 \le j < i$). For example, for the trajectory in Table 2, $|s_6, \sigma| = 4$. When it is clear which trajectory the state s_i refers to then the trajectory can be left out and we write $|s_i|$ instead of $|s_i, \sigma|$. Where no confusion will occur, we may also abbreviate $|s_i|$ to its position number in the trajectory $|i|$. Note that $|s_0| = 0$ (the number of ticks until the initial state is zero).

Fig. 2 displays the three important moments needed for defining the time bounds: the moments of *enablement*, *eligibility* and *occurrence*. A transition τ is constrained not to occur between its moments of enablement and eligibility (despite being enabled) due to its lower time bound. At its moment of eligibility the transition τ may be taken. By the time the upper time bound number of ticks has elapsed, τ must either have been taken or become disabled (by being preempted by some transition that disables τ). The formal definition of these various moments will now be given.

FIGURE 2. Moments of enablement, eligibility and occurrence



A state s_i (of a trajectory σ) is a *moment of enablement* $moe(s_i, \tau)$ for a transition τ if the transition is reenabled at s_i, i.e.

$$moe(s_i, \tau) \stackrel{\text{def}}{=} [s_i \Vdash e_\tau] \wedge ([s_{i-1} \Vdash \neg e_\tau \vee (\varepsilon = \tau)] \vee (i = 0)) \qquad \text{(EQ 10)}$$

Thus s_i is a moment of enablement if τ is enabled in s_i, and in the previous state either τ is disabled or τ occurred (and after occurring remains enabled[2]), or there is no previous state and the current state s_i is an initial state. (By convention $s_j \models p$ is false if $j < 0$).

A state s_j is a *moment of eligibility* $mel(s_j, \tau, s_i)$ for a transition τ if the lower time bound number of clock ticks has occurred since its moment of enablement s_i, i.e.

$$mel(s_j, \tau, s_i) \stackrel{\text{def}}{=} (j \geq i) \wedge moe(s_i, \tau) \wedge [s_{j-1} \models (\varepsilon = tick)] \wedge [|s_j| = |s_i| + l_\tau] \qquad \text{(EQ 11)}$$

If s_i is the moment of enablement for τ, then the lower time bound condition states that τ may not occur up to and including its moment of eligibility, i.e.

$$lower(s_i, \tau) \stackrel{\text{def}}{=} (l_\tau > 0) \rightarrow \exists j \geq i \,|\, mel(s_j, \tau, s_i) \wedge (\forall m \,|\, i \leq m < j \cdot s_m \models (\varepsilon \neq \tau)) \qquad \text{(EQ 12)}$$

The upper time bound constraint of τ states that either τ occurs or becomes disabled by the time u_τ ticks have elapsed from the moment of enablement s_i, i.e.

$$upper(s_i, \tau) \stackrel{\text{def}}{=} (u_\tau < \infty) \rightarrow \exists k \geq i \,|\, s_k \models (\varepsilon = \tau \vee \neg e_\tau) \wedge [|s_k| \leq |s_i| + u_\tau] \qquad \text{(EQ 13)}$$

Definition of a Legal Trajectory $\sigma \in \Sigma_{\mathcal{M}}$:

A legal trajectory σ of a TTM $M = (\mathcal{V}, I, \mathcal{T})$ is defined to be an infinite sequence of states $\sigma = s_0 s_1 s_2 \ldots s_i s_{i+1} \ldots$ satisfying the following requirements:

- *Initialization*: The first state of the trajectory satisfies the initial condition, i.e. $s_0 \models I$.
- *Succession*: For each state s_i, $s_i \models (\varepsilon = \tau) \rightarrow e_\tau$ (i.e. if the transition τ is taken at s_i then τ must be enabled in s_i), and $\hat{s}_{i+1} \in h_\tau(\hat{s}_i)$. (If the transformation function is in assignment format then s_i has a single successor state s_{i+1} with $\hat{s}_{i+1} = h_\tau(\hat{s}_i)$.)
- *Ticking*: The clock ticks infinitely often, i.e. for each state s_i in the trajectory there is always a later state s_j (where $j \geq i$) such that $s_j \models (\varepsilon = tick)$.
- *Justice*: If a justice set $\mathcal{J}$ is defined (as explained in the beginning of this section), then for each $\tau \in \mathcal{J}$, if τ is continually enabled beyond some state s_i then τ must be taken infinitely often beyond s_i.
- *Real-Time*: For each moment of enablement of each transition in the trajectory, the relevant lower and upper time bound constraints must apply, i.e.

$$\forall \tau \in \hat{\mathcal{T}} \cdot \forall i \geq 0 \,|\, moe(s_i, \tau) \rightarrow lower(s_i, \tau) \wedge upper(s_i, \tau)$$

where $\hat{\mathcal{T}} = \mathcal{T}_M - \{tick\}$.

The set of all legal trajectories of a TTM M is denoted by Σ_M. The initialization, succession and real-time requirements are *finitary* constraints, because they only constrain the finite prefixes of a legal trajectory. There is always a finite prefix of the trajectory in which a violation of the constraint can be detected. No extension of the faulty prefix can therefore be a legal trajectory.

2. e.g. a periodically occurring selfloop.

The *infinitary* requirements of ticking and justice, constrain the entire infinite sequence. Violation of the infinitary requirements can never be detected in a finite prefix. Every finite sequence that satisfies the finitary requirements can be extended into an infinite sequence that satisfies all five requirements. The need for the justice requirement will be explained more fully in Section 2.6.

Refinements

Consider a TTM M having an observable (usually an input or output) variable y. A *reduced trajectory* σ^r of the TTM (relative to y), corresponding to the trajectory σ, is obtained from σ by (a) replacing each state s_i of σ with a reduced state which is the restriction of s_i to y, and (b) omitting from the trajectory any reduced state that is identical to its predecessor, but not identical to all its successors (so as not to delete suffixes which are infinite repetitions of the same state). An example of such a reduced trajectory is: $\langle y{:}0\rangle\langle y{:}1\rangle\langle y{:}2\rangle\ldots$. The set of all reduced trajectories of M is called its reduced behaviour.

A reduced trajectory can also be obtained with respect to an observable subset $\mathbb{O}$ of the TTM variables. We require that $t \in \mathbb{O}$ where t is time variable. A lower level TTM M_l is a *refinement* of M_h relative to $\mathbb{O}$ if the reduced behaviour of M_l is a subset of the reduced behaviour of M_h. Since $t \in \mathbb{O}$ it follows that M_l agrees with M_h on when to take the clock ticks so that real-time behaviour is preserved in any refinement.

2.5 AND–composition of TTMs

In the level crossing example (Fig. 1), the sensor at the entrance to the train crossing area works in parallel with the sensor at the exit. This is represented by the AND-composition of *InSensor* and *OutSensor* in Fig. 3, for which we have $Sensors = InSensor \parallel OutSensor$. To be in the root activity *Sensors* is to be in both subactivities *InSensor* and *OutSensor* simultaneously (there are two separate threads of execution).

AND-composition of two charts can be represented by parallel composition of the two corresponding TTMs. Given

$$M_1 = (\mathcal{V}_1, I_1, \mathcal{T}_1, \mathcal{J}_1) \quad \text{and} \quad M_2 = (\mathcal{V}_2, I_2, \mathcal{T}_2, \mathcal{J}_2)$$

the parallel composition

$$\begin{aligned} M &= M_1 \parallel M_2 \\ &= (\mathcal{V}, I, \mathcal{T}, \mathcal{J}) \end{aligned}$$

is defined as follows:

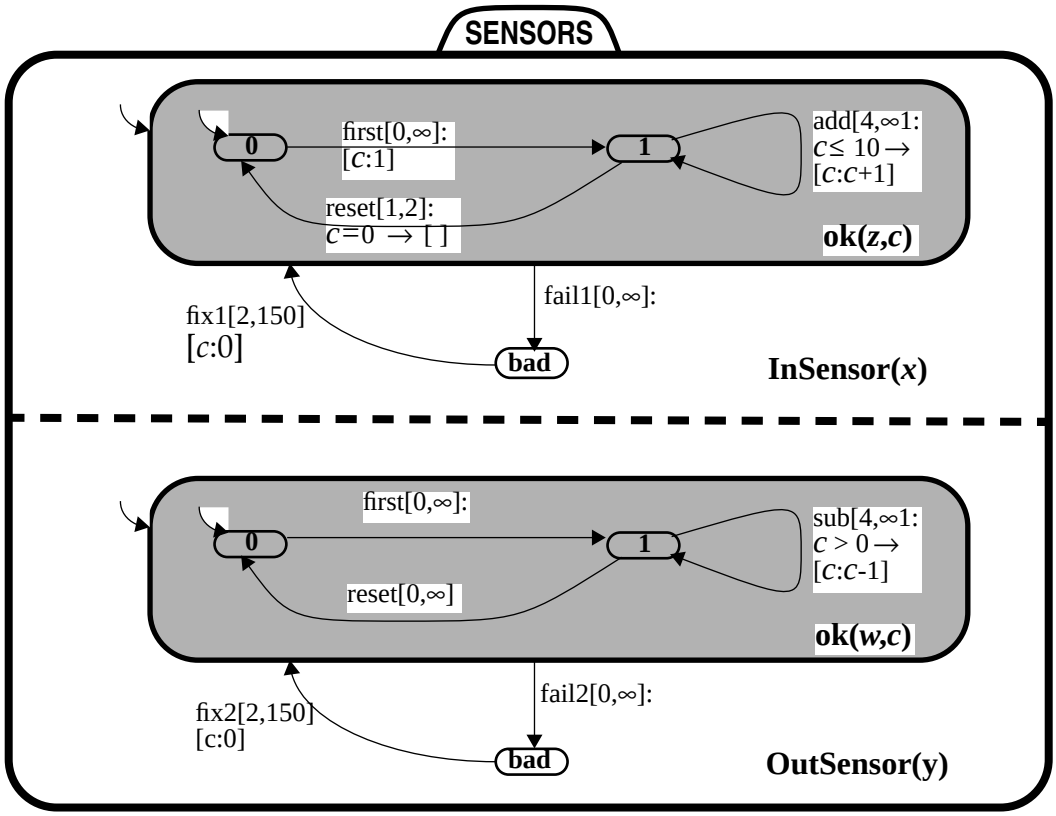
$$\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2$$

$$I = I_1 \wedge I_2$$

$$\mathcal{T} = \mathcal{T}_1 \parallel \mathcal{T}_2$$

$$\mathcal{J} = \mathcal{J}_1 \cup \mathcal{J}_2$$

FIGURE 3. AND-composition — Sensors = InSensor || OutSensor



where $\mathcal{T}_1 \parallel \mathcal{T}_2$ is defined as $\mathcal{T}_1 \cup \mathcal{T}_2$ if there are no *shared transitions* (i.e. all transitions are local). A shared transition models a situation in which there is an interlock or rendezvous between two parallel processes that prevents either one from making progress until they are both able to take the shared transition (as in CSP [14]).

The transition $first$ is an example of a shared transition which involves the simultaneous execution of *InSensor* and *OutSensor* in Fig. 3. Thus, the *InSensor* "arms" the *OutSensor* via $first$ so that the latter decrements the counter c by one every time a train exits. Either sensor is blocked at subactivity 0 in *ok* until both are enabled to be taken simultaneously.

The shared transition $first$ is the composite of the two components $first_I$ in *InSensor* and $first_O$ in *OutSensor.* By convention, the composite and component transitions all bear the same name. For explanatory convenience, we have labelled the composites as $first_I$, $first_O$.

The enabling condition of the composite is given by $e_{first} = e_{first_I} \wedge e_{first_O}$, i.e. $e_{first} \stackrel{\text{def}}{=} (x = ok \wedge z = 0) \wedge (y = ok \wedge w = 0)$, where x, z are the activity variables of *InSensor* and y, w the corresponding activity variables of *OutSensor.* The composite transformation function is the combination of assignment updates, i.e.

$$h_{first} = [z{:}1, c{:}1, w{:}1] .$$

The composite lower and upper time bounds are defined by

$$l_{first} = max(l_{first_I}, l_{first_O}) \quad \text{and} \quad u_{first} = min(u_{first_I}, u_{first_o})$$

which in this case are trivially zero and infinity respectively. In general, however, a component transition in one process can "force" its corresponding component in the other process to execute within the first component's upper time constraint.

The transition *reset* is also shared (Table 3). The transitions *add* and *sub* are examples of local transitions. These transitions do not occur in lockstep with any other transitions, but are each taken independently subject only to their own time bound constraints. A shared transition may have components in finitely many parallel TTMs. In such a case, all components must be taken in lockstep together.

If there are shared transitions, then $\mathcal{T}_1 \parallel \mathcal{T}_2$ is defined as the union of the two transition sets, but with the components of the shared transitions replaced by the corresponding composite transitions. The transition set of *Sensors* is shown in Table 3.

TABLE 3. Transition Set of the TTMchart "Sensors" (see Fig. 3)

Name	Enabling Condition	Transformation Function	Lower	Upper
first	$(x = ok \wedge z = 0) \wedge (y = ok \wedge w = 0)$	$[z{:}1, c{:}1, w{:}1]$	0	∞
add	$(x = ok \wedge z = 1 \wedge c \leq 10)$	$[c{:}c + 1]$	4	∞
sub	$(y = ok \wedge w = 1 \wedge c > 0)$	$[c{:}c - 1]$	4	∞
reset	$(x = ok \wedge z = 1 \wedge c = 0) \wedge (y = ok \wedge w = 1)$	$[z{:}0, w{:}0]$	1	2
fail1	$(x = ok)$	$[x{:}bad]$	0	∞
fail2	$(x = ok)$	$[x{:}bad]$	0	∞
fix1	$(x = bad)$	$[x{:}ok, z{:}0]$	2	150
fix2	$(y = bad)$	$[y{:}ok, w{:}0]$	2	150
tick	true	$[t{:}t + 1]$	—	—

Since the parallel composition of two TTMs is itself a TTM, the behaviour of the resulting composite TTM is also subject to the requirements for legal trajectories enumerated in Section 2.4. In order to develop a compositional proof system. it will be necessary to define an interface specification for a TTM (Section 5.1).

In the chart of Fig. 3, events in different TTMs having the same name are considered shared. In the StateTime tool, an event must be declared shared. If not, it is local and does not synchronize with any external transition. Hence both *fail1*, *fail2* can be called *fail* in StateTime. Which variables and events are exported is a matter that must ideally be described in the interface specification of a TTM.

2.6 Interleaving and Concurrency

An essential element of the generic TTM model is that concurrency is represented by *interleaving*. Two transitions in parallel processes are never taken at precisely the same instant, but take turns in executing transitions, which are thus treated as atomic. The interleaving approach reduces concurrency to nondeter-

minism. This is similar to the expansion law for process algebras such as CSP for which $\alpha \| \beta = \alpha + \beta$ where $\alpha + \beta$ stands for nondeterministic choice between the events α and β. Either event is eligible to be taken but may be ignored forever.

The interleaving model of computation simplifies analysis but must be used carefully to model truly concurrent systems which have *overlapped execution* (in which the execution of statements on different processors usually overlap with each other rather than interleave).

Two problems must be resolved to ensure that the interleaved model adequately represents true concurrency. These two problems revolve around the issues of *independent progress* and *interference*.

Independent Progress

In a truly concurrent system in which each process may reside on different processors, each processor is independently responsible for its own progress (as opposed to the multiprogramming case in which a single CPU is shared by multiple processes). Thus some action in each process is always eventually executed.

In an interleaved computation (that is not subject to the justice or real-time constraints), the only requirement is that the tick of the clock happen infinitely often. There is nothing to disallow a trajectory in which transitions from one process only are always chosen. Thus, interleaved trajectories may manifest behaviours that will never be observed in truly concurrent executions.

The justice requirement imposes restrictions on the basic model, guaranteeing progress for all processes in the interleaved model. The justice requirement tells the scheduler that builds up a trajectory that it cannot forever ignore a transition that is continually enabled beyond some point. There are higher levels of attention that can be paid to processes (e.g. fairness), but justice is the main requirement needed so that the interleaving model represents true concurrency. The justice constraint is imposed on $\alpha \| \beta$ by setting the justice set to $\mathcal{J} = \{\alpha, \beta\}$.

The real-time requirements ensure that even more attention must be paid to transitions to ensure the overall progress of the system. The justice set imposes a qualitative liveness constraint, whereas the real-time bounds can be used impose a hard time response.

Interference

Consider a process with shared variable y executing the statement **when** $y=y$ **do** S. The guard $y=y$ is tested in one atomic step in an interleaved computation and so always yields the value *true*. By comparison, overlapped execution of this statement will reference y twice (under naive implementation with no optimization). If a statement in some other concurrent process interferes with y by changing y between these two references, then the guard may evaluate to *false*.

The issue of interference is potentially problematic because the set of all interleaved trajectories may miss certain behaviours present in overlapped executions. The dual issue of independent progress is potentially problematic because the set of all interleaved trajectories may exhibit behaviours that are not manifested in actual concurrent execution.

There are two solutions to the interference problem. One possibility is to admit more interference in the interleaved computation (by refining a statement into more substeps of finer granularity). An assignment statement has distinct fetch, compute and store steps. The fewer such critical references in a transition the finer the grain of atomicity.

A second possibility requires more protection against overlapped executions. This can be achieved by enclosing the critical sections within release-request pairs of semaphore statements or by other constructs such as monitors [21].

3.0 RTTL — Real-Time Temporal Logic

The view of a TTM as a generator of legal trajectories is an *operational* description of the ongoing behaviour of the system modelled by the TTM. Real-Time Temporal Logic is a specification language that provides an alternative characterization of TTM behaviour that is more *descriptive*.

The language of temporal logic is constructed from state-formulas to which we apply temporal operators, boolean connectives and quantification. While state-formulas can be checked for satisfiability over a single state, temporal formulas must be checked for satisfiability over infinite sequences of states (trajectories). Thus, in general, temporal formulas are satisfied in some trajectories and falsified in other trajectories.

A set of temporal formulas (a *specification*) specifies any TTM whose legal trajectories satisfy all the formulas in the specification. A temporal specification rarely specifies a unique TTM — any TTM with the appropriate legal trajectories will do. If after developing a specification, we suddenly realize that the specification is incomplete, the situation can be rectified by adding the missing property to the specification as additional conjuncts in an incremental fashion.

The original version of RTTL was based on a future fragment of the floating interpretation of linear time temporal logic [22]. In the floating version, satisfiability is defined with respect to all positions of a trajectory (there is no special significance to the initial state of a trajectory). The anchored version introduced various difficulties such as invalidating the deduction rule and requiring suffix closure of legal trajectories. The anchored version was found to be simpler and more concise [20]. In the anchored version, satisfaction is defined with respect to position zero of a trajectory. Suffix closure is not required, and the deduction rule is valid.

In this section, we pose RTTL on the basis of the anchored version of untimed temporal logic [21]. RTTL inherits the proof system of the standard logic. In addition, RTTL has additional proof rules for dealing with real-time properties.

3.1 RTTL semantics

For each of the operators and subformulas allowed in RTTL, we present below a definition of its interpretation in a trajectory. This definition is based on the notion of a formula p holding at a position j, $j \geq 0$ in a trajectory σ. These definitions of satisfaction are:

$(\sigma, j) \models p$ — trajectory σ *satisfies* p *at position* j, and

$(\sigma, 0) \models p$ — trajectory σ *satisfies* p (p holds at the initial state), and we write $\sigma \models p$.

The symbol $\models$ is used for holding over trajectories while the symbol $\Vdash$ is used for holding over states.

1. *State-formulas*. If p is a state-formula then it is also an RTTL formula. For a state-formula p,

$$(\sigma, j) \models p \quad \text{iff} \quad s_j \Vdash p$$

2. *Boolean operators*. If p and q are RTTL formulas then so are $\neg p$ and $p \vee q$.

   For a negation $\neg p$, we define

$$(\sigma, j) \models \neg p \quad \text{iff} \quad \text{not } [\,(\sigma, j) \models p\,].$$

   For a disjunction $p \vee q$, we define

$$(\sigma, j) \models p \vee q \quad \text{iff} \quad [\,(\sigma, j) \models p\,] \text{ or } [\,(\sigma, j) \models q\,]$$

   It is easy to extend these definitions to the other boolean connectives $\wedge$, $\rightarrow$ and $\leftrightarrow$ using their definitions in terms of negation and disjunction.

3. The *next* operator $\bigcirc$. If p is an RTTL formula then so is $\bigcirc p$, and we define

$$(\sigma, j) \models \bigcirc p \quad \text{iff} \quad (\sigma, j+1) \models p.$$

4. The *waiting-for* (or *unless*) operator $\mathcal{W}$ (the formula $p \mathcal{W} q$ is read as: p waiting for q). If p and q are RTTL formulas, then so is $p \mathcal{W} q$, and we define

$$(\sigma, j) \models p \mathcal{W} q \quad \text{iff} \quad \begin{array}{l} \text{there exists a } k \geq j, \text{ such that} \\ (\sigma, k) \models q, \text{ and for every} \\ i,\ j \leq i < k,\ (\sigma, i) \models p, \\ \qquad \text{or} \\ \text{for all } k \geq j,\ (\sigma, k) \models p. \end{array}$$

   Thus, $p \mathcal{W} q$ holds at position j iff p holds continually from position j and onwards either until the next occurrence of q or throughout the rest of the trajectory.

The *future operators* $\bigcirc$ and $\mathcal{W}$ defined above, are a basic set that may be used to derive all the other future operators, as shown below:

Operator Name	Definition
$\Box p$ — *Henceforth*	$\Box p \stackrel{\text{def}}{=} (p \mathcal{W} \mathit{false})$
$\Diamond p$ — *Eventually*	$\Diamond p \stackrel{\text{def}}{=} \neg \Box \neg p$
$p \mathcal{U} q$ — *Until*	$p \mathcal{U} q \stackrel{\text{def}}{=} (p \mathcal{W} q) \wedge (\Diamond q)$

All the operators defined thus far are part of the standard linear time temporal logic. To capture real-time properties we define the *real-time until* operator $p\,\mathcal{U}_{[l,u]}\,q$, which is defined as:

$$(\sigma, j) \models (p\,\mathcal{U}_{[l,u]}\,q) \quad \text{iff} \quad \begin{array}{l}\text{there exits a } k \geq j, \text{ such that } (\sigma, k) \models q \text{ and} \\ l \leq (|s_k| - |s_j|) \leq u, \text{ and for every} \\ i,\ j \leq i < k,\ (\sigma, i) \models p.\end{array}$$

The notation $|s_k|$ was defined in (EQ 9) as the number of ticks from the initial position of the trajectory to the position k.

The ordinary until operator (a) predicts the eventual occurrence of q at some position subsequent to j, and (b) asserts that p holds continuously until (at least the first) occurrence of q. The real-time operator adds the assertion that there must be an occurrence of q between l and u ticks from position j (with p holding continuously true until then).

The following derived operators may be obtained using the real-time until operator:

$\Diamond_{[l,u]}\,p$	$\stackrel{\text{def}}{=\!=}$	$(true\,\mathcal{U}_{[l,u]}\,p)$	*bounded response*
$\Diamond_{\leq u}p$	$\stackrel{\text{def}}{=\!=}$	$\Diamond_{[0,u]}\,p$	(e.g p must become true within u ticks from now).
$\Box_{<l}p$	$\stackrel{\text{def}}{=\!=}$	$(p\,\mathcal{U}_{[l,\infty]}\,true)$	*bounded invariance* — p must remain true until l ticks of the clock have been taken).
$\Diamond_{=l}\,p$	$\stackrel{\text{def}}{=\!=}$	$(true\,\mathcal{U}_{[l,l]}\,p)$	*exact time* — p is true in exactly l ticks from now.

The real-time operators are called *bounded operators*.

Past operators may also be defined. For example, the *once* formula $\diamondminus p$ is defined by

$$(\sigma, j) \models \diamondminus p \quad \text{iff} \quad \begin{array}{l}(\sigma, k) \models p \text{ for some} \\ k,\ 0 \leq k \leq j.\end{array}$$

Thus $\diamondminus p$ holds at position j if iff p holds at position j or some preceding position. Bounded versions of the past operators can also be defined. Similarly, the *previous* formula $\ominus p$ is true at a position (which is not the first position) if p is true in the previous position, i.e.

$$(\sigma, j) \models \ominus p \quad \text{iff} \quad j > 0 \text{ and } (\sigma, j-1) \models p$$

The *previous* operator can be used to define a state-formula $first$ that asserts that there is no previous position. Since all positions (states) in a trajectory satisfy $true$, we have that

$$first \stackrel{\text{def}}{=\!=} \neg\ominus true. \qquad \text{(EQ 14)}$$

The reader is referred to [21] for a complete treatment of the past operators.

3.2 Simple examples of RTTL specifications

The following are some frequently used formulas and their verbal interpretations. The verbal interpretation characterizes the trajectories σ such that $\sigma \models f$ for the considered formula f, i.e. that f holds at position 0 of σ. For simplicity, assume that the subformulas p, q and r below are state-formulas.

- $p \rightarrow \Box q$

  If initially p then henceforth q.

  This formula states that if the trajectory satisfies p, then it also satisfies $\Box q$. A trajectory σ satisfies p if p is true in state s_0 (recall that p is a state-formula). The trajectory satisfies $\Box q$, if for all positions $j \geq 0$, q holds at s_j. Consequently, this formula expresses the following property that the trajectory σ satisfies: "if initially p then henceforth q".

- $\Box (p \rightarrow \Diamond q)$

  Every position satisfying p coincides with or is followed by a position satisfying q (there is no cause without effect).

  The subformula $p \rightarrow \Diamond q$, interpreted with respect to a position j of a trajectory, states the property: "if p holds at position j then q holds at some position after (or coinciding with) j". Adding the henceforth operator in front of the subformula states that this property holds for all positions $j, j \geq 0$. Consequently, the complete formula expresses the following property about the trajectory σ: every position that satisfies p coincides with or is followed with a position satisfying q. (Without the henceforth operator, the property would only apply to the initial position of the trajectory).

- $p \rightarrow (q \,\mathcal{U}_{[3,7]}\, r)$

  If p initially, then eventually between 3 and 7 ticks r, and q must hold continuously until then. This property is asserted only at the initial state.

- $\Box (p \rightarrow (q \,\mathcal{U}_{\leq 4} r))$

  Every position satisfying p is followed within 4 ticks by r, and q holds continuously until then.

  Unlike the immediately preceding property, this property is asserted at all states (because of the henceforth operator at the beginning of the formula).

- $\Diamond \Box p$

  Eventually permanently p.

- $\Box (p \rightarrow \Box_{<l} \neg q)$

  The property q cannot become true sooner than 3 ticks after any occurrence of the property p.

3.3 Validity, entailment and congruences

The table below summarizes the definitions needed for satisfaction, validity and entailment of temporal logic formulas.

Abbreviation:	Abbreviation is read as:	Definition of abbreviation
$s \Vdash p$	State s *state-satisfies* the state-formula p.	Already defined.
$\Vdash p$	State-formula p is *state-valid*.	p holds in every state.
$\Vdash^M p$	State-formula p is *M-state-valid* (M is a TTM).	p holds on every state that appears in a legal trajectory of M.
$\sigma \vDash p$	Trajectory σ *satisfies* RTTL formula p.	Already defined
$\vDash p$	RTTL formula p is *valid*.	$\forall \sigma \mid \sigma \vDash p$
$\vDash^M p$	RTTL formula p is *M-valid*.	$\forall \sigma \in \Sigma_M \mid \sigma \vDash p$
$F \vDash p$	The set of RTTL formulas F *semantically-entails* the formula p.	$\forall \sigma \mid (\sigma \vDash F) \rightarrow (\sigma \vDash p)$
$F \vDash^M p$	The set of RTTL formulas F *M-semantically-entails* the formula p.	$\forall \sigma \in \Sigma_M \mid \sigma \vDash \dot{F} \rightarrow \sigma \vDash p$
$p \Rightarrow q$	RTTL formula p *entails* formula q	$\Box (p \rightarrow q)$
$p \Leftrightarrow q$	RTTL formula p is *congruent* with formula q.	$\Box (p \leftrightarrow q)$
$\{p\}\,\tau\,\{q\}$	The transition τ *leads from* RTTL formula p *to* formula q. The Hoare-like triple $\{p\}\,\tau\,\{q\}$ is called the *leads-to* relation.	$(\varepsilon = \tau) \wedge p \Rightarrow \bigcirc q$
$\{p\}\,\mathcal{T}_1\,\{q\}$	The set of transitions $\mathcal{T}_1 \subseteq \mathcal{T}_M$ leads from RTTL formula p to formula q.	$\forall \tau \in \mathcal{T}_1 \mid \{p\}\,\tau\,\{q\}$

Consider a TTM M whose only data variable is y and whose only reduced trajectory is given by

$$\langle y{:}0\rangle\,\langle y{:}1\rangle\,\langle y{:}2\rangle \ldots . \qquad \text{(EQ 15)}$$

The state-formula $p{:}\,(y = 0)$ is not *M-state-valid* as there are reachable states in which it evaluates to false. However, p *is* M-valid (i.e. p holds at every initial position of every legal trajectory). The state-formula $y \geq 0$ is *M-state-valid*, and hence trivially $\Box (y \geq 0)$ is M-valid.

The relationship GEN below enables a two-way transformation between state and temporal validities for any state-formula p [21, p247]:

$$\textbf{GEN:} \qquad \Vdash p \quad \text{iff} \quad \vDash \Box p \qquad \text{and} \qquad \Vdash^M p \quad \text{iff} \quad \vDash^M \Box p \qquad \text{(EQ 16)}$$

Also, every general validity implies a TTM validity, i.e.

for any state-formula p and TTM M, if $\Vdash p$ then $\Vdash^M p$, and (EQ 17)

for any RTTL formula p and TTM M, if $\vDash p$ then $\vDash^M p$. (EQ 18)

3.4 The RTTL proof system

The general *verification problem* is: given a TTM M and a specification S (a set of RTTL formulas), prove that $\vDash^M S$ holds (i.e prove that each formula in S is M-valid). The RTTL deductive system of axioms and rules is used to prove M-validities. The RTTL proof system is divided into three parts:

- The *state reasoning* (**SR**) part of RTTL proof system consists of only one axiom that allows us to introduce at any step of a proof the line $\Vdash p$ if p is state-valid, e.g. $\Vdash (y = 0) \rightarrow (y \geq 0)$ may be introduced at any line of a proof with justification SR. We may then use the interface rule GEN (EQ 16) to derive, on the next line, $\vDash \Box((y = 0) \rightarrow (y \geq 0))$. The above two step proof may be written more concisely in one step with the justification SR+GEN. In this paper, we use the justification SR (state reasoning) to justify any distinctly non-temporal reasoning involving the use of SR or any other propositional, predicate or domain reasoning (e.g. for integers, lists and sets).
- The *temporal logic reasoning* (**TLR**) part of the RTTL proof system provides those axioms and rules that preserve validities over arbitrary (uninterpreted) trajectories. The Manna-Pnueli axiom FX5

  $$(p \Rightarrow \bigcirc p) \rightarrow (p \Rightarrow \Box p) \quad \text{(EQ 19)}$$

  is an example of an axiom of the TLR part. The rule MP given by $p, p \rightarrow q \vdash q$ is an example of a rule. Instead of reproducing the TLR part in this paper, we will justify any reasoning performed with this part by writing TLR, which means that the proof step so justified can be obtained by using the axioms, basic rules, theorems and derived rules of the Manna-Pnueli deductive system [21].
- The *TTM reasoning* (**MR**) part of the RTTL proof system provides the extra rules needed to reason about validities with respect to reachable states and legal trajectories of TTMs. For example, the formula $\Box(y = 0 \vee y \neq 0)$ is generally valid whereas the formula $\Box(y \geq 0)$ is not. However, $\Box(y \geq 0)$ is valid for the particular TTM whose only reduced trajectory is given in (EQ 15).

The introduction of the rule SR makes the RTTL proof system nonrecursive, i.e. it can no longer be effectively (algorithmically) checked that each step of a proof is fully justified. To counter this problem, we could introduce a deductive system for proving state-validities, and restrict axiom SR to only those formulas that can be proved in the state deductive system. Such a system could be any of the ones found in texts on the predicate calculus (and the Peano axioms for integers or whatever other data domain is required).

However, it is preferable make a separation between the process of establishing state-validity and temporal-validity. In many proofs of temporal properties of a particular TTM, much of the work involves establishing M-state-validities in the particular data domain treated by the TTM being verified (e.g. integers, sets or lists). This is then followed by a few applications of the interface rules (e.g GEN)

between the state and temporal systems, and a small amount of actual temporal reasoning.

The StateTime tool (Section 6.0) uses PrologIII [9] for establishing state-validity. PrologIII effectively and efficiently solves constraints on boolean, linear integer and rational data domains. We do not want to commit ourselves unnecessarily to a particular state deductive system when that is really dependent on the nature of the data domains (and its theorem provers) for the given TTM. We therefore maintain a strict separation between state and temporal reasoning, and summarize all the different possible ways of effectively obtaining state-validities by the single rule SR.

The general part of the RTTL proof system is described in detail in [21]. In this paper, we will therefore only need to describe the TTM part of the proof system in detail.

Entailment and Congruence.

The notions of entailment ($p \Rightarrow q$) and congruence ($p \Leftrightarrow q$) are stronger types of implication than the standard conditional $\rightarrow$ and biconditional $\leftrightarrow$ respectively. The formula $p \rightarrow q$ states only that $p \rightarrow q$ holds at the first position of a trajectory σ. The stronger entailment $\Box(p \rightarrow q)$ states that $p \rightarrow q$ holds at all positions of σ. If $\Box(p \leftrightarrow q)$ is valid, then p and q have the same truth value in all positions of every model.

Let $\varphi(f)$ be a formula schema with one or more occurrences of the subformula f. Then

$$\vDash (p \Leftrightarrow q) \rightarrow (\varphi(p) \Leftrightarrow \varphi(q)) \qquad \text{(EQ 20)}$$

The *substitutively property* described by (EQ 20) allows us to derive the validity of

$$(q \,\mathcal{U}\, \Box p) \Leftrightarrow (q \,\mathcal{U}\, (\neg \Diamond \neg p))$$

from the validity of $\Box p \Leftrightarrow \neg \Diamond \neg p$.

An occurrence of f in φ is said to be *positive* if f is embedded in an even (explicit or implicit) number of negations, and it does not occur in a subformula of the form $p \leftrightarrow q$ (which contains an implicit negative occurrence of p and q). If all occurrences of f in $\varphi(f)$ are positive, then we have the Manna-Pnueli rule that

$$\vDash (p \Rightarrow q) \rightarrow (\varphi(p) \Rightarrow \varphi(q)) \qquad \text{(EQ 21)}$$

For example, taking $\varphi(f) = \Box f$, we can show from (EQ 21) that

$$\vDash (p \Rightarrow q) \rightarrow (\Box p \Rightarrow \Box q)$$

Soundness and Semantic Justification

From the basic Manna-Pnueli axioms and rules, we can deduce theorems and derived rules. An example of a derived rule is:

Rule **E-TRNS** (entailment transitivity)

For any RTTL formulas p, q, r

$$\frac{p \Rightarrow q \text{ and } q \Rightarrow r}{p \Rightarrow r}$$

Rules (such as E-TRNS) from the general part of the proof system, hold in the set of all possible trajectories, as well as in subsets of trajectories. One such subset is the legal set corresponding to a TTM M. In the axioms and rules presented below for the MR part of the proof system, we assume that we are dealing with M-validity, i.e. with a subset of legal trajectories.

With regards to the MR part of the proof system, the *soundness* of the RTTL proof system requires that each new axiom that is introduced must be shown to be *M*-valid, and each new rule that is introduced must be shown to preserve *M*-validity.

One way to show validity is via the use of *semantic justification* — using the semantic definition of the temporal operators, we can show that the axiom is true in all legal trajectories. Consider the rule L-TRNS given below.

Rule **L-TRNS** (less than transitivity)

For any RTTL formulas p, q, r, s, t

$$\frac{p \Rightarrow (q\, \mathcal{U}_{\leq u_1}\, r) \qquad r \Rightarrow (s\, \mathcal{U}_{\leq u_2}\, t)}{p \Rightarrow (q \vee s)\, \mathcal{U}_{\leq u_1 + u_2}\, t}$$

The state-formula r is called the *link* formula. The soundness of the rule can be demonstrated by semantic justification. The first premise of the rule means that for any position j of a legal trajectory σ we have that $(\sigma, j) \vDash p \rightarrow (q\, \mathcal{U}_{\leq u_1}\, r)$, i.e. $(\sigma, j) \vDash p$ implies $(\sigma, j) \vDash (q\, \mathcal{U}_{\leq u_1}\, r)$. We may then argue that:

$(\sigma, j) \vDash (q\, \mathcal{U}_{\leq u_1}\, r)$

iff < by definition of $\mathcal{U}_{\leq u_1}$ >

$\exists k \geq j \,|\, (\sigma, k) \vDash r$ and $|k| - |j| \leq u_1$ and $(\forall i \,|\, j \leq i < k \cdot (\sigma, i) \vDash q)$

implies <by premise 2, for any k $[(\sigma, k) \vDash r] \rightarrow [(\sigma, k) \vDash (s\, \mathcal{U}_{\leq u_2}\, t)]$ >

$\exists k \geq j \cdot (\exists m \geq k \,|\, (\sigma, m) \vDash t$ and $|m| - |k| \leq u_2)$ and

$(\forall i \,|\, k \leq i < m \cdot (\sigma, i) \vDash s)$ and $|k| - |j| \leq u_1$ and $(\forall i \,|\, j \leq i < k \cdot (\sigma, i) \vDash q)$

implies < predicate logic, j, k, m non-negative integers, & eliminate k >

$\exists m \geq j \,|\, (\sigma, m) \vDash t$ and $|m| - |j| \leq u_1 + u_2$ and $(\forall i \,|\, j \leq i < m \cdot (\sigma, i) \vDash q \vee s)$

iff < by definition of $\mathcal{U}_{\leq u_1 + u_2}$ >

$(\sigma, j) \vDash (q \vee s)\, \mathcal{U}_{\leq u_1 + u_2}\, r$

Thus, for any position j of a legal trajectory σ we have that $(\sigma, j) \vDash p$ implies $(\sigma, j) \vDash (q \vee s)\, \mathcal{U}_{\leq u_1 + u_2}\, r$, which is the conclusion of MR1 as required.

In a similar fashion we may prove the soundness of T-TRNS rule below which is more general than the formulation of L-TRNS (L-TRNS may be derived from T-TRNS). Using semantic justification we can show that the bounded operators can be used to characterize the unbounded operators, i.e.

$$(p\, \mathcal{U} q) \stackrel{\text{def}}{=} [\exists u \geq l \,|\, (p\, \mathcal{U}_{[l, u]}\, q)] \qquad \text{(EQ 22)}$$

Rule **T-TRNS** (time transitivity)

For any RTTL formulas p, q, r, s, t

$$\frac{\models^{\mathcal{M}} p \Rightarrow (q\, \mathcal{U}_{[l_1, u_1]}\, r) \qquad \models^{\mathcal{M}} r \Rightarrow (s\, \mathcal{U}_{[l_2, u_2]}\, t)}{\models^{\mathcal{M}} p \Rightarrow (q \vee s)\, \mathcal{U}_{[l_1+l_2,\, u_1+u_2]}\, t}$$

where u, l are rigid variables having the same value in each state of a trajectory. It then follows that

$$\models^{M} (p\, \mathcal{U}_{[l, u]}\, q) \Rightarrow (p\, \mathcal{U} q) \qquad \text{(EQ 23)}$$

which means that the quantitative "until" operator entails the qualitative version. A complete characterization of the bounded operators can be provided in an analogous way to the fixed point characterizations of the unbounded temporal operators. There are 3 cases that must be considered: (a) $l = u = 0$, (b) $(l = 0)$ and $(u \geq 1)$, and (c) $u \geq l \geq 1$. For example, the fixpoint characterization for case (c) is

$$(p\, \mathcal{U}_{[l, u]}\, q) \Leftrightarrow [(\varepsilon = tick) \wedge p \wedge \bigcirc (p\, \mathcal{U}_{[l-1, u-1]}\, q)] \vee [\varepsilon \neq tick \wedge p \wedge \bigcirc (p\, \mathcal{U}_{[l, u]}\, q)] \qquad \text{(EQ 24)}$$

The TCHAIN rule may be derived from L-TRNS and induction on k.

Rule **TCHAIN** (timed chaining of until)

For state-formulas $p, p_0, p_1, \ldots, p_k$ let $p \stackrel{\text{def}}{=} p_0 \vee \ldots \vee p_k$, and let $p' \stackrel{\text{def}}{=} p_1 \vee \ldots \vee p_k$. Let $q_i \stackrel{\text{def}}{=} p_i \vee p_{i-1} \vee \ldots \vee p_0$, and for each i let d_i be an integer with $d \stackrel{\text{def}}{=} d_1 + \ldots + d_k$.

$$\frac{\forall i \,|\, 1 \leq i \leq k \cdot p_i \rightarrow (p_i\, \mathcal{U}_{\leq d_i}\, q_{i-1})}{p \Rightarrow (p'\, \mathcal{U}_{\leq d}\, p_0)}$$

The TCHAIN rule considers a ranked sequence of state formulas $p_k, p_{k-1}, \ldots, p_0$. The premise asserts that for each state formula p_i, we can prove that once p_i becomes true it remains true until within d_i ticks of the clock progress is made towards an element of the sequence with lower ranking. The conclusion is that from any p_i it is possible to reach the lowest ranked goal state p_0 in a time no less than the sum of the individual steps. The TCHAIN rule also implies the weaker consequence $p \Rightarrow \Diamond_{\leq d}\, p_0$.

3.4.1 The temporal semantics axiom and relative completeness

From a theoretical point of view there is a single axiom, denoted by $\mathcal{A}_M$, that is adequate for proving all the temporal properties of the TTM M. This axiom consists of several conjuncts corresponding to the requirements of legal trajectories.

1. The initial condition is represented by the axiom:

$$\models^M I_M \text{ and } \models^M (first \Rightarrow I_M) \qquad \text{(EQ 25)}$$

(The right conjunct $\models (first \Rightarrow I_M)$ is not strictly speaking necessary, but allows us to relate $first$ in specifications to the initial condition.)

2. For the succession requirement, we have:

for any TTM transition τ, $\models^M [(\varepsilon = \tau) \Rightarrow e_\tau] \wedge \{p^\tau\}\,\tau\,\{p\}$ for any state-formula p not containing any occurrences of the event variable ε. (EQ 26)

The left conjunct of (EQ 26) represents that part of the succession requirement that specifies that if a transition is taken it must be enabled. The right conjunct represents the fact that a (restricted) state must be related to its successor via the transformation function. This follows by the same type of reasoning used to justify the assignment axiom in Hoare logic.

3. The ticking requirement for legal trajectories may be written:

$$\models^M \Box\Diamond(\varepsilon = tick) \qquad \text{(EQ 27)}$$

4. The justice requirement for legal trajectories may be written as

$$\models^M \forall \tau \in \mathcal{J}_M \mid \Diamond\Box e_\tau \rightarrow \Box\Diamond(\varepsilon = \tau) \qquad \text{(EQ 28)}$$

5. The real-time requirement for legal trajectories may be written:

For any transition τ with lower and upper time bounds l, u respectively
$\models^M moe(\tau) \Rightarrow [\Box_{<l}(\varepsilon \neq \tau) \wedge (e_\tau \mathcal{U}_{\leq u} (\varepsilon = \tau \vee \neg e_\tau))]$, where
$moe(\tau) \stackrel{\text{def}}{\equiv} e_\tau \wedge [first \vee \ominus(\varepsilon = \tau \vee \neg e_\tau)]$. (EQ 29)

A single adequate axiom

The single assertion $\mathcal{A}_M$, that is adequate for proving all RTTL properties of a TTM M, can now be defined as

$$\mathcal{A}_M \stackrel{\text{def}}{\equiv} \text{(EQ 25) \& (EQ 26) \& (EQ 27) \& (EQ 28) \& (EQ 29)} \qquad \text{(EQ 30)}$$

where we can show by semantic justification that each of the five conjuncts ensures that one of the five requirements for the legal trajectories of M is satisfied. This means that $\mathcal{A}_M$ is *relatively complete* for proving M-validities, i.e. if we are equipped with an omniscient oracle that is guaranteed to provide confirmation of each valid temporal formula of the SR and TLR parts of the proof system, then instead of proving $\models^M p$ (for some specification p), we need only seek confirmation from the oracle for the general temporal logic validity of $\models \mathcal{A}_M \rightarrow p$.

Since each of the five conjuncts is M-valid, the axiom $\mathcal{A}_M$ is also M-valid, and we can therefore introduce $\mathcal{A}_M$ as an axiom of the proof system without compromising the *soundness* of the proof system (i.e. only M-validities can be proved). We therefore have the following:

Proposition 1 (Soundness and Completeness):
The proof system constructed from SR *(state reasoning),* TLR *(temporal logic reasoning) and the additional axiom $\mathcal{A}_M$ is sound and relatively complete for proving M-validities.*

While the use of $\mathcal{A}_M$ is theoretically adequate it is not very practical. We therefore introduce special rules into the MR reasoning part of the proof system that allow us to deduce those specifications commonly encountered in practice.

3.4.2 Rules for the MR part of the proof system

The additional rules discussed below have the advantage that their premises are state-validities rather than (general) temporal logic validities. Thus, familiar first order reasoning may be employed rather than general temporal reasoning, while preserving the succinctness of temporal logic as a specification language.

Since the special rules use mainly predicate reasoning, we can also provide a disciplined approach to proving the correctness of specifications using weakest preconditions. This methodology is amenable to semi-automation.

Each rule that is introduced for MR reasoning is either a *basic rule* (which must be semantically justified) or a *derived rule* (which can be deduced from the axioms and basic rules of the proof system).

One of the basic rules of the TTM part of the RTTL proof system is the INIT rule (below). The INIT rule allows us to establish the temporal validity of p given

Rule **INIT** (initiality)

For a TTM M with initial condition I, and state-formula p

$$\frac{\Vdash I \rightarrow p}{\models^{M} p}$$

the state validity of $I \rightarrow p$. The first state of any legal trajectory satisfies the initial condition (by the initiality requirement for legal trajectories), and hence by the premise of INIT the first state also satisfies the state-formula p. Hence p is M-valid.

The leads-to relation

For $\tau \in \mathcal{T}$, the *leads-to* relation was defined earlier as the triple $\{p\}\,\tau\,\{q\}$, which is an abbreviation for $(\varepsilon = \tau) \wedge p \Rightarrow \bigcirc q$. If all transitions of a TTM lead from p to q, then we have $\{p\}\,\mathcal{T}\,\{q\}$, with its obvious consequence, the STEP rule (below). The STEP rule together with axiom FX5 (see (EQ 19)), may be used

Rule **STEP** (single step)

For a TTM $\mathcal{M}$ with transition set $\mathcal{T}$, and state-formulas p, q

$$\frac{\{p\}\,\mathcal{T}\,\{q\}}{p \Rightarrow \bigcirc q}$$

to prove the rule INV, using the principle of computational induction based on the invariance p.

Rule **INV** (invariance)

For a TTM $\mathcal{M}$ with transition set $\mathcal{T}$, and state-formulas p, q

$$\frac{p \Rightarrow q \qquad \{p\}\,\mathcal{T}\,\{p\}}{p \Rightarrow \Box q}$$

The proof of INV from STEP is as follows:

1. $p \Rightarrow q$ — Premise
2. $\{p\}\,\mathcal{T}\,\{p\}$ — Premise
3. $p \Rightarrow \bigcirc p$ — STEP 2
4. $(p \Rightarrow \bigcirc p) \rightarrow (p \Rightarrow \Box p)$ — TL R (axiom FX5)
5. $(p \Rightarrow \Box p)$ — TLR (MP) 3,4
6. $(p \Rightarrow q) \rightarrow (\Box p \Rightarrow \Box q)$ — TLR (see (EQ 21))
7. $\Box p \Rightarrow \Box q$ — MP 1,6
8. $p \Rightarrow \Box q$ — TLR 5,7 (see derived rule E-TRNS)

Since all the axioms of RTTL deductive system are M-valid, and all the proof rules preserve M-validities, the derived proof rule INV also preserves M-validity. There are two ways to show validity when introducing a new rule: (a) through semantic justification, and (b) by using the RTTL deductive system to derive the new rule from already existing ones. In the above rules, the basic rules INIT and STEP were introduced using (trivial) semantic justifications, whereas INV was derived using the proof system and the basic rules.

The S-INV rule trivially follows from INIT and INV. The S-INV rule is a good

Rule **S-INV** (simple invariance)

For a TTM M with transition set $\mathcal{T}$, initial condition I, and state-formulas p, q

$$\frac{\models I \rightarrow p, \quad \models p \rightarrow q \quad \text{and} \quad \models \{p\}\,\mathcal{T}\,\{p\}}{\models^{M} \Box q}$$

illustration of the use of (implicit) computational induction on the invariance p to prove a safety property ($\Box q$). One chooses an invariant p at least as strong as q. Then show that p holds initially and is preserved by any transition of the TTM. Based on induction on the position of a computation, it follows that p (and hence q) hold on all positions.The rule M-INV may also be deduced from INIT and INV.

In contrast to safety properties, liveness properties such as $p \Rightarrow \Diamond q$, usually have several important intermediate stages that the computation usually must go through on the way from a p-state to a q-state. Thus a chain rule such as TCHAIN, or some well founded argument must be made to prove the property.

Rule **M-INV** (multiple invariance)

Let the TTM M have transition set $\mathcal{T}$ and initial condition I. Let $q, p, p_1, \ldots, p_n$ be state-formulas with $p \stackrel{\text{def}}{\equiv} p_1 \vee \ldots \vee p_n$.

$$\models I \rightarrow p$$

$$\forall i \,|\, 1 \le i \le n \cdot \models (p_i \rightarrow q)$$

$$\forall i \,|\, 1 \le i \le n \cdot \models \{p_i\}\, \mathcal{T}\, \{p\}$$

$$\models^M \Box q$$

Verification Conditions

To prove temporal properties using rules such as STEP and INV, we will need to prove the validity of the leads-to relation $\{p\}\,\tau\,\{q\}$ (for each transition τ). In the Manna-Pnueli deductive system, this is done by using a transition relation that relates the values of the variables in the current state with their values in the next state. The disadvantage of this approach is that an auxiliary requirement must be asserted that those variables not referenced in the transformation function of the transition are left unmodified.

In this paper, we use the notion of a verification condition similar to that of the Hoare assignment axiom for sequential programs. The rule VC (see below) is added to the TTM part of the RTTL deductive system. VC allows us to establish the leads-to relation by establishing the state validity of the *verification condition* $(p \wedge e_\tau \rightarrow q^\tau)$ for each transition τ in the transition set. If $h = [v_1{:}a_1 \ldots v_n{:}a_n]$ is the

Rule **VC** (verification condition)

For a TTM M and a transition set $\mathcal{T}_1 \subseteq \mathcal{T}_M$, and state-formulas p, q and q contains no occurrences of the event variable ε

$$\models \forall \tau \in \mathcal{T}_1 \,|\, p \wedge e_\tau \rightarrow q^\tau$$

$$\models^M \{p\}\, \mathcal{T}_1\, \{q\}$$

transformation function in assignment format of transition τ, then for any state-formula q

$$q^\tau \stackrel{\text{def}}{\equiv} q^{v_1 \ldots v_n}_{a_1 \ldots a_n} \qquad \text{(EQ 31)}$$

i.e. q^τ is the formula obtained from q by simultaneously replacing, for each i, all (free) occurrences of v_i by the expression a_i.

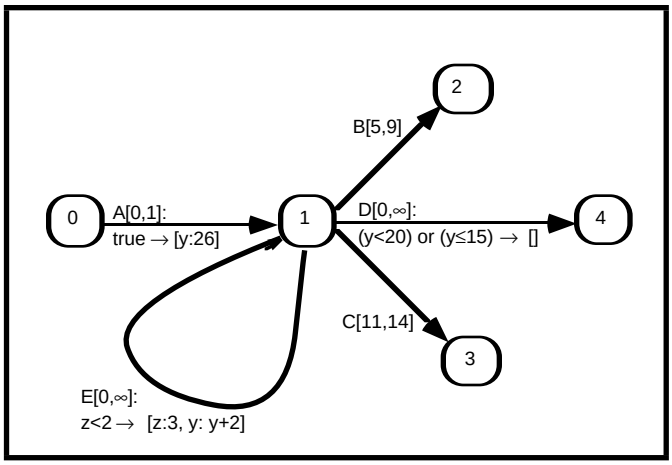
Using the verification conditions

Consider the infinite state TTMchart *sample* in Fig. 4 with activity variable x, and with initial condition

$$I_{sample} = (x = 0) \wedge (z < 2) \wedge (y \ge 25)$$

$$type(x) = \{0, 1, 2, 3, 4\}$$

FIGURE 4. The TTMchart *sample*



Suppose we want to prove that the specification $\Box(x \neq 4)$ is *sample*-valid, i.e. the TTM never enters the activity 4. In rule S-INV set $p \stackrel{\text{def}}{=} (y \geq 25) \wedge (x \neq 4)$, and $q \stackrel{\text{def}}{=} (x \neq 4)$. The first two premises of S-INV may be immediately derived (see lines 3 and 4 below) as follows:

1. $\models I_{sample} \rightarrow (y \geq 25) \wedge (x \neq 4)$ SR
2. $\models (y \geq 25) \wedge (x \neq 4) \rightarrow (x \neq 4)$ SR
3. $\models I_{sample} \rightarrow p$ 1 and the definition of p
4. $\models (p \rightarrow q)$ 2 and the definition of p, q

The third premise $\models \{p\}\, \mathcal{T}_{sample}\, \{p\}$ of S-INV must now be proved, i.e. we must show that each transition τ of the TTM *sample* leads from p to p. We may use the verification conditions and rule VC to obtain the desired result.

The verification condition to show that $\{p\}\, A\, \{p\}$ is given by $p \wedge e_A \rightarrow p^A$. Since $h_A = [x{:}1, y{:}26]$, we have that

$$\begin{aligned} p^A \quad & \stackrel{\text{def}}{=} \quad p^{x,\,y}_{1,\,26} \\ & \text{iff} \quad (y \geq 25 \wedge x \neq 4)^{x,\,y}_{1,\,26} \\ & \text{iff} \quad (26 \geq 25) \wedge (1 \neq 4) \\ & \text{iff} \quad \mathit{true} \end{aligned}$$

The verification condition is therefore state-valid irrespective of the antecedent $p \wedge e_A$, and we have

5. $\models (p \wedge e_A \rightarrow p^A)$ SR
6. $\models \{p\}\, A\, \{p\}$ VC 5

Similarly, for transitions B, C, D and E we can obtain

7. $\models \{p\}\, B\, \{p\}$ SR+VC
8. $\models \{p\}\, C\, \{p\}$ SR+VC
9. $\models \{p\}\, D\, \{p\}$ SR+VC
10. $\models \{p\}\, E\, \{p\}$ SR+VC

For the transition E, we have that

$$\begin{array}{lll} p^E & \stackrel{\text{def}}{=} & p^{x,\,y}_{1,\,y+2} \\ & \text{iff} & (y \geq 25 \wedge x \neq 4)^{\,x,\,y}_{1,\,y+2} \\ & \text{iff} & (y+2 \geq 25) \wedge (1 \neq 4) \\ & \text{iff} & y \geq 23 \end{array}$$

and hence $p \wedge e_E \rightarrow p^E$ is state-valid because $p \rightarrow (y \geq 23)$.

For transition D, the antecedent of the verification condition is false

$$\begin{array}{lll} p \wedge e_D & \stackrel{\text{def}}{=} & (y \geq 25 \wedge x \neq 4) \wedge [\,(x = 1) \wedge (y < 20 \vee y \leq 15)\,] \\ & \text{iff} & \mathit{false} \end{array}$$

i.e. D is disabled from occurring in any state satisfying p. Since the antecedent is false the verification condition $p \wedge e_D \rightarrow p^D$ is valid! Recall that the verification condition indicates only partial correctness, i.e. if D is taken from a state p it leads back to p. Since D is never taken from p partial correctness is preserved. We thus have

11. $\models \{p\}\, \mathcal{T}_{sample}\, \{p\}$ — TLR 6,7,8,9,10
12. $\models \Box(x \neq 4)$ — S-INV 3,4,11 and the definition of p

as required.

Summarizing the proof

The above proof of the specification $\Box(x \neq 4)$ was presented in great detail only to familiarize the reader with the notation and conventions of the RTTL proof system. In future, only the general outline need be written, leaving it up to the reader to check the details. The proof outline for the property $\Box(x \neq 4)$, using the invariance $p \stackrel{\text{def}}{=} (y \geq 25) \wedge (x \neq 4)$, is written as follows:

13. $\Vdash I_{sample} \rightarrow p$ — SR
14. $\Vdash p \rightarrow (x \neq 4)$ — SR
15. $\Vdash \forall \tau \in T_{sample} \,|\, p \wedge e_\tau \rightarrow p^\tau$ — SR
16. $\{p\}\, \mathcal{T}_{sample}\, \{p\}$ — VC 15
17. $\Box(x \neq 4)$ — S-INV 13,14,16

The M-validity symbol is usually left out (as in steps 16 and 17 of the summarized proof). The state-validity symbol (as in proof steps 13, 14 and 15) is inserted in order to distinguish between those steps that are derived using state reasoning (SR) and those that are derived using temporal logic reasoning temporal logic reasoning (TLR and MR).

Most of the reasoning in the proof of the temporal logic property $\Box(x \neq 4)$ involved the use of ordinary propositional, predicate and domain reasoning to prove the verification conditions which are state-validities (hence the use of SR in proof steps 6–10). Very little actual temporal reasoning was needed. This is the main reason for distinguishing between state and temporal validity. Automating the proof of properties such as $\Box(x \neq 4)$ to a large part means finding the right tool for doing ordinary predicate logic in appropriate domains (integers, lists and sets).

The state reasoning part including the check on the verification conditions involves a large amount of detailed (but trivial) reasoning. In Section 6.0, we describe how the StateTime tool automates this kind of reasoning.

VC and INV are not sufficient for proving all invariances of timed systems

Although rules like VC (together with INV) can be shown to be complete for proving invariances in the Manna-Pnueli proof system, this is not the case when it comes to the timed behaviour of TTMs.

Consider the specification $\Box(x \neq 3 \wedge x \neq 4)$. This specification is obviously *sample*-valid because the upper time bound of B is less than the lower time bound of C. Thus B must always occur before C becomes eligible.

If we employ the same type of reasoning that was used previously (for the proof of the weaker specification $\Box(x \neq 4)$), the proof will break will break down at the verification condition for C. Setting $p \triangleq (y \geq 25) \wedge (x \neq 3) \wedge (x \neq 4)$ we obtain

$$\begin{array}{lll} p^C & \stackrel{\text{def}}{=} & p_3^x \\ & \text{iff} & (y \geq 25 \wedge x \neq 3 \wedge x \neq 4)_3^x \\ & \text{iff} & \mathit{false} \end{array}$$

The antecedent of the verification condition is $(x = 1) \wedge (y \geq 25)$ (which is true in some reachable states). Hence the verification condition for C is invalid.

The VC rule is based on the succession requirement of legal trajectories. The rule does not take into account the real-time requirement of legal trajectories. From the point of view of the succession requirement, it is possible for C to be taken and hence for the specification to be falsified. It is the real-time requirement that prevents C from occurring.

An additional rule DTB, that takes into account timing information, will be introduced in Section 3.4.3, from which the validity of $\{p\}\, C\, \{p\}$ can be deduced.

18. $\{p\}\, C\, \{p\}$ DTB

Taking the transition C preserves the invariance p (because in point of fact the transition C can never be taken). The proof of the property $\Box(x \neq 3 \wedge x \neq 4)$ can now be undertaken in a fashion similar to that of the proof of the weaker property $\Box(x \neq 4)$, with the exception that step 18 above uses DTB for proving the leads-to relation for C instead of VC.

The above methods of proof requires a fair amount of guesswork. For example, to use the S-INV rule, the "loop" invariant $p \stackrel{\text{def}}{=} (y \geq 25) \wedge (x \neq 3 \wedge x \neq 4)$ must be discovered. The choice of the right conjunct of p is obvious, as it is just the specification that is to be proved. The choice of the left conjunct $(y \geq 25)$ is more difficult to obtain in the general case, as it involves already knowing something of the behaviour of the TTM.

A large amount of detailed but trivial reasoning and guesswork was required just to prove an invariance property of a simple system. However, a significant part of the analysis is amenable to computer aided verification. The heuristics presented in Section 4.0 will help guide proofs with the use of proof diagrams. In

Section 6.0, tools for automating the heuristics and completely automated model-checking are discussed.

The rule S-INV was used above in conjunction with DTB to isolate the real-time part of the reasoning. Thus, ordinary temporal reasoning (and most of that using SR) was used for the most part. The heuristics will work in a similar fashion, only requiring the use of real-time reasoning where necessary.

3.4.3 Some additional proof rules

The rule DTB below uses transition time bounds to deduce that an eligible

Rule DTB — Disablement by time bounds

Let τ_l and τ_u be transitions of any TTM M with enabling conditions e_l and e_u respectively. Let $l > u$ where l is the lower finite time bound of τ_l and u is the upper time bound of τ_u. Let $moe(\tau_l) \stackrel{\text{def}}{\equiv} e_l \wedge [\, first \vee \ominus(\varepsilon = \tau_l \vee \neg e_l)\,]$. Then for any state-formulas p, q

H1. $l > u$

H2. $\Vdash (p \wedge q) \leftrightarrow false$

H3. $\vDash \{\neg p \wedge \neg q\}\ \mathcal{T}_M\ \{\neg q\}$

H4. $\vDash p \Rightarrow \bigcirc [moe(\tau_l) \wedge q]$

H5. $\Vdash (q \rightarrow e_u)$

H6. $\vDash \{q\}\ \tau_u\ \{\neg q\}$

$\vDash \neg q \Rightarrow \Box [q \rightarrow (\varepsilon \neq \tau_l)]$

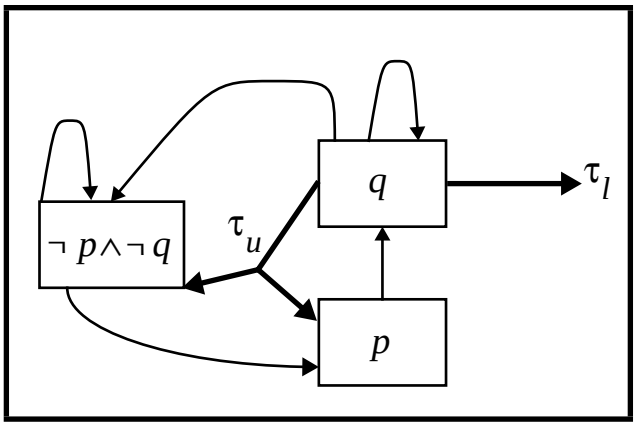
transition τ_l will not occur from a state satisfying q because there is some other transition τ_u that will always occur first. The transition τ_u can be thought of as a "progress edge" that guarantees that the TTM will exit any state satisfying q before τ_l can be taken. All the hypotheses of the DTB rule are established by state reasoning either directly or indirectly via the verification conditions.

The intuition behind the rule is displayed graphically in the form of a *proof diagram* in Fig. 5. A box with label p represents all the states satisfying the state-formula p. The edges leaving a box (also called a node) indicate all the possible transitions which the TTM can take while in the box. If the edge τ_u is taken from a state satisfying q, then it leads either to a state satisfying $\neg p \wedge \neg q$ or to p. An unlabelled edge indicates that zero or more transitions may satisfy the leads-to relation. Any edge τ leading from a node r to a node w must satisfy $\vDash \{r\}\ \tau\ \{w\}$.

Since hypothesis H2 of the DTB rule requires that $p \wedge q \leftrightarrow false$, the three boxes do not intersect with or subsume each other (although it is possible in general proof diagrams for one box to be a subset or to intersect with a second box[3]). Hence we also have that $p = p \wedge \neg q$.

3. If $p \rightarrow q$, then the box p is a subset of q. If $(p \wedge q \neq false)$, then the two boxes intersect.

FIGURE 5. Proof Diagram for the rule DTT



Proof of the soundness of DTB: An outline of the proof of the soundness of DTT is as follows. Assume that the j-th position of any legal trajectory σ satisfies $\neg q$, i.e. $(\sigma, j) \models \neg q$. We must show that $\forall k \geq j | (\sigma, k) \models (q \rightarrow (\varepsilon \neq \tau_l))$. From the assumption we are either in p or in $\neg p \wedge \neg q$ at position j. The third premise H3 of DTB asserts that the only way to get from $\neg p \wedge \neg q$ to q is via p. Thus we need only consider the case in which at some position no earlier than j we reach p. Hypothesis H4 guarantees that a p-state must be followed by a q-state that is a moment of enablement for the transition τ_l, i.e. the lower time bound requirement applies constraining τ_l not to occur for l ticks of the clock. H5 asserts that the edge τ_u must be enabled everywhere in q. Since τ_u must satisfy its upper time bound requirement, it is a progress edge that must occur by u ticks of the clock, or be disabled. But by H5 it cannot be disabled while in a q-state. The last premise asserts that when τ_u is taken, it "exits" from q, i.e. it leads to somewhere outside of q. By H1, $l > u$, and hence τ_l cannot be taken before τ_u. Thus there is no position subsequent to j in which q holds and where τ_l can be taken. (**end of proof**).

The antecedent $\neg q$ is needed in the conclusion of DTB, because the moment of enablement p is needed to start the count for the lower time bound. If q is reached some time after τ_l becomes enabled, it may be possible for τ_l to be taken prior to τ_u.

It was claimed earlier (see Step 18. on page 36) that the rule DTB could be used to obtain $\{y \geq 25 \wedge x \neq 3 \wedge x \neq 4\}\, C\, \{y \geq 25 \wedge x \neq 3 \wedge x \neq 4\}$ for the TTM "sample" of Fig. 4. In DTB, set

$$p \stackrel{\text{def}}{\equiv} (\varepsilon = A)$$
$$q \stackrel{\text{def}}{\equiv} (x = 1)$$
$$\tau_u \stackrel{\text{def}}{\equiv} B$$
$$\tau_l \stackrel{\text{def}}{\equiv} C$$

The first hypothesis of DTB is trivially true as $11 > 9$. By the MR succession axiom (EQ 26) it follows that $(\varepsilon = A) \Rightarrow (x = 0) \wedge \bigcirc (x = 1)$. Thus the second hypothesis of DTB holds. The hypothesis H3 of DTB is obtained as follows. Let $\mathcal{T}' = \mathcal{T} - \{A\}$. Then

1. $\models \forall \tau \in \mathcal{T}' | (\neg q \wedge e_\tau) \rightarrow (\neg q)^\tau$ SR

2. $\varepsilon \in \mathcal{T}' \wedge \neg q \Rightarrow \bigcirc \neg q$ VC 1 and rewriting $\{\neg q\}\, \mathcal{T}'\, \{\neg q\}$
3. $(\varepsilon = A \wedge \neg p) \Rightarrow \bigcirc \neg q$ TLR $((\varepsilon = A \wedge \neg p) \equiv false)$
4. $\{\neg p \wedge \neg q\}\, \mathcal{T}_{sample}\, \{\neg q\}$ TLR 2, 3

Using the MR succession axiom (EQ 26) we can easily prove $p \Rightarrow \neg e_C \wedge \bigcirc (e_C \wedge q)$ which trivially entails H4. The hypothesis H5 of DTB is also trivially valid. The last hypothesis is obtained using the verification condition for transition B, i.e. since

$$(\neg q)^B \stackrel{\text{def}}{\equiv} \neg (x = 1)^x_2 \equiv true$$

we have

5. $\models (q \wedge e_B) \rightarrow (\neg q)^B$ SR
6. $\{q\}\, B\, \{\neg q\}$ VC 5

Since all the hypotheses of DTB are valid, it therefore follows that

7. $\neg q \Rightarrow \Box (q \rightarrow (\varepsilon \neq C))$ DTB
8. $(I_{sample} \Rightarrow \neg q)$ SR+GEN
9. $(\varepsilon = C) \Rightarrow \neg q$ TLR 7,8

But (EQ 26) applied to C yields $(\varepsilon = C) \Rightarrow q$. Hence for any invariant r for the TTM of Fig. 4 it follows that $\{r\}\, C\, \{r\}$, because in point of fact $\Box (\varepsilon \neq C)$. We have thus validated Step 18. on page 36 required for the proof of the stronger invariance property.

In the RR rule (real-time response) below, the set of transitions of a TTM is divided into three disjoint sets: those transitions that are selfloops (i.e. do not exit the state formula p), those that do exit p (these transitions have some at least one successor state outside of p), and the singleton set consisting of the transition τ (called the progress transition). The progress transition τ is enabled in all states satisfying p, and has a finite upper time bound u. Thus by the upper bound requirement of legal trajectories, the state-formula q must hold within u ticks, where q asserts that either τ is taken or one of the exiting transitions is taken. The RR rule is given by:

Rule RR—Real-Time Response

Let the set of transitions $\mathcal{T}$ of a TTM be divided into the disjoint union $\{\tau\} \cup \{\mathcal{T}_{selfloop}\} \cup \{\mathcal{T}_{exiting}\}$. The progress edge τ has a finite upper time bound u. Then for any state-formula p:

H1. $q \stackrel{\text{def}}{\equiv} (\varepsilon = \tau \vee \varepsilon \in \mathcal{T}_{exiting}) \wedge p$

H2. u is finite

H3. $\models (p \rightarrow e_\tau)$

H4. $\models \{p\}\, \mathcal{T}_{selfloop}\, \{p\}$

Con. $\models p \Rightarrow (p\, \mathcal{U}_{\leq u}\, q)$

The RR rule also implies the weaker consequence $p \Rightarrow \Diamond_{\leq u} q$.

4.0 Proof heuristics (pragmatics)

A proof system on its own is not sufficient to make a framework useful. There must be a methodology for using the proof system systematically (the "pragmatics" of the proof system). Furthermore, it is beneficial if parts of the methodology can be automated.

We provide heuristics for proving invariances and real-time response properties. Many other properties can be reduced to these properties with the help of an observer or watchdog. Many properties that are commonly encountered by the verifier fall into the class treated by the heuristics. These heuristics can be semi-automated so as to take care of the tedious but simple verification conditions that must be checked.

The heuristics (a) help to guide the proof of some important properties, (b) help with the choice of invariant, and (c) decouple the real-time reasoning (MR) from the rest of the reasoning (SR+TLR). A further feature of the heuristics is that they involve relatively little temporal logic reasoning; most of the reasoning is state reasoning (propositional and predicate logic based on the data types of the system variables). This makes it possible to use off-the-shelf theorem provers to do most of the reasoning.

The heuristics use the notion of a proof diagram (e.g. see Fig. 5) to visually display proofs. Proof diagrams can be seen as a high-level view of the TTM reachability graph. The nodes of the proof diagram are state-formulas rather than states. A node of a proof diagram can represent an infinite number of states. It is therefore possible for the proof diagram to be finite while representing the behaviour of an infinite state system.

We first define strongest postconditions and weakest preconditions for TTMs analogous to those of the Dijkstra calculus [10,11] for sequential systems. We then present and illustrate the use of the heuristics.

Definition [psp — partial strongest postcondition]
Let τ *be a transition with transformation function* $h = [v_1{:}a_1, \ldots, v_k{:}a_k]$. *The* psp *of* τ leading from *a precondition* p *(which is a state-formula) is*

$$psp(\tau, p) = \exists V_1 \ldots \exists V_k \,|\, p^{v_1, \ldots, v_k}_{V_1, \ldots, V_k} \wedge (v_1 = (a_1)^{v_1, \ldots, v_k}_{V_1, \ldots, V_k}) \wedge \ldots \wedge (v_k = (a_k)^{v_1, \ldots, v_k}_{V_1, \ldots, V_k})$$

As an example, consider a transition τ with $h_\tau = [x{:}x+1]$. We then have that

$$\begin{aligned} psp(\tau, x = 7) &= \exists X \,|\, (x = 7)^{x}_{X} \wedge (x = (x+1)^{x}_{X}) \\ &= \exists X \,|\, (X = 7) \wedge (x = X + 1) \\ &= (x = 8) \end{aligned}$$

There is no guarantee that τ will be taken from state-formula p as we are dealing with concurrent nondeterministic systems (some other transition may equally well be eligible). Hence there is no total correctness implied but only *partial* correctness. A direct consequence of the definition is that $\models \{p\}\,\tau\,\{psp(\tau, p)\}$.

We refer the reader to [28] for the *psp-heuristic* for invariance properties such as $\Box q$. The initial node of the proof diagram is the initial condition I (or a weaker

state-formula) of the TTM. All successor nodes p are computed using $p = psp(\tau, I)$ for each transition τ that exits from I. A transition τ *exits from the state-formula* I if there is at least one I-state s in which τ is enabled and when taken leads to a state not in I, i.e. $[s \models I \wedge e_\tau \wedge \neg I^\tau] = true$.

Each of the successor nodes are then explored in turn for exiting transitions until all paths lead back to already computed nodes. If all nodes p in the diagram satisfy $\Vdash (p \rightarrow q)$, then the invariance $\Box q$ is valid (this is guaranteed by the M-INV rule).

Two strategies are used to reduce the size of the proof diagram. (a) Any node can be replaced by a weaker formula r provided $\Vdash (r \rightarrow q)$. By weakening a node the number of exiting transitions can often be reduced. (b) The rule DTB can be used to eliminate an edge from one node to another despite the fact that the corresponding transition is eligible to be taken.

As shown in [28], the psp-heuristic may be helpful in the construction of invariants. However, in practice, the psp-heuristic is most helpful in conjunction with the pwp-heuristic described below.

Definition [pwp — partial weakest precondition]
The pwp of the transition τ *with respect to a postcondition* q *(which is a state-formula) is* $pwp(\tau, q) = [e_t \wedge q^\tau]$.

As an example, consider the same transition τ as above with $h_\tau = [x{:}x+1]$ and $e_\tau \stackrel{\text{def}}{=} (x \leq 6)$. We then have that

$$\begin{aligned} pwp(\tau, x \geq 7) &= (x \leq 6) \wedge (x \geq 7)^{\,x}_{x+1} \\ &= (x \leq 6) \wedge (x + 1 \geq 7) \\ &= (x \leq 6) \wedge (x \geq 6) \\ &= (x = 6) \end{aligned}$$

A direct consequence of the definition is that $\models \{pwp(\tau, q)\}\,\tau\,\{q\}$.

4.1 Invariant heuristic

To prove the invariance property $\Box q$ we note that $\Box q \equiv \neg\Diamond(\neg q)$. Set $bad = \neg q$ and $good = \neg bad$. Then check if there is a way to get to the "bad" part of the state from some state-formula p (in the "good" part of the state) in one step. If not, then the invariance is valid, i.e. there is no path from the initial condition to the goal node bad. If there is such a node p, then work backwards from it in turn, and so on until the initial condition is reached. The backward node p is computed using the *pwp*, i.e.

$$p = pwp(\tau, bad) \wedge e_\tau \wedge good. \qquad \text{(EQ 32)}$$

If $p = false$, then there is no way to get to the bad part of the state in one step and no further exploration backward from p is required. If $p \neq false$ (i.e. there are some states that satisfy p), we also require that

$$\Vdash (p \wedge I) \rightarrow false \qquad \text{(EQ 33)}$$

(i.e. no p-state is an initial state).

Let all nodes on all backward paths ultimately backtrack to $false$, or cycle back to an already computed node. Then, since no node intersects with the initial condition, there is no way to get to the bad part of the state from the good, i.e. $\neg\Diamond bad$ is valid, and thus $\models \Box q$.

Only state reasoning SR is required in the heuristic to check (EQ 32) and (EQ 33). We may again use the two strategies, mentioned earlier with regard to the invariant-heuristic, to reduce the size of the proof diagram.

Example using the invariant-heuristic

The real-time mutual exclusion protocol shown below (due to M. Fischer) uses time bounds on its actions to ensure conformance to the protocol. The kernel of each process $fish_i$ (where $i \in \{1, 2\}$) executes the following code:

```
0: await <Y=0>                {transition a_i}
1: <Y := i>                   {transition b_i}
2: await <Y=i>                {transition c_i}
3: critical section           {transition c_i}
```

where the angle brackets denote atomic actions. Initially the shared variable `Y` is set to zero. We may assume that

$$l_{\tau_1} = l_{\tau_2}, u_{\tau_1} = u_{\tau_2} \qquad \text{(EQ 34)}$$

for each $\tau \in$ `{a, b, c, d}` (e.g. the bounds on `a_1` are the same as for `a_2)`. There are no other constraints on the bounds.

To actually use the above kernel, `c_i` would have to be modified to

```
2: if <Y # i> then goto 0
```

and the modified kernel would have to appear in the following loop:

```
begin loop non-critical section;
           kernel;
           <Y := 0>
end loop
```

The TTM $fish = fish_1 \| fish_2$ with activity variables X_1, X_2 respectively are shown in Fig. 6 using the BUILD tool. The BUILD tool will be discussed later in Section 6.0. The invariant heuristic will be used to deduce the mutual exclusion requirement R given by:

$$R: \quad \Box\neg(X_1 = 3 \wedge X_2 = 3).$$

Furthermore, we must *derive* the constraints on the lower and upper time bounds to ensure mutual exclusion of the critical sections. We assume that bounds can be imposed on `c_i` and `b_i`, but not on `a_i` and `d_i` (as these are determined by when each process requests entry into the critical section, or release the critical resource).

The proof diagram using the invariant-heuristic is shown in Fig. 7. There are only two transitions that can get to Bad in one step: `c_1` and `c_2`. We explore `c_1`. The proof diagram for the `c_2` path is symmetric to that of `c_1`.

The node $SF1$ is computed as

FIGURE 6. BUILD display of TTM $fish = fish_1 \parallel fish_2$

The dotted lines around `fish1` and `fish2` respectively indicate that these two TTMs are AND-composed with each other. The events of each TTM such as `a`, `b`, `c`, etc. are declared local and hence do not synchronize with each. BUILD refers to these events as: `a_1`, `a_2`, `b_1`, `b_2`, etc. Each TTM $fish_i$ has activity variable X_i which ranges over its type $type(X_i) = \{0, 1, 2, 3\}$. The shared data variable Y has $type(Y) = \{1, 2\}$.

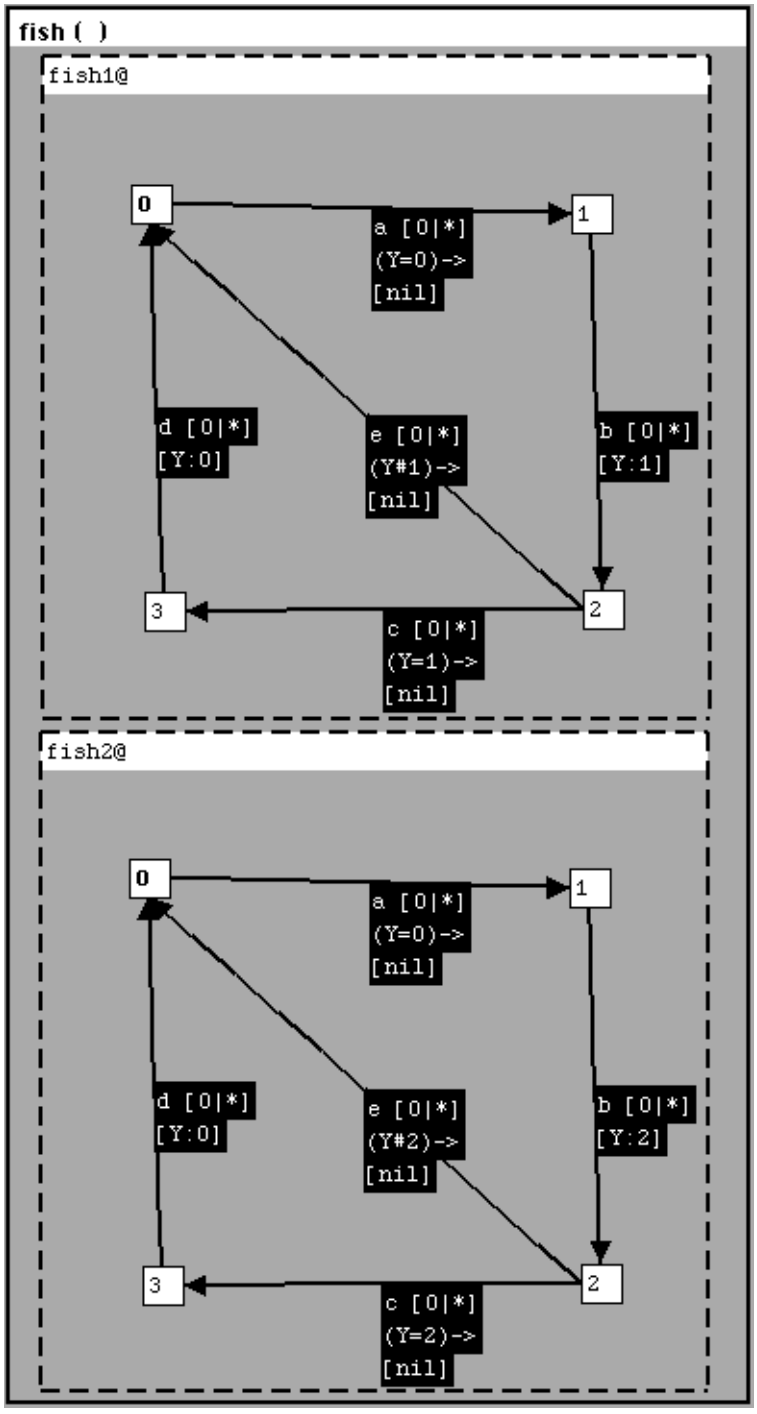
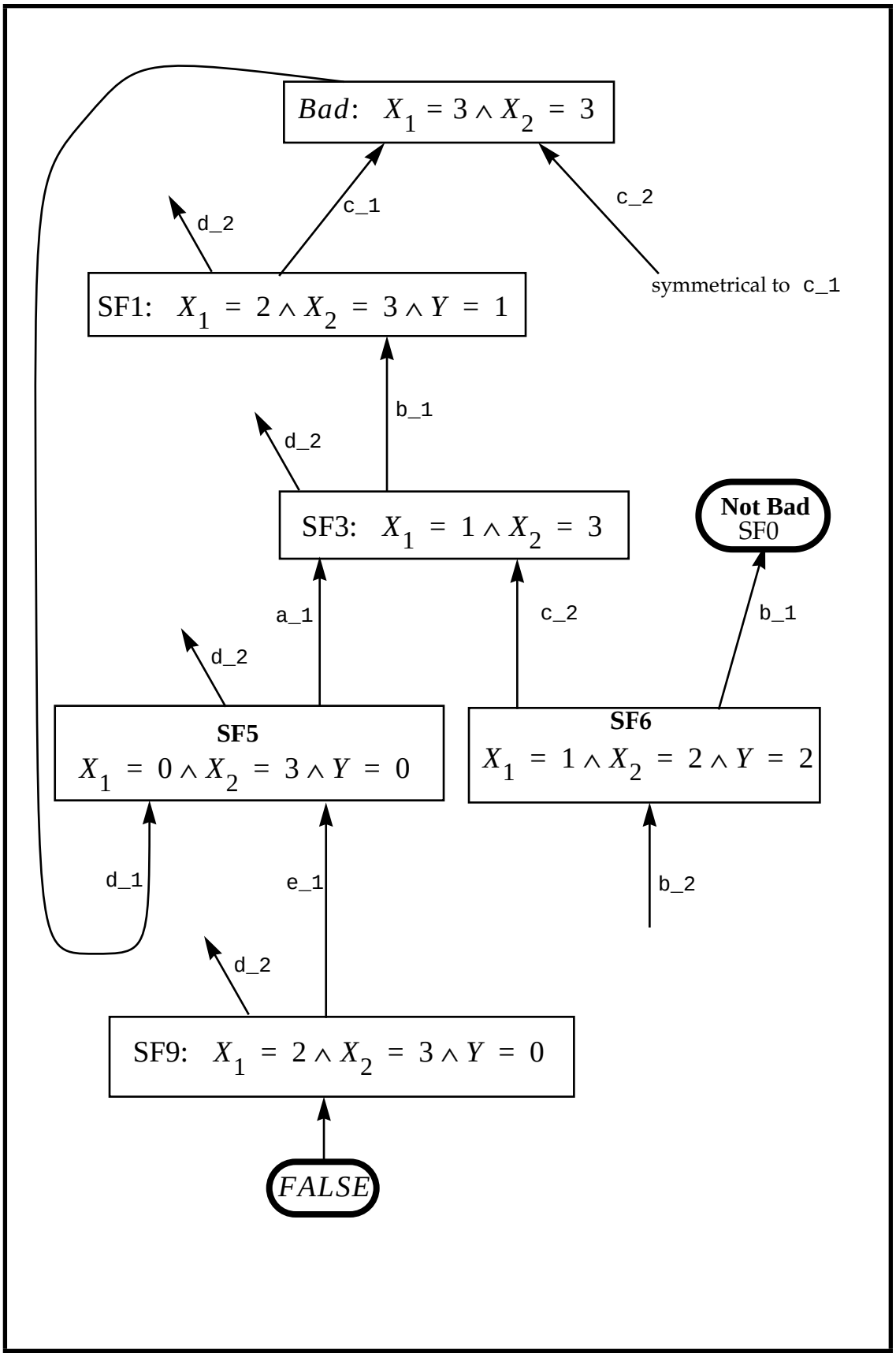


FIGURE 7. Proof diagram for mutual exclusion property of $fischer = process_1 \| process_2$



$$
\begin{aligned}
SF1 &= pwp(\mathtt{c\_1}, Bad) \wedge e_{\mathtt{c\_1}} \wedge Good \\
&= (X_1 = 3 \wedge X_2 = 3)^{X_1}_{3} \wedge (X_1 = 2 \wedge Y = 1) \wedge Good \\
&= (X_2 = 3 \wedge X_1 = 2 \wedge Y = 1)
\end{aligned}
$$

All transitions other than `c_i` cannot get from the good part of the state to Bad in one step. For example, consider the *pwp* computation for the transition `b_1`:

$$
\begin{aligned}
pwp(\mathtt{b\_1}, Bad) \wedge e_{\mathtt{b\_1}} &= (X_1 = 3 \wedge X_2 = 3)^{X_1, Y}_{2,1} \wedge (X_1 = 1) \\
&= (2 = 3 \wedge X_2 = 3 \wedge X_1 = 1) \\
&= false
\end{aligned}
$$

At node $SF6$ in the proof diagram, the rule DTB is used to ensure that the transition `c_2` is never taken from $SF6$. This is the first node on the backward path which has both `c` and `b` as exiting transition, and is hence a candidate for the DTB rule.

The transition `b_1` is a possible progress edge with respect to the node $SF6$ (i.e. $SF6 \rightarrow e_{\mathtt{b\_1}}$). Therefore, in the DTB proof rule we set

$$
\begin{aligned}
p &= (\varepsilon = \mathtt{b\_2} \wedge X_1 = 1) \\
q &= SF6 \\
\tau_u &= \mathtt{b\_1} \\
\tau_l &= \mathtt{c\_2}
\end{aligned}
$$

The progress edge `b_1` leads to a state-formula $SF0$ that can be computed using the *psp*, i.e.

$$
\begin{aligned}
SF0 &= psp(\mathtt{b\_1}, SF6) \\
&= (Y = 1 \wedge X_1 = 2 \wedge X_2 = 2)
\end{aligned}
$$

$SF0$ is not in the bad part of the state, which is now redefined as

$$
NewBad = Bad \vee SF1 \vee SF3 \vee SF5 \vee SF9
$$

because of its incompatible conjunct $X_2 = 2$.

All the hypotheses H2-H6 in DTB are satisfied. Thus the conclusion of DTB will follow provided that H1 is true. Therefore, if $l_{\mathtt{c\_2}} > u_{\mathtt{b\_1}}$, then `c_2` will never be taken from $SF6$, as required. By the bound constraints in (EQ 34), we therefore have that the mutual exclusion requirement R is valid provided that

$$
l_{\mathtt{c\_i}} > u_{\mathtt{b\_i}} \qquad \text{(EQ 35)}
$$

All the reasoning is state reasoning (SR). There is no temporal logic required. The mutual exclusion property is verified mechanically by directly following the rules of the heuristic. The bound constraints are deduced automatically as part of the proof. The real-time rule DTB was used only at the end where necessary thus decoupling the real-time component of the reasoning from the rest.

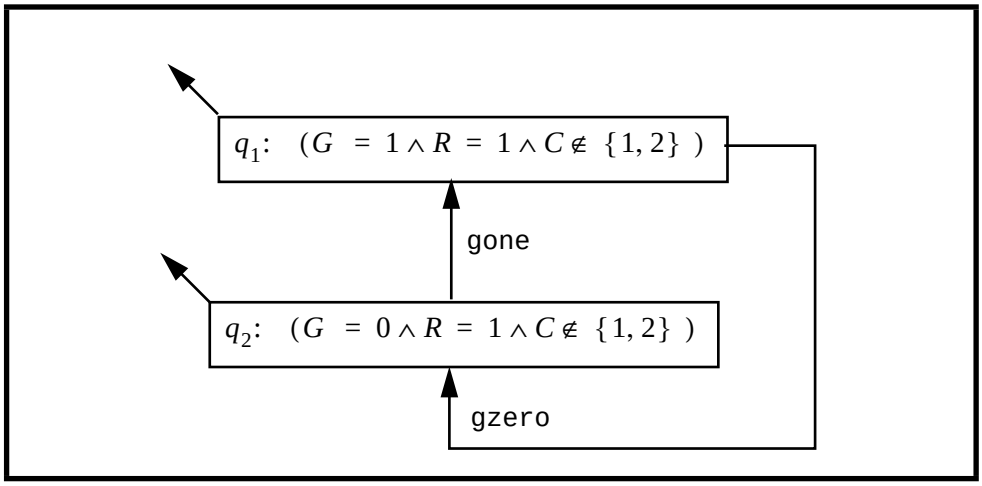
If the initial condition is reached on a backward path, then the invariant property is invalid. A legal trajectory from an initial state to the bad part of the state

along that path is the counterexample. The counterexample provides valuable diagnostic information for debugging the system.

In some situations it is possible to hit an infinite backward path that neither hits the initial condition (in which case the invariant is invalid), nor stops at *false*. In such a case, the abovementioned strategy of weakening the nodes of the proof diagram must be used.

When backtracking, it may happen that there is a cycle involving some nodes with paths that do not backtrack beyond the cycle itself. In such a case, the cycle of nodes is isolated from the initial states and is thus an unreachable part of the state. For example, consider the TTM $cae = client \| env$ (Fig. 9 and Table 4) which is part of a resource allocation system that will be discussed later. Let q_1, q_2 be the state-formulas shown in Fig. 8. The only way to get to q_1 is from q_2 and vice

FIGURE 8. An isolated unreachable part of the state space



versa. The initial condition of the TTM *cae* is $(C = 0 \wedge R = 0 \wedge G = 0)$, which is inconsistent with that part of the state space q, where $q = q_1 \vee q_2$. Hence it follows that $\models \Box\neg q$ because there is no path from an initial state to q. In particular, it follows that

$$\models \Box\neg(G = 1 \wedge R = 1 \wedge C \notin \{1, 2\}) \qquad \text{(EQ 36)}$$

4.2 Real-time response heuristic (and liveness)

The real-time response property $p \Rightarrow \Diamond_{[l, u]} r$ can be verified using a heuristic that builds on the invariant-heuristic. The liveness property $p \Rightarrow \Diamond r$ can be verified using the same principles, but without the need for special real-time considerations. The heuristic is easily modified so as to check the validity of the until property $p \Rightarrow (q \,\mathcal{U}_{[l, u]}\, r)$.

The *response-heuristic* also uses the backward path computation and weakest preconditions for checking invariants. The *response node* is r, and we must ensure that on every backward path, computed via weakest preconditions, that the *cause node* p is reached. At every intermediate node we must check the exiting transitions, and must ensure that these exiting nodes lead to intermediate nodes closer

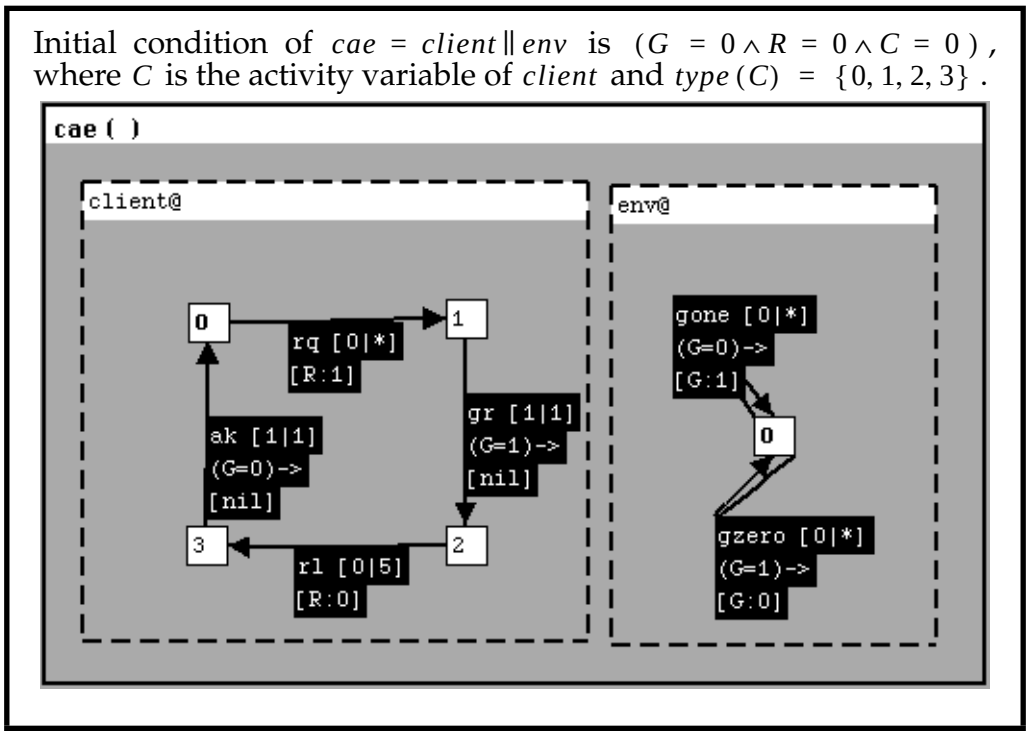
to the response node. Each node must have at least one progress edge that will move it closer to the response node. The RR rule is used to make the appropriate progress assertion. Finally, we must check that there are no cycles in the proof diagram, which would indicate that there are legal trajectories which are not guaranteed to lead to the goal. The T-TRNS rule can then be used to chain all the RR progress assertions together.

The upper bound u is computed by counting the maximal number of ticks on any path between the node p and the goal, and the lower bound l is computed by looking for the minimal number of ticks.

Unlike the invariant-heuristics where the nodes can be replaced with a weaker property, in the response-heuristic there is often a need to strengthen a node so as to obtain a progress edge, or to project out part of the state that is unreachable. These strategies of the heuristic are best illustrated using an example.

Consider the TTM $cae = client \| env$ (Fig. 9) which is used later to illustrate

FIGURE 9. Client TTM with environment



resource allocation. The *client* must function in an unconstrained environment *env* that can change the value of the grant variable G at any moment. It is clear that $\forall i | 3 \le i \ge 1 : \Box (G_i \in \{0, 1\} \wedge R_i \in \{0, 1\})$ (grant and request variables range over $\{0, 1\}$. The table of transitions is given in Table 4.

We must verify the response requirement given by

$$(G = 1) \Rightarrow \Diamond_{\le 6} (R = 0 \vee G = 0) \qquad \text{(EQ 37)}$$

for which $(R = 0 \vee G = 0)$ is the response node. Intuitively, it may appear that the response requirement is invalid. If the client is initially at $(C = 0)$, and the

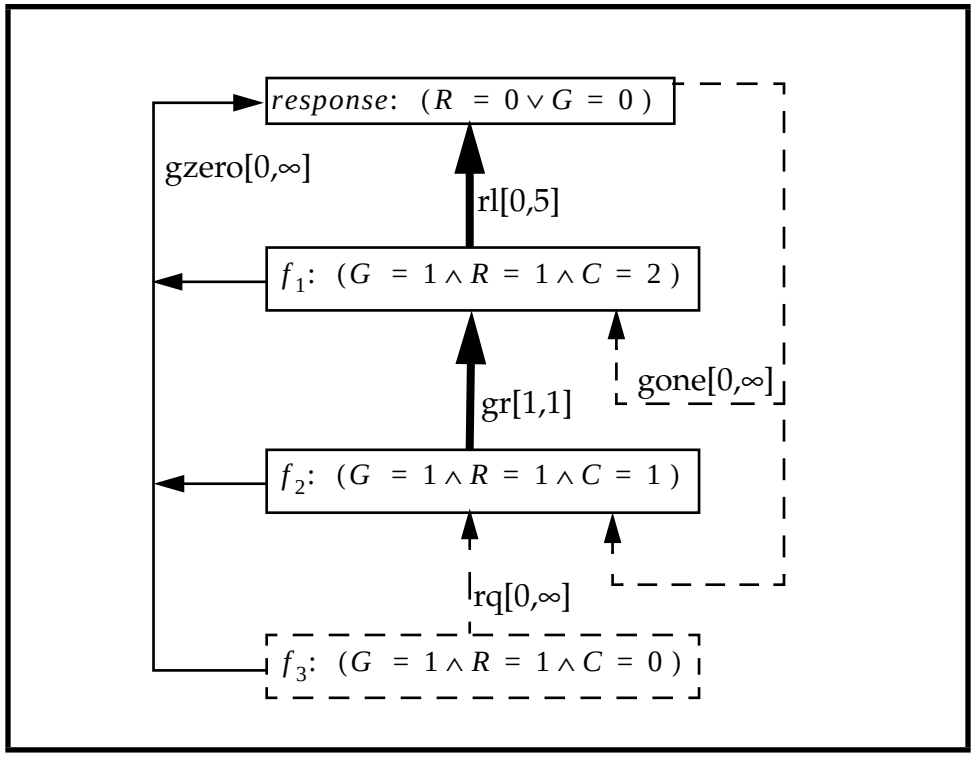
TABLE 4. Transitions of the client

transition name	enabling condition	transformation function	lower bound	upper bound
ak (acknowledge)	$(C = 3 \wedge G = 0)$	$[C{:}0]$	1	1
gr (grant)	$(C = 1 \wedge G = 1)$	$[C{:}2]$	1	1
rl (release)	$(C = 2)$	$[C{:}3, R{:}\ 0]$	0	5
rq (request)	$(C = 0)$	$[C{:}1, R{:}\ 1]$	1	1
gone (environment)	$(G = 0)$	$[G{:}1]$	0	∞
gzero (environment)	$(G = 1)$	$[G{:}0]$	0	∞

grant variable is set by *env* to one, there is no guarantee that eventually $(R = 0)$, because the event `rq[0,∞]` is spontaneous (and hence may never be taken). The argument for the correctness of (EQ 37) will therefore have to appeal to the fact that any state satisfying $(C = 0 \wedge R = 1)$ is not reachable (if $C = 0 \wedge R = 0$ then the response property is trivially true no matter what the value of G is).

The response heuristic will naturally guide us to the unreachability argument mentioned earlier.The proof diagram is provided in Fig. 10.

FIGURE 10. Proof diagram for a client



Working backwards from the *response* node, there are only two transitions that can get there in one step: `rl` and `gzero`. Using weakest preconditions, similar to (EQ 32), to compute the one step backwards node f_1 for `rl` we obtain

$$f_1 = pwp(\text{rl}, response) \wedge e_{\text{rl}} \wedge \neg response$$
$$= (R = 0 \vee G = 0)^{C,R}_{3,0} \wedge (C = 2) \wedge \neg(R = 0 \vee G = 0) \quad \text{(EQ 38)}$$
$$= (G = 1 \wedge R = 1 \wedge C = 2)$$

Performing a similar calculation for `gzero`, we obtain $f_4 = (G = 1 \wedge R = 1)$. However, the only backward path from f_4 is via rq. Hence f_4 may be strengthened to $f'_4 = f_4 \wedge (C = 1)$ which is the same as f_2, as shown in the proof diagram.

Next, we must check which transitions, in addition to `rl`, exit from f_1. Using (EQ 31), the only other exiting node is `gzero` as shown in the proof diagram. But using strongest postconditions, `gzero` leads to $response$. Although `gzero` is not guaranteed to occur, `rl[0,∞]` is a progress edge (i.e. $f_1 \rightarrow e_{rl}$ and its upper time bound is finite). Thus, from the RR rule we obtain

$$f_1 \Rightarrow \Diamond_{\leq 5}(response) \quad \text{(EQ 39)}$$

In the proof diagram bold arrows indicate progress edges. Similarly, we obtain $f_2 \Rightarrow \Diamond_{\leq 1}(response \vee f_1)$, and using the T-TRNS rule, we obtain

$$\vDash (f_1 \vee f_2) \Rightarrow \Diamond_{\leq 6}(response) \quad \text{(EQ 40)}$$

There is no need to explore the `gone` backward paths in the proof diagram, because these paths have their origin in states that already satisfy $response$.

At this point we have not yet verified the required response property. However, the node f_3 has no progress edge, and thus there is no way to weaken the antecedent of (EQ 40) so as to obtain the cause $(G = 1)$. This suggests that f_3 is in fact unreachable. We note that $f_1 \vee f_2 \equiv (G = 1 \wedge [R = 1 \wedge C \in \{1, 2\}])$. What we need is the cause, i.e. the conjunct $(G = 1)$. We may thus compute as follows:

$$(G = 1) \equiv (G = 1 \wedge ([R = 1 \wedge C \in \{1, 2\}] \vee \neg[R = 1 \wedge C \in \{1, 2\}])$$
$$\equiv (f_1 \vee f_2) \vee (G = 1 \wedge R = 0) \vee (G = 1 \wedge R = 1 \wedge C \notin \{1, 2\})$$

We know that the disjunct $f_1 \vee f_2$ satisfies the response requirement (EQ 40). The disjunct $(G = 1 \wedge R = 0)$ trivially satisfies the response requirement. In fact

$$\vDash (G = 1 \wedge R = 0) \Rightarrow \Diamond_{\leq 0}(response) \quad \text{(EQ 41)}$$

Our suspicion is therefore that the last disjunct is in an unreachable part of the space. The procedure for automatically checking such a hypothesis was provided under the discussion of invariant heuristics. The last disjunct is indeed unreachable as shown in (EQ 36). We thus have that

$$\vDash \Box[(G = 1) \equiv (f_1 \vee f_2) \vee (G = 1 \wedge R = 0)] \quad \text{(EQ 42)}$$

The required response property $(G = 1) \Rightarrow \Diamond_{\leq 6}(R = 0 \vee G = 0)$ is a direct consequence of (EQ 40), (EQ 41) and (EQ 42).

As a proof outline for the response property, we need only present the proof diagram Fig. 10, its conclusion (EQ 40), and the proof diagram Fig. 8 behind (EQ 42). The reader can then check the details mechanically.

Experience has shown that the procedure of (a) unwinding backwards using weakest preconditions, and (b) the elimination of part of the state space using an unreachability argument is the main idea behind many response and liveness proofs.

It is straightforward to check the until property $p \Rightarrow (q \,\mathcal{U}_{[l,u]}\, r)$. We need only check that $p \rightarrow q$ and that each node f_i of the proof diagram satisfies $f_1 \rightarrow q$.

5.0 Compositional Reasoning

The TTM/RTTL framework is based on the temporal logic for reactive systems in [21], with the necessary extensions and proof rules for timed transitions. This means that any valid temporal formula in [21] is also valid in RTTL. We may therefore use the theory of compositional reasoning presented in [21] within the TTM/RTTL framework as well. We describe in this section how compositional reasoning is done in RTTL.

Large systems are built from smaller parts. We want to reduce the verification of a complex low-level system, to proving properties of a higher-level specification of low-level components taken one at a time. Decomposing proofs in this way will also allow us to apply automated procedures (e.g. model-checking or heuristics) to components rather than the complete system. The Decomposition Theorem of Section 5.2 provides the needed proof rule, in which the *composition* of TTM modules is reduced to the logical *conjunction* of their RTTL specifications.

A module is fully specified by its *body* (a TTM), a well-defined *interface specification* and its r*eactive behaviour* (an RTTL formula in the parameters of the interface specification). In bottom-up development, the body is supplied and the module must be checked for modular validity. In top-down development, the interface specification and reactive behaviour are provided, and it is the task of the designer to develop a body that satisfies its reactive behaviour requirements.

To deduce useful properties of the module, we must specify how the module interacts with its (possibly hostile) environment. For example, a digital circuit is only guaranteed to operate if its input voltages are a clear 0 or 1, but may fail with an improper voltage level of 1/2. So too, a module specification will specify behaviour both with respect to a cooperative environment as well as a hostile environment.

5.1 Interface specifications

A *module* consists of an *interface specification* and a *body*. The purpose of the interface specification is to list all the shared variables (or channels if message passing is used) through which the module interacts with its environment. A variable declaration in the interface specification is preceded by one or more of the modes **in**, **out** or **external**.

Let y be a variable in the interface specification of module M_1. A statement in the body may have a reading reference to y only if y is declared to be of mode **in**, and a writing reference only if y is declared to be of mode **out**. A statement in a

module M_2 parallel to M_1 may have a writing reference to y only if y is declared in M_1 to be of mode **external**.

Consider the allocator module A in Fig. 11. The allocator will be discussed in more detail later. For now, we focus on its interface specification.

Since the array variable g (with mode **out**) is not declared as **external**, no other module (e.g a client) may change g — at best another module may read the value of g by declaring its mode as **external in**.

The body of a module may start with some internal (or local) variable declarations (e.g. see Fig. 11). These internal variables may not be referenced outside of its body.

Two concurrent modules are *interface compatible* if the declarations for any variable declared in both modules are consistent. The types specified in both declarations must be identical. The initial values assigned must be consistent. Finally, if one of the declarations specifies an **out** mode, the other specifies an **external** mode.

5.2 Modular validity and the decomposition theorem

We say that an RTTL formula p is *modularly valid* for a module M_1 if

$$[M_1 \| M] \models p \text{ for every module } M \text{ that is interface compatible with } M_1 . \qquad \text{(EQ 43)}$$

Thus, modular validity ensures that M_1 satisfies p independently of the behaviour of its environment, provided that its environment respects the constraints imposed by the interface specification. Usually, the formula p will refer only to the variables in the interface specification, and not to any of the local variables.

An immediate consequence of the definition in (EQ 43) is the following theorem

Proposition 2 (Decomposition Theorem):
Let RTTL formulas p_1, p_2 be modularly valid over compatible modules M_1, M_2 respectively. Then:

(a) $[M_1 \| M_2] \models (p_1 \wedge p_2)$ *, and*

(b) A program $M = M_1 \| M_2$ *satisfies RTTL formula* p *if* $\models (p_1 \wedge p_2) \rightarrow p$ *.*

A top-down hierarchical method for developing real-time systems may now be followed. To develop a program M satisfying p, by decomposing M into two modules M_1, M_2, so that $M = M_1 \| M_2$, proceed as follows:

- The systems analyst produces an *abstract specification* (I_i, p_i) for each module M_i, where I_i is the interface specification and p_i the reactive behaviour of the module.
- The systems analyst must check that I_1, I_2 are compatible interfaces, and that $\models (p_1 \wedge p_2) \rightarrow p$.
- Each coding team is given the abstract specification (I_i, p_i) for the module M_i. The team must then implement the module by coming up with a body B_i so that p_i is modularly valid for M_i.

- Finally, the required modules are $M_i :: [\textbf{module};I_i;B_i]$ for $i \in \{1, 2\}$. The Decomposition Theorem guarantees that $M = M_1 \| M_2$ satisfies property p.

A team assigned to the implementation of a module M_i is given its interface specification I_i and an RTTL formula p_i in the variables of the interface specification, describing the expected reactive behaviour of the module. The task of the team is then to find a body B_i of M_i so that p_i is *modularly valid* for M_i. Many different bodies may satisfy the required constraints.

The *verification problem* we now wish to consider is: how does the coding team check that the reactive behaviour p_i is modularly valid for the module given by $M_i :: [\textbf{module};I_i;B_i]$.

(EQ 43) seems to require that modular validity for a module can only be checked by considering all its infinitely many interface compatible partners. However, there is a more direct approach to the problem.

The body B_i of the module can be mapped into a timed transition model (TTM). We add to the set of transitions of the TTM, an environmental transition τ_E representing all possible interferences of the environment with the operation of the module. This environmental transition is then arbitrarily interleaved with the transitions of the module, and represents all possible interferences that a (possibly hostile) environment may inflict upon the module. Transition τ_E pledges to preserve the values of all internal data variables, but it may arbitrarily change external data variables in accordance with the constraints imposed by the interface specification. We illustrate modular specification and verification by presenting a comprehensive example of real-time resource allocation.

5.3 The real-time resource allocator problem

5.3.1 Description of the problem

We illustrate modular reasoning by considering a real-time version of the resource allocator discussed in [21]. A resource allocator must manage the allocation of a shared resource among several competing processes (clients). The need to share resources is common not only in networked computers (e.g. disks and printers) or databases (e.g. record locking), but also in real-time devices.

In a flexible manufacturing system, a job may need to gain exclusive access to an automated guided vehicle or a forklift. In the real-time version of resource allocation, it is no longer sufficient to guarantee a process eventual use of the resource. It may be necessary to ensure that the resource is released in a timely fashion, or a critical job may not be serviced in time. We consider the simple case where there is one indivisible resource. However, more general cases can be specified, e.g. simultaneous exclusive access to a forklift, guided vehicle and workstation stand.

The code for an allocator module A with its client modules $C_i, i \in \{1, 2, 3\}$, is provided in Fig. 11. Each module has an interface specification and a body. The arrays g, r contain the grant and request variables respectively. We will shorten $g([i], r[i])$ to g_i, r_i respectively.

FIGURE 11. Allocator and clients

SUD = system under development, where: $SUD = A \parallel C$, $C = C_1 \parallel C_2 \parallel C_3$

module A {allocator with activity variable $a \in \{0, 1, 2, 3\}$ }
  external in r **: array[1..3] of integer** {array of request variables}
  out g **: array [1..3] of integer where** $g = 0$ {grant variables}
  internal i **: integer where** $i = 1$ { for fair allocation to the three clients}
  internal v **: integer where** $v = 0$ {variable to limit critical references}
  loop forever do
    0: $v := r[i]$
    1: if $v = 1$ **then**
    do $g[i] := 1$ **; 2: when** $r[i] = 1$ **then** $g[i] := 0$ **od**
    {count i modulo 3, i ranges between 1..3}
    3: $i := (i \oplus 1)$
end module A

---

module $C[i]$ {Client i with activity variable $c_i \in \{0, 1, 2, 3, 4\}$ }
  external in $g[i]$ **: integer** {grant variable is either 0 or 1}
  out $r[i]$ **: integer where** $r[i] = 0$ {request variable is either 0 or 1}
  loop forever do
    {noncritical processing at activity 0 or a request for the resource}
    0: $r[i] := 1$ **;** {request the resource}
    1: await $(g[i] = 1)$ **;** {await the allocator to grant the request}
    {critical processing at activity 2 and then release of the resource}
    2: $r[i] := 0$ **;** {release the resource}
    3: await $(g[i] = 0)$ {await an acknowledgement from the allocator}
end module $C[i]$

In the interface specification for A, the grant variables g_i are assigned a mode of **out** (but not **external**). Thus these variables cannot be written to by any other process that is interface compatible with A. However, other processes are allowed to import these variables as read only by declaring them as **external in**. Similarly the request variable r_i of client C_i may only be written to by C_i.

The timing requirements will be that the non-critical processing parts can take as long as they like [0,∞]. A request can be made for a resource at any point in time. The critical region processing takes no more than five clock ticks [0,5]. Assignments to r_i, g_i and guard checking takes exactly one tick. The increment (modulo 3) of the local variable i takes precisely one tick of the clock.

We first specify the requirements that the total system SUD must satisfy including mutual exclusion, conformance with the protocol, and real-time response. We then use the Composition Theorem to modularize the specifications. Finally, we show how to verify each module, from which the correctness of the total system follows.

Mutual exclusion:

$$\text{R1:} \quad \Box(g_1 + g_2 + g_3 \leq 1)$$

The formula R1 states mutual exclusion, i.e. at most one client can be granted access at any one time. Since the only operations allowed on the r_i, g_i is to set them to zero or one, it is clear that

$$\forall i | 3 \leq i \geq 1: \Box(g_i \in \{0, 1\} \wedge r_i \in \{0, 1\}) \quad \text{(EQ 44)}$$

is modularly valid for any of the modules in the figure.

Conformance with the protocol:

Both the allocator and the clients must conform to the protocol, i.e. the order of events should always be: C_i requests access, A grants access, C_i releases the resource, and A acknowledges the release.

We can use the formula

$$(r_i = 0 \wedge g_i = 0) \Rightarrow (r_i = 0 \wedge g_i = 0) \,\mathcal{W}\, (r_i = 1 \wedge g_i = 0)$$

(where r_i, g_i are the release and grant variables respectively for some client C_i) to characterize the next change allowed from the state $(r_i = 0 \wedge g_i = 0)$. Thus the resource is not granted to the client unless the client has previously requested the resource (there must be no unsolicited granting of the resource).

Using PTL (propositional temporal logic), the above property is actually equivalent to the simpler formula

$$\text{R2:} \quad (g_i = 0) \Rightarrow (g_i = 0) \,\mathcal{W}\, (r_i = 1 \wedge g_i = 0).$$

Once the customer makes a request, the request remains in place waiting for the allocator to grant the request.

$$\text{R3:} \quad (r_i = 1) \Rightarrow (r_i = 1) \,\mathcal{W}\, (r_i = 1 \wedge g_i = 1).$$

The resource will not be prematurely withdrawn (i.e. before the client releases it)

$$\text{R4:} \quad (g_i = 1) \Rightarrow (g_i = 1) \,\mathcal{W}\, (g_i = 1 \wedge r_i = 0).$$

The client will not make another request unless until after the allocator acknowledges the release of the resource

$$\text{R5:} \quad (r_i = 0) \Rightarrow (r_i = 0) \,\mathcal{W}\, (r_i = 0 \wedge g_i = 0).$$

Real-time response

The safety properties (R1–R5) can be satisfied in a system in which neither the clients never send a request message and hence the allocator need never in turn grant a request. The real-time response property will ensure that certain vital actions are taken.

Only the state satisfying $(r_i = 0 \wedge g_i = 0)$ is stable. Each of the other protocol states must be exited within a time bound. This is most succinctly specified by

$$\Box\Diamond_{\leq 31}(r_i = 0 \wedge g_i = 0) \quad \text{(EQ 45)}$$

i.e. the system will always reach a stable state within 30 ticks of the clock. The actual time will depend on the bounds of the atomic transitions. For illustration we will assume that evaluating a guard and then doing some assignment takes 1 tick of the clock.

The above response property makes decomposition into modular specifications difficult. This is because the property constrains at the same time variables owned by C_i (r_i) and variables owned by A (g_i). We must try to break (EQ 45) into smaller properties that constrain only one variable at a time.

The property (EQ 45) can be replaced with the next three properties. Every request for the resource must be granted within two ticks by the allocator, i.e.

$$\text{R6:} \quad (r_i = 1) \Rightarrow \Diamond_{\leq 24} (g_i = 1) .$$

Some cooperation from the clients is required. A client that has a resource must release it within 6 ticks of the clock, i.e.

$$\text{R7:} \quad (g_i = 1) \Rightarrow \Diamond_{\leq 6} (r_i = 0) .$$

We are assuming that a client uses the resource for no more than 5 ticks of the clock. We add one tick for resetting the request variable to come up with a 6 tick total.

Clearly, if a client C_i appropriates the resource for more than one tick, then the allocator cannot guarantee service to another customer C_j, $j \neq i$, without violating the mutual exclusion requirement.

An equally important allocator responsibility is to ensure that the client's release of the resource is duly acknowledged, i.e.

$$\text{R8:} \quad (r_i = 0) \Rightarrow \Diamond_{\leq 1} (g_i = 0) .$$

Since we have assumed that assignments and guard evaluation take zero time (relative to resource usage in the critical area), we may require in R8 an immediate acknowledgment on the part of the allocator when a resource is released.

Due to the safety requirements, a customer cannot make a next request unless its previous release was acknowledged by the allocator. R8 outlaws that type of devious behaviour on the part of the allocator that withholds service from the client by not acknowledging a release.

5.3.2 Modular specifications

The global requirements R1–R8 do not directly translate into a set of modular requirements. This is because a modular specification must hold in an environment that "misbehaves", e.g. the allocator cannot always count on clients that stick to the required protocol. Even if the environment behaves respectably, individual requirements may be stated more strongly than required.

There will often be a need to refer to changes in the request and grant variables. For example, to record the fact that the request variable r_i goes from zero to one (flagging a request) we could write $rq_i \stackrel{\text{def}}{=} (r_i = 1) \wedge \ominus (r_i = 1)$ (where $\ominus$ is the previous operator). The other definitions are given below:

$$rq_i \stackrel{\text{def}}{=} (r_i = 1) \wedge \ominus (r_i = 0)$$
$$gr_i \stackrel{\text{def}}{=} (g_i = 1) \wedge \ominus (g_i = 0)$$
$$rl_i \stackrel{\text{def}}{=} (r_i = 0) \wedge \ominus (r_i = 1)$$
$$ak_i \stackrel{\text{def}}{=} (g_i = 0) \wedge \ominus (g_i = 1)$$

The global requirement R for SUD is the conjunction of requirements R1–R8, which must be decomposed into a modular specification. Thus, we must come up with modularly valid requirements R_A, R_{C_i} for the allocator and clients respectively so that $R_A \wedge R_{C_1} \wedge R_{C_2} \wedge R_{C_3} \Rightarrow R$.

A possible methodology is to inspect the global requirements R1–R8 one by one, and to determine whether the considered module is the one responsible for that requirement. By the interface specifications, only the clients may write to the release variables r_i, and only the allocator may write to the grant variables g_i. Thus, for example, the mutual exclusion requirement R1 must be the responsibility of the allocator.

Modular specification of a client C_i

The first global requirement that a client must ensure is R3 given by

$$(r_i = 1) \Rightarrow (r_i = 1) \,\mathcal{W}\, (r_i = 1 \wedge g_i = 1) .$$

As explained by Manna-Pnueli (page 368), this property is far to strict. What is needed is the weaker specification

R9: $\quad rq_i \Rightarrow (r_i = 1) \,\mathcal{W}\, (r_i = 1 \wedge g_i = 1)$

i.e. once the client sets r_i to one, the request variable must remain high at least until the allocator responds by resetting the grant variable. The complementary global requirement R5 must be represented by the weaker property

R10: $\quad rl_i \Rightarrow (r_i = 0) \,\mathcal{W}\, (r_i = 0 \wedge g_i = 0)$

i.e. the modular version of the property must be stated with respect to a point at which r_i is reset.

The only real-time response property that constrains variables owned by C_i is R7 given by $(g_i = 1) \Rightarrow \Diamond_{\leq 2} (r_i = 0)$, which claims that the client must release the resource within two ticks of having gained access to it. It is impossible to guarantee a response to the grant variable being set if it is not set to 1 sufficiently long. To obtain a modular specification we therefore require a response to the setting of the grant variable only if the grant variable remains set at least until the response is generated, i.e.

R11: $\quad (g_i = 1) \Rightarrow \Diamond_{\leq 6} (r_i = 0 \vee g_i = 0) .$

The above property is satisfied if $(r_i = 0)$ holds (before the third tick) in response to the grant variable being set. Alternately, it will also be satisfied if g_i is reset before the third tick. If g_i is reset after $(r_i = 0)$ well and good. If not, then there is nothing the client can do about it, as the grant variable was reset by the allocator before the client could respond.

The complete modular specification for the client C_i is thus given by

R_{C_i}: R9 & R10 & R11

Modular specification of the allocator

The mutual exclusion property R1 given by $g_1 + g_2 + g_3 \leq 1$ is clearly the responsibility of the allocator as it refers to variables owned by A.

The two remaining protocol conformance specifications must be stated from points of change. We therefore obtain

$$\text{R12:} \quad (initial \vee ak_i) \Rightarrow (g_i = 0)\, \mathcal{W}\, (r_i = 1 \wedge g_i = 0)\,.$$

The property $ak_i \stackrel{\text{def}}{\equiv} (g_i = 0) \wedge \ominus (g_i = 1)$ cannot be true in the first position of a legal trajectory. The state-formula *initial* must therefore be inserted into the antecedent in order to obtain the appropriate unsolicited response property. The global property R4 becomes:

$$\text{R13:} \quad gr_i \Rightarrow (g_i = 1)\, \mathcal{W}\, (r_i = 0 \wedge g_i = 1)\,.$$

The global response properties R6 and R8 constrain the behaviour of the grant variable g_i of the allocator. R8 requires that the grant variable must be reset in response to the release variables being set. We again require that r_i remains zero long enough, so that we obtain:

$$\text{R14:} \quad (r_i = 0) \Rightarrow \Diamond_{\leq 1} (g_i = 0 \vee r_i = 1)\,.$$

The requirement R6 given by $(r_i = 1) \Rightarrow \Diamond_{\leq 7} (g_i = 1)$ is more complex to state modularly. A client C_i that makes a request to the allocator may not eventually be granted that request because some other rebellious client may refuse to release the resource. To release the allocator from the obligation of granting a request when there is a rebellious client, we may require

$$\text{R15:} \quad (r_i = 1) \Rightarrow \Diamond_{\leq 24} (g_i = 1 \vee r_i = 0 \vee [\exists j : j \neq i : \Diamond \Box_{<7} (g_j = 1)])\,.$$

This property states that if C_i has made a sustained request, then either the allocator will grant it the resource, or we can identify a rebellious client that at sometime holds the resource for at least a tick longer than it is supposed to (i.e. up to the 7th clock tick rather than up to the 6th tick of the clock as specified by R7).

It might be thought that the allocator can misuse the leniency specified by the right disjunct of R15 by leaving g_j set to one for two ticks of the clock (even after the client has released the resource). However, by R14, the allocator is obliged to reset g_i immediately in response to the resource being released. By R12, g_i must remain reset at least until the next request. Hence the leniency in R15 cannot be misused.

The modular specification for the allocator is thus defined by

R_A: R1 & R12 & R13 & R14 & R15.

It is now tedious but easy to confirm that $R_A \wedge R_{C_1} \wedge R_{C_2} \wedge R_{C_3} \Rightarrow R$. A theorem prover exits for the propositional untimed fragment of RTTL, which is useful for semi-automating this check [21]. We must now check that each of the conjuncts in the antecedent is modularly valid.

5.3.3 Modular verification

The environment of the module can be modelled by attaching to the transition set of the module an environmental transition τ_E, that can arbitrarily vary the external variables according to the interface specification. For example, the environmental transition for the allocator is $choose(g_1, g_2, g_3)$ (Table 5). Its partial

TABLE 5. Transitions of the allocator module (with its environment)

Name	enabling	transformation	lower	upper
a	$(a = 0)$	$[v{:}r_i, a{:}1]$	1	1
b	$(a = 1 \wedge v = 0)$	$[a{:}3]$	1	1
c	$(a = 1 \wedge v = 1)$	$[g_i{:}1, a{:}2]$	1	1
d	$(a = 2 \wedge r_i = 0)$	$[g_i; 0, a{:}\ 3]$	1	1
e	$(a = 3)$	$[i{:}\ (i \oplus 3)\ , a{:}\ 0]$	1	1
environment	$true$	$choose(g_1, g_2, g_3)$	0	∞

weakest precondition is defined, analogously to the *choose* command in [11], as: $pwp(choose(v), p) \stackrel{\text{def}}{=} (\forall i \in type(v) | p_i^v)$. The extension to many variables is defined in the obvious manner, i.e. $choose(v_1, v_2)$ is the simultaneous assignment to v_1, v_2 of some arbitrary values in their respective types. Hence, we have that $pwp(choose(v_1, v_2), p) \stackrel{\text{def}}{=} pwp(choose(v_1), p) \wedge pwp(choose(v_2), p)$.

The BUILD tool does not currently allow for the environment of a module to be built automatically. The environment must be specified explicitly as shown in the discussion of the response-heuristic (Fig. 9, Section 4.2). The proof diagram Fig. 10 demonstrates the modular validity of the response requirement R11 for the client module.

The mutual exclusion property R1 for the allocator module can be verified modularly by using the transition set shown in Table 5, and the invariant-heuristic.The bad part of the state is defined as

$$\begin{aligned} Bad &= \neg(g_1 + g_2 + g_3 \leq 1) \\ &= g_1 + g_2 + g_3 > 1 \end{aligned} \qquad \text{(EQ 46)}$$

The heuristic may now be used to prove that $\Box\neg Bad$ as shown in Fig. 12. We use the functional view of arrays and the notation of [11] for array assignment. Thus, an assignment $g[i] := d$ for the array g with index i and data expression d, is redefined as $g := (g; i{:}d)$, i.e. the array g is replaced by an array similar to g, except that the element at index i has the value of the expression d.

When the weakest precondition computation of the heuristic was used, node B was initially computed as the stronger formula $B \wedge (a = 1 \wedge v = 1)$. This stronger formula could also have been used, but it results in a larger proof diagram than necessary. Weakening the node to B as in Fig. 12, produces a simpler smaller diagram. The other nodes are weakened in the same way.

FIGURE 12. proof diagram for R1 of allocator

Bad: $(\Sigma\ j | 1 \le j \le 3 :\ g\ [j]\ ) > 1$

B: $(\Sigma\ j | 1 \le j \le 3 :\ (g;i:1)\ [j]\ ) > 1$

C: $(\Sigma\ j | 1 \le j \le 3 :\ (g;i \oplus 1:1)\ [j]\ ) > 1$

D: $(\Sigma\ j | 1 \le j \le 3 :\ (\ (g;i:0)\ ;i \oplus 1:1)\ [j]\ ) > 1$

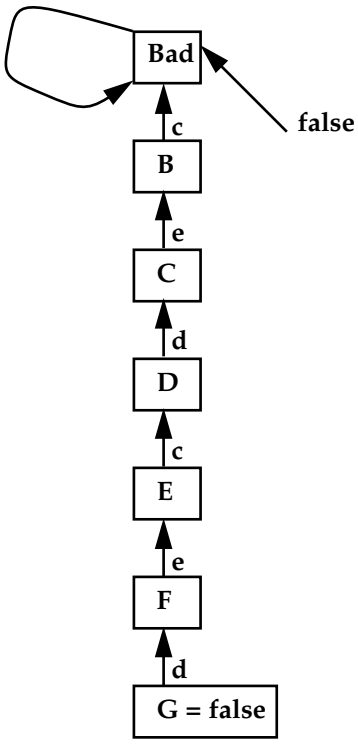
E: $(\Sigma\ j | 1 \le j \le 3 :\ (\ (\ (g;i:1)\ ;i:0)\ ;i \oplus 1:1)\ [j]\ ) > 1$

$E \equiv D$

F: $(\Sigma\ j | 1 \le j \le 3 :\ (\ (g;(i \oplus 1):0)\ ;i \oplus 2:1)\ [j]\ ) > 1$

G: $(\Sigma\ j | 1 \le j \le 3 :\ (\ (\ (g;i:0)\ ;(i \oplus 1):0)\ ;i \oplus 2:1)\ [j]\ ) > 1$

$G \equiv false$



Only connecting edges that are not selfloops are shown in the proof diagram. The environmental transition is always a selfloop as are the transitions a and b. State-formula $E = D$ because the outer assignment is dominant. $G = false$ because the value of array elements at indices $i, i + 1$ are zero. Hence it is impossible for the summation to be greater than one.

Since none of the nodes are consistent with the initial condition (of the allocator) given by $G_1 + G_2 + G_3 = 0$, the bad part of the state is unreachable, and hence requirement R1 is modularly valid for the allocator. It is straightforward to extend this argument to an allocator for an arbitrary number of clients.

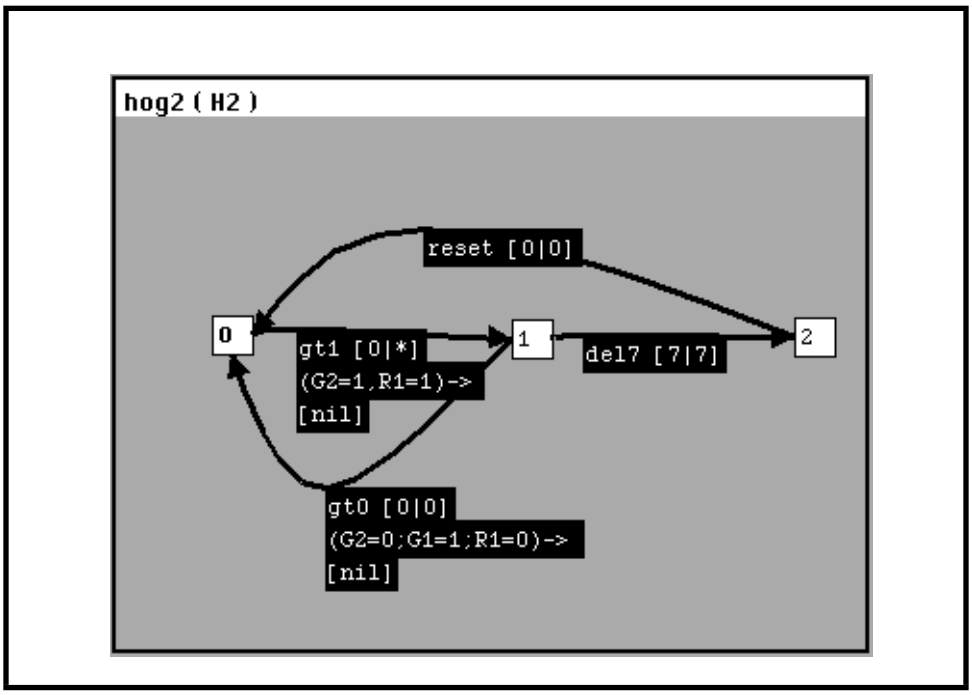
Observers (watchdogs)

We have shown how to use the heuristics to do modular verification of invariances and real-time response properties. What about those modular requirements (such as R13 and R15) that are not in the appropriate invariant or response formats.

Although it is worthwhile to provide heuristics for some other commonly appearing properties (e.g. the waiting-for requirement), there are too many permutations for this to be a general approach. Instead, we use the concept of an *observer* to convert other properties either into response or invariant properties. The heuristics can then be applied in the standard way.

An observer (or "watchdog") is a non-invasive TTM that observes and reacts to the module without changing any parameters of the module. The TTM `hog2` (with activity variable H_2) in Fig. 13 is an example of an observer that detects

FIGURE 13. Watchdog called "hog2" to detect a rebellious client



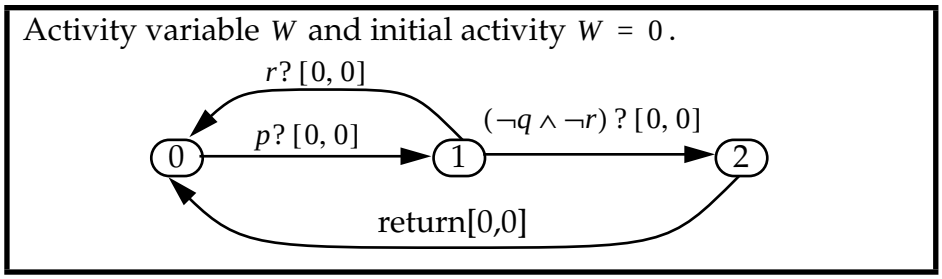
when client C_2 rebels, i.e. when the subformula $\Diamond\Box_{<7}(g_2 = 1)$ in R15 holds. A similar TTM `hog3` can be constructed to detect rebellion of C_3. To verify R15 it is sufficient to verify that

$$(r_1 = 1) \Rightarrow \Diamond_{\leq 7}(g_1 = 1 \vee r_1 = 0 \vee H_2 = 2 \vee H_3 = 2) \qquad \text{(EQ 47)}$$

for the TTM $A \| \texttt{hog1} \| \texttt{hog2}$. The property is in a format suitable for the application of the response-heuristic.

An observer TTM for the waiting-for property given by $p \Rightarrow (q \,\mathcal{W}\, r)$ (where p, q, r are state-formulas) is provided in Fig. 14. A transition $p?[0,0]$ has an enabling condition p and must be taken immediately (within zero ticks of becoming enabled). An observer transition always has precedence over transitions from

FIGURE 14. Observer TTM `obsw` for the waiting-for property $p \Rightarrow (q \mathcal{W} r)$



the system under observation in case both are eligible to be taken from a node in a proof diagram. The waiting-for property can then be checked by verifying the invariance

$$\models \Box (W \neq 2) \qquad \text{(EQ 48)}$$

for the composition of the observer and the module it is observing. The property is in a format to which the invariance-heuristic is applicable.

Consider the waiting-for requirement R13 of the allocator given by

$$gr_i \Rightarrow (g_i = 1) \mathcal{W} (r_i = 0 \wedge g_i = 1)$$
$$gr_i \stackrel{\text{def}}{=} (g_i = 1) \wedge \ominus (g_i = 0)$$

The antecedent gr_i is not a state-formula. However, the transition $p?[0, 0]$ in the observer of Fig. 14 can be replaced by two sequential transitions, the first of which detects $(g_i = 0)$ and the second of which detects $(g_i = 1)$. This new observer can then be composed in parallel with the allocator and the standard invariance-heuristic can be applied to verify R13.

The use of an observer extends the range of properties that can be checked by the heuristics to almost all RTTL properties.

6.0 Tools for automated specification and verification

Any framework such as TTM/RTTL will need tools to automate the design and verification of large systems. Real systems are complex. Systems engineers will want powerful visual tools for specifying their systems as well as for simulating and verifying them mechanically where possible. Some tools are already available for the TTM/RTTL framework and more are being planned. We have already mentioned that a theorem prover is available [21] for the untimed propositional fragment of RTTL, which is particularly useful in modular specification. We also describe below the development of a prototype toolset called StateTime [27].

StateTime uses visual specifications and temporal logic for automated design and verification. The BUILD tool allows the designer to model a system using a graphical language (TTMcharts) which are easily converted into TTMs. This tool was used to generate the charts shown in Fig. 6 and Fig. 9. The activities of a chart can be hierachically decomposed (AND or XOR decomposition) in a fashion similar to statecharts.

Any partial or complete model is immediately executable, which allows for rapid prototyping and validation. The description language is capable of describing the *given* behaviour of the environment (which may be unstructured and nondeterministic) as well as the *required* behaviour of the computer system (e.g. written in a structured high level language such as Ada). Thus, concurrency, nondeterminism, hierarchy, synchronization and communication, time bounds and integer data variables are supported.

The VERIFY and DEVELOP tools take TTMcharts as their input and allow for their computer-aided verification against RTTL requirements. The VERIFY tool [29] is used to model-check finite state TTMs. For an example of the use of BUILD and VERIFY for checking part of the shutdown procedure for the Candu reactor see [30]. The VERIFY tool can be used to do automated modular model checking which is useful for treating large systems [31].

The DEVELOP tool [24] is used for verifying infinite state systems using the RTTL proof system. Although initially written in CLP($\Re$) [25], it now uses the constraint logic programming language PrologIII [9]. Constraint logic can solve verification conditions and compute weakest preconditions in various domains such as integers, rationals, lists and booleans.

As an example of the use of the DEVELOP tool, consider verifying the requirement $\Box q$ (where $q \stackrel{\text{def}}{=} (x \neq 4)$) for the infinite state TTM in Fig. 4. Suppose a suggested proof outline (given in Step 13. to Step 17. on page 35) using the S-INV rule is proposed. The first step is to use BUILD to construct the equivalent chart and to enter the state-formulas p, q required by the rule. The script below, that confirms the correctness of the proof outline, illustrates the use of the tool:

```
%%% Check that “initial->p”
?- implies(initial,p).
*** YES ***

%%% Check that “p->q”
?- implies(p,q).
*** YES ***

%%% Check the leads-to relation “{p}Transitions{p}” using the VC-rule
?- leads_all(sample, p, p).
*** YES ***
```

The DEVELOP tool can also be used to check weakest preconditions and exiting edges of proof diagrams as required by the invariant and response heuristics. Proof diagrams such as those in Fig. 7 and Fig. 10 were constructed and checked in a few minutes with the help of the tool.

The DEVELOP and VERIFICATION tools are currently being experimented with for verifying large realistic systems that are too complex to verify easily by hand. The main technique is to use the compositional theorem to break the system into modules. Where a module is small enough to be model-checked the VERIFY tool is used. For modules that have very large or infinite timed reachability graphs, the DEVELOP tool is used.

7.0 Conclusions

The TTM/RTTL framework has a computational model, specification language, proof system and automated procedures and a toolset for treating a wide variety of real-time reactive systems. The development methods presented in this paper augment the framework with modular reasoning and automated heuristic proof methods for modules, and the ability to derive their timing constraints. These methods together with the use of the finite state verifier on modules [31], and the theory of refinement proposed in [18], provide a feasible approach to the systematic hierarchical and modular development of real-time reactive systems.

The toolset will benefit by the addition of interface specifications for modules, and the support of the decomposition and refinement techniques discussed in this paper. With these additions, more industrial examples will need to be specified and verified before an industrial strength tool can be developed.

8.0 References

[1] Abadi, M. and L. Lamport. *Conjoining Specifications*. DEC Research Center. SRC 118, 1993.

[2] Alur, R., C. Courcoubetis, and D.L. Dill. "Model Checking for Real-Time Systems." In *Proc. 5th Conference on Logic in Computer Science*, IEEE Computer Society Press, 1990.

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

[4] Bernstein, A. and P.K. Harter. "Proving Real-time Properties of Programs with Temporal Logic." In *Proc. of 8th annual ACM Symposium on Operating Systems Principles*, 1-11. 1981.

[5] Berthomieu, B. and M. Diaz. "Modeling and Verification of Time Dependent Systems Using Time Petri Nets." *IEEE Transactions on Software Engineering*, 17(3): 259-273, 1991.

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

[9] Colmeraurer, A. "An Introduction to PrologIII." *Communications of the ACM*, 33(7): 69–90, 1990.

[10] Dijkstra, E.W. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, 1976.

[11] Gries, D. *The Science of Programming*. Springer-Verlag, 1985.

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

[13] Henzinger, T.A., X. Nicollin, J. Sifakis, and S. Yovine. "Symbolic Model Checking fo Real-Time Systems." In *Proc. 7th Symposium of Logics in Computer Science*, IEEE Computer Society Press, 1992.

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

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

[16] Jahanian, F. and D. Stuart. "A Method for Verifying Properties of Modechart Specifications." In *Proc. 9th Real-time Systems Symposium*, IEEE Computer Society Press, 12-21, 1988.

[17] Koymans, R., J. Vytopil, and W.P.d. Roever. "Real-time Programming and Asynchronous Message Passing." In *Proc. 2nd Annual Symposium on Principles of Distributed Computing*, 187-197. Montreal: 1983.

[18] Lawford, M., W.M. Wonham, and J.S. Ostroff. "State-Event Labels for Labelled Transition Systems." In *Proc. 1994 Conference on Decision and Control*, Orlando, FL, 1994 (to appear).

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

[20] Manna, Z. and A. Pnueli. "The Anchored Version of the Temporal Framework." In *Models of Concurrency: Linear, Branching and Partial Orders*, 201-284. LNCS 354. Springer-Verlag, 1989.

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

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

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

[24] Ostroff, J.S. "Constraint Logic Programming for Reasoning about Discrete Event Processes." *The Journal of Logic Programming*, 11(3&4): 243–270, 1991.

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

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

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

[28] Ostroff, J.S. "Verification of Safety Critical Systems Using TTM/RTTL." In *Real-Time: Theory in Practice*, Springer-Verlag, 1992.

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

[30] Ostroff, J.S. "Visual Tools for Verifying Real-Time Systems." In *Theories and Experiences in Real-Time Systems*, AMAST Series in Computing, Vol. 2. Iowa City: World Scientific Press, 1994.

[31] Ostroff, J.S. "Automated Modular Specification and Verification of Real-Time Reactive Systems." In 1995 (submitted for publication).

[32] 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.

[33] Pnueli, A. "The Temporal Semantics of Concurrent Programs." *Theoretical Computer Science*, 13:45-80, 1981.

[34] Toi, H.W. and D.L. Dill. "Approximations for Verifying Timing Properties." In *Theories and Experiences for Real-Time System Development*, eds. T. Rus and C. Rattray. AMAST Series in Computing, Vol. 2. World Scientific Press, 1994.

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