Automated Modular Specification and Verification of Real-Time Reactive Systems

Jonathan S. Ostroff
Department Of Computer Science, York University[1],
4700 Keele Street, North York Ontario, Canada, M3J 1P3.
Email: jonathan@cs.yorku.ca Tel: 416-736-2100 X77882 Fax: 416-736-5872
Electronic Technical Report Number: CS-ETR-94-06
ftp.cs.yorku.ca:/pub/TECH-REPORTS/General-CS/CS-ETR-94-06/text.ps2.Z

Abstract: Model-checking is a powerful automated technique for verifying finite state real-time safety critical systems, but suffers from a combinatorial explosion of states as system complexity increases. In this paper, we introduce a method for compositional reasoning in real-time temporal logic that is suitable for model-checking finite state real-time reactive modules with data variables. This allows for the formal development of systems by top-down hierarchical program derivation. A system can be decomposed into modules, and the modules checked separately instead of checking the complete system all at once. This procedure often results in a significant decrease in the size of the reachability graphs that must be checked, particularly if the modules are loosely coupled.

The modular model-checking method developed in this paper is illustrated using a real time resource allocation problem and the StateTime toolset. StateTime is a prototype toolset that uses visual specifications and temporal logic for the design and verification of real-time systems. The StateTime toolset has been used on small but non-trivial industrial examples. The incorporation of the modular methods discussed in this paper will allow StateTime to evolve from a prototype into an industrial strength tool.

1.0 Introduction

The record of successful applications of formal verification techniques is slowly growing, although misconceptions surrounding the use of formal methods still abound [8]. The ultimate aim is to firmly integrate verification techniques into the software design cycle as an essential part of quality control. The sale of a non-verified piece of software, protocol, algorithm or device should be as unthinkable as the deployment of bridges without suitable stress analysis or the prescription of potentially harmful drugs without testing [12].

The design of most software can benefit from the use of formal methods, but their use becomes crucial in the design of real-time safety critical software. While CASE tools may be useful for software design in general, tools for safety critical

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

systems that mechanize specification and verification are essential if these methods are to be applied in industry.

In this paper, we show how an existing prototype toolset called StateTime can be used to automate modular verification of real-time reactive systems specified using a visual state based language. The compositional techniques developed in this paper are crucial for dealing with the problem of combinatorial explosion of states.

In compositional verification, the properties of a system (compound construct) can be deduced from specifications for its constituent parts (modules), without any further information about the internal structure of these parts. A module is specified by its interface specification I (e.g. its communication channels or external variables), a behavioural description R in real-time temporal logic that specifies the ongoing interaction of the module with its environment, and the body (or "secret") of the module.

The requirements R refer only to parameters of the interface I, and not to local variables declared in the body of the module. Once the body is model-checked against its module requirements R, the body together with all its local variables can be hidden. The module is then represented by its abstract specification (I,R). All the assumptions which are needed regarding the environment — because these influence the behaviour of the model — are incorporated as explicit parameters in R. Thus the module will behave according to its abstract specification in any environment and regardless of the inner syntactic structure of the body.

StateTime, visual languages and the TTM/RTTL framework

Many engineers prefer to work with pictures when specifying reactive systems. Tools that use formal graphical languages have been developed [9,15,27], which can be used for model execution and reachability analysis.

StateTime [22] is a prototype toolset that 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 based on timed transition models (TTMs). 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 tool is used to model-check finite state TTMs using real-time temporal logic (RTTL). The DESIGN tool is used for verifying infinite state systems using a proof system. For an example of the use of StateTime for checking part of the shutdown procedure for the Candu reactor see [24].

TTMs are a generic computational model for real-time reactive systems. Concurrency is represented by interleaving of atomic actions (transitions) chosen, one at a time, from parallel processes. A conceptual external global clock is a generator of tick events that are also interleaved with other actions. Time bounds on actions are specified with respect to clock ticks. An action $\tau[l, u]$ must wait for l clock ticks before it can be taken. If it is continually enabled, and not preempted from occurring, it must be taken by u ticks of the clock. The system evolves either

by performing an instantaneous action (or sequence of actions) or by letting time pass (taking a clock tick).

A variety of concrete programming languages such as Ada, or concurrent formalisms such as Petri nets and CSP can be mapped into TTMs [18,19]. The TTM-chart is used by the StateTime tool as its concrete visual specification language. However, the techniques developed in this paper are equally applicable to other concrete languages provided they can be mapped to TTMs.

RTTL can describe ongoing real-time reactive behaviour abstractly (free from implementation details) and concisely. System requirements can be succinctly stated in RTTL, refined into TTMs, and then implemented in some concrete language. Alternatively, an already implemented program can be transformed into a TTM. The program requirements can then be specified in RTTL, and verified using StateTime. StateTime supports top down decomposition as well as bottom up development.

The survey article [21] compares the TTM/RTTL framework to other logics and process algebras.

Model-checking real-time systems

Model-checking is a powerful technique for verifying finite state systems because it is easily automated, and does not require the same skill on the part of the software engineer that the use of a proof systems entails. Model-checking together with a visual specification language is thus a good candidate for inclusion in an industrial strength tool.

Model-checking was first introduced by the authors of [6], and extended to real-time systems in [19,20], 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 [2,4].

For VERIFY, in the worst case, the timed reachability graph may be larger than the untimed graph by a factor proportional to the largest finite upper time bound. The subset of RTTL properties treated by the verifier can be checked in time complexity linear in the size of the timed reachability graph. The extensions, proposed in [7], will allow arbitrary branching time properties to be checked in time linear with respect to the product of the length of the property and the size of the graph.

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 [3,5,11,26]. 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. However, a modular and hierarchical approach will still be needed to beat state explosion.

By decomposing proofs into modular parts, verification of systems that hitherto required completely hand-guided proofs, can now be checked automatically using decision procedures [16]. Although compositional proof systems for real-time temporal logics and a compositional axiomatization of statecharts are available [13,14], there are as yet no tools reported for compositional model-checking

of real-time reactive systems with hierarchical visual specification languages that use data variables.

Contribution and organization of this paper

In this paper, we present a method for using the visual tools and model-checker of StateTime compositionally for specification and verification of real-time reactive systems. We suggest improvements to the tool to facilitate modular reasoning. Compositional model-checking as discussed in this paper, model reduction via bisimulations as described in [17], and the further development of the TTMchart visual specification language are all essential components for evolving StateTime from a prototype into an industrial strength tool.

The state-event observers discussed in [17] allow for high level abstractions (quotients) of TTMs computable in polynomial time. Thus, large hierarchical systems can be treated in the following way. Use state-event observers to effect as much hierarchical model reduction as possible on the variables and events of interest. Then use compositional reasoning on the reduced modules to verify the requirements.

This paper is organized as follows. In Section 2.0, we summarize some features of the StateTime toolset. In Section 3.0, we describe the theory of compositional model-checking in RTTL. Section 4.0 applies the theory to a comprehensive example (real-time resource allocation). Modular verification is significantly more efficient than constructing the total reachability graph. Modular specification requires more insight from the designer, but results in a more robust structured design.

2.0 The StateTime toolset, TTMcharts and RTTL

The StateTime toolset [22] consists of various tools for designing real-time safety critical systems. The BUILD tool is used to construct TTMcharts, and the VERIFY tool can model-check these charts for various properties specified in real-time temporal logic.

TTMcharts are similar to statecharts [9], but with (non-blocking) broadcast communication replaced by (blocking) synchronization as in the Ada rendezvous or CSP message passing. A richer class of timing properties can be directly expressed in TTMcharts than in the charts of the Statemate tool [10]. An event τ can have a closed time interval as a firing condition (e.g. $\tau[3, 7]$), or be spontaneous (e.g. $\tau[0, \infty]$). A spontaneous transition may occur at any moment or never.

A typical chart built with the StateTime tool is shown in Figure 1, where hierarchy (clustered activities, default states and XOR-composition), concurrency (AND-composition), synchronization (shared events), and timing are illustrated. The TTMchart can be mapped into a TTM. A TTM consists of a set of variables (activity and data variables), an initial condition, and a set of transitions corresponding to the chart events. The transitions of the TTM corresponding to the chart of Figure 1are shown in Figure 2. Transitions have an enabling condition, transformation function and time bounds, on the basis of which the formal semantics and timed reachability graph of the TTM can be defined [19,20,25].

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

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

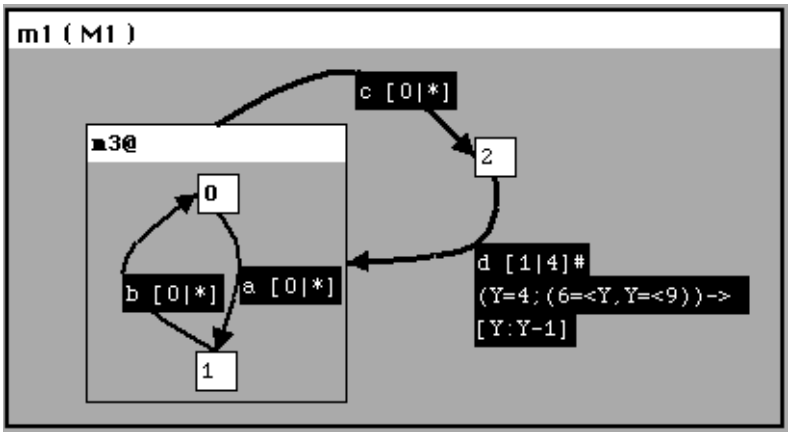
TTMcharts can be developed top down or bottom up. Working top down, the root activity m is AND-decomposed into subactivities m1 and m2, i.e. m = m1 || m2. AND-composition is indicated by dashed boxes (see bottom picture). The root activity m is also called a TTMchart.

The structured activity m1 is XOR-decomposed into the structured activity m3 and the leaf activity 2. An activity with internal structure is followed by the "@" symbol. Leaf activities have no internal structure.

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

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

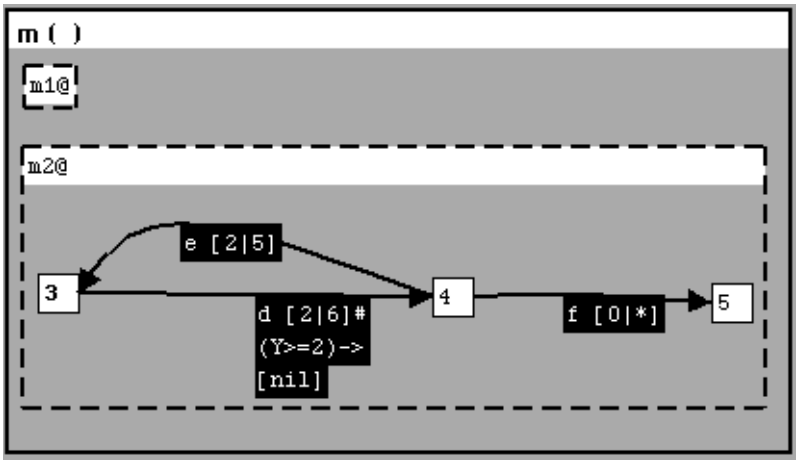
Each structured activity has its own activity variable. Thus M1, M2, M3 are the activity variables of activities m1, m2, m3 respectively.



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

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

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



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

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

The shared event d is taken after having been in activities 3 and 2 simultaneously (with both guards continuously true) for between two and four ticks of the clock. The lower bound is the maximum of the two component events in m1 and m2 and the upper time bound is the minimum of the two components (see the bounds d[2,4] for event d in the table below).

A TTMchart can be converted into a TTM. A TTM consists of a set of variables, an initial condition, and a set of transitions. Each transition of the TTM corresponds to an event in the chart (or pair of synchronizing events). A transition has an enabling condition, transformation function, and lower and upper time bounds as shown in Figure 2.

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

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

such as M1, M2 and M3 (see Figure 1) are used to write state-formulas describing the state of the chart. For example, the state-formula

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

asserts that the chart is in subactivity 1 of the clustered activity m3 and the data variable Y is less than 7.

A computation of a TTMchart is an infinite sequence of global states executed by the chart, starting in an initial state, with successor states computed by the transition of enabled events. The tick transition occurs an infinite number of times in such a computation. The tick transition always eventually fires even if there are no other eligible events.

State-formulas are boolean valued expressions in the activity and data variables, e.g. (EQ 1). RTTL formulas are constructed from state-formulas together with special temporal logic operators such as □ (*henceforth*) and ◇ (*eventually*). Let $f, f_1, f_2 \ldots$ stand for state-formulas in Table 1. The table documents some of

TABLE 1. Some properties that can be model-checked by VERIFY

How the property is read	Property	Definition of the property
Invariance: f_1 entails henceforth f_2.	$f_1 \Rightarrow \Box f_2$	In any reachable state s in which the state-formula f_1 holds, the formula f_2 must also hold in s and in all following states.
Real-time response: f_1 entails eventually f_2 within l to u ticks	$f_1 \Rightarrow \Diamond_{[l,u]} f_2$	In any reachable state s in which f_1 holds, f_2 must also hold in some following state s' which is at least l ticks but no more than u ticks after s.
Unless or *waiting-for:* f_1 entails f_2 waiting for f_3.	$f_1 \Rightarrow (f_2 \mathcal{W} f_3)$	If f_1 holds in any reachable state s, then in s and all following states the formula f_2 holds continuously or until the next occurrence of f_3.

the RTTL properties that the VERIFY tool can check. For example, the chart m (of Figure 1) can be checked for the property

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

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

The temporal formula $\ominus p$ (which uses the *previous* operator) is true in a state (but not the initial state) of a computation, if p is true in the previous state. The temporal formula $\Box_{<t} p$ asserts that p holds true up to and including the t-th tick of the clock. Thereafter, p's truth value is unconstrained.

While a state-formula can be checked for satisfaction in a global state, an RTTL formula must be checked for satisfaction in a computation. Given a TTMchart M and an RTTL formula p, we say that p is *M-valid* if all computations of M satisfy p, and we write $M \models p$. The reader is referred to [19,20,23,25] for a complete treatment of the TTM/RTTL framework and the verifier.

3.0 Compositional reasoning in RTTL

The TTM/RTTL framework is based on the temporal logic for reactive systems in [18], with the necessary extensions and proof rules for timed transitions. This means that any valid temporal formula in [18] is also valid in RTTL. We may therefore use the theory of compositional reasoning presented in [18] within the TTM/RTTL framework as well. We briefly describe below how compositional reasoning is done.

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 module A in Figure 3. 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 local variable declarations (e.g. see Figure 3). These local 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.

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

$$[M_1 \| M] \models p \text{ for every module } M \text{ that is interface compatible with } M_1. \qquad \text{(EQ 2)}$$

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 2) is the following theorem

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 method for developing real-time systems may now be followed. To develop a program M satisfying p, design compatible interfaces I_1, I_2 and behavioral specifications p_1, p_2 for the respective modules so that $(p_1 \wedge p_2) \rightarrow p$. Then develop bodies B_1, B_2 that are modularly valid for (I_1, p_1) and (I_2, p_2) respectively. Finally, the required modules are $M_i :: [\textbf{module};I_i;B_i]$ for $i \in \{1, 2\}$.

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: if a design team provides us with a module $M_i :: [\textbf{module};I_i;B_i]$ and an RTTL behavioural specification p_i, how can we check (automatically if possible) that p_i is modularly valid over M_i.

(EQ 2) 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.

As explained above, a TTMchart can be converted into a timed transition system (TTM) consisting of a variables set, initial condition and set of transitions (see Figure 1). We add to the set of transitions an environmental transition τ_E representing all possible interferences of the environment with the operation of the module. This environmental transition is arbitrarily interleaved with the transitions of the module. Transition τ_E pledges to preserve the values of all non-external data variables, but it may arbitrarily change external data variables.

This suggests that we can model-check a module M_i for a property p_i using StateTime, provided all variables range over finite types, as follows:

- Use the BUILD tool to construct a chart corresponding to the body of M_i.
- Construct an environmental chart that arbitrarily varies the external variables of module M_i.
- AND-compose the body chart M_i and environmental charts M_E together to form a new chart $M = M_i \| M_E$, representing the combined interaction of the module and its environment (e.g. see Figure 5 which is the combined chart for a client module in Figure 3).
- Use VERIFY to model-check that $M \models p_i$.

The above procedure is usually significantly more efficient than checking the complete program. This is particularly so if the external interface variables are few in number (i.e. there is only a loose coupling between the module and its environment) compared to the number of local variables of its partner modules. Furthermore, the module may not be required to perform in an arbitrarily unconstrained environment. It may only be required to satisfy its behavioral property if the environment is guaranteed to behave in a certain constrained fashion. Such a constrained environment will further reduce the size of the reachability graph that must be generated.

We illustrate modular model-checking by considering a real time version of the resource allocator discussed in [18]. In the real-time version, it is no longer adequate that a client will eventually give up the shared resource. We will require that there is an upper time bound by which the resource must be released. Similarly, the allocator must grant the resource once requested in a timely fashion. The requirements of the various modules will thus require real-time temporal logic for its behavioral specification.

4.0 An example

A resource allocator must manage the allocation of a shared resource among several competing processes (clients). In a real-time system, it is not sufficient to guarantee a critical process eventual use of the resource. Rather, there will be strict time bounds by which the resource must be made available.

The need to share resources is common not only in computers (e.g. disks or printers) and 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. We consider the simple case where there is one indivisible resource. However, more general cases can also be specified, e.g. simultaneous exclusive access to a forklift, guided vehicle and workstation stand.

An allocator A with its clients C_1, C_2, C_3 is shown in Figure 3. The arrays g, r contain the grant and request variables respectively. We will refer to the array variables $r[i], g[i]$ as r_i, g_i respectively.

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 non-critical processing parts can take as long as they like, i.e. they have time bounds of [0,∞]. Hence 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 is assumed to take one tick of the clock. The increment (modulo 3) of the local variable i also takes precisely one tick of the clock [1,1].

The corresponding TTMchart is shown in Figure 4. TTMcharts do not (yet) allow for arrays, so the arrays g, r must be modelled by independent variables.

FIGURE 3. Allocator A and clients $C[i]$

$$SUD = A \| C$$
$$\text{where } C = C_1 \| C_2 \| C_3$$

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

---

module $C[i]$ {Client i }
external in $g[i]$: boolean {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;
$r[i]$:=1 ; {request the resource}
await $(g[i] = 1)$; {await the allocator to grant the request}
critical processing;
$r[i]$:=0 ; {release the resource}
await $(g[i] = 0)$ {await an acknowledgement from the allocator}
end module $C[i]$

4.1 Requirements for the complete system

We provide the requirements for the complete system. We then describe modular requirements which together entail the complete specification. Finally we do modular model-checking using the VERIFY tool. The first system requirements is:

Mutual exclusion:

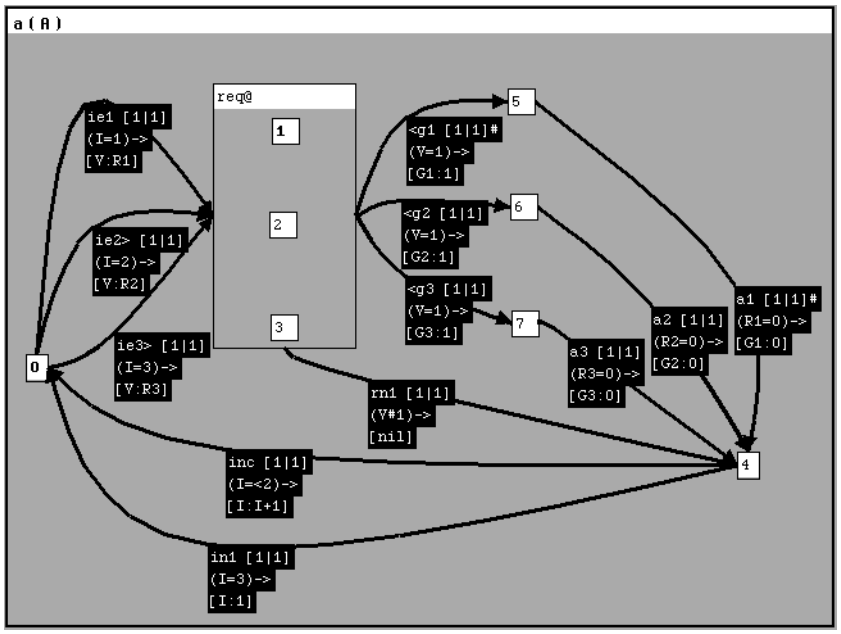
R1: $\quad \Box [(g_1 + g_2 + g_3) \leq 1]$

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

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

is modularly valid for any of the modules of the resource allocator.

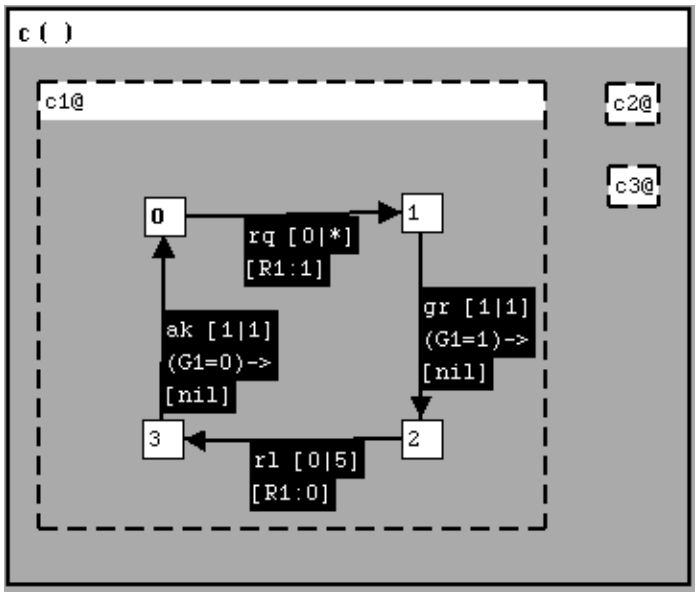
FIGURE 4. TTMchart of resource allocator corresponding to program in Figure 3



$$a = allocator$$
$$c = clients$$
$$SUD = a \parallel c$$
$$c = c_1 \parallel c_2 \parallel c_3$$

SUD = a || c, where a is the allocator and c is the parallel composition of the clients, i.e. c = c1 || c2 || c3. The arrays g and r are represented by the independent data variables G1, G2, G3 and R1, R2, R3 respectively. I and V are local integer data variables.

The event rn1 (see above) has the default behaviour for exiting from the clustered activity req. This event is taken no matter where in the clustered activity execution is provided the guard V#1 is true, i.e. the enabling condition of rn1 is V#1,A=req where A is the activity variable of the allocator.



The event ie1 in the allocator has the default behaviour for entering req, i.e. it leads to the default activity **1** of req. The events ie2, ie3, g1, g2, g3 are changed from the default behaviour for the clustered activity req. Event iei leads to activity i of req, and gi leads from activity i (where i is 1,2 or 3). The change from the defaults is indicated by the symbols "<" and ">" after the event names iei and gi.

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 (by setting r_i to one), A grants access

(by setting g_i to one), C_i releases the resource (by resetting r_i back to zero), and A acknowledges the release (by resetting g_i). We can use the formula

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

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 unsolicted 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 the clients never send a request message and hence the allocator need never grant a request. The real-time response property will ensure that certain vital actions are taken in bounded time.

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

$$\Box\Diamond_{\leq 31} (r_i = 0 \wedge g_i = 0) \qquad \text{(EQ 4)}$$

i.e. the system will always reach a stable state within 31 ticks of the clock. The actual time in any given implementation will depend on the bounds of the atomic transitions. For illustration, we have assumed that evaluating a guard and then doing some assignment takes one tick of the clock in the implementation of Figure 4. However, the requirements can ignore implementation details of this kind.

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 (i.e. the request variable r_i) and variables owned by A (i.e. the grant variable g_i). We must try to break (EQ 4) into smaller properties that constrain only sets of variables from a single module at a time.

The property (EQ 4) can be replaced with the next three requirements. 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 its allotted time, 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.

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

Due to the safety requirements, a customer cannot make its 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.

The requirements R1–R8 specify all the properties that must be satisfied by $SUD = A \| (C_1 \| C_2 \| C_3)$. The verifier was used to generate the graph and check each of these properties with the complete analysis performed in a total time of under three minutes. However, as will be shown later, as the system to be checked increases in size, the resulting combinatorial explosion of states makes it imperative that a modular analysis be undertaken.

The time bounds in the temporal operators of requirements R6, R7 and R8 need not be specified explicitly to the verifier. The verifier will return the most liberal interval $[l, u]$ over which the property is true. This is an important feature of the verifier that is useful in debugging the system. For example, if an interval $[0, \infty]$ is returned, then the implementor knows immediately that there is at least one trajectory in which the system can cycle forever without reaching the goal predicate, and the path to that cycle from an initial state is part of the diagnostic information returned by the verifier.

4.2 Modular specification of the resource allocator

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.

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 $rq_i \stackrel{\text{def}}{=} (r_i = 1) \wedge \ominus (r_i = 1)$.

The other definitions of changes to the request and grant variables are

$$\begin{aligned} rq_i &\stackrel{\text{def}}{=} (r_i = 1) \wedge \ominus (r_i = 0) \\ gr_i &\stackrel{\text{def}}{=} (g_i = 1) \wedge \ominus (g_i = 0) \\ rl_i &\stackrel{\text{def}}{=} (r_i = 0) \wedge \ominus (r_i = 1) \\ ak_i &\stackrel{\text{def}}{=} (g_i = 0) \wedge \ominus (g_i = 1) \end{aligned} \qquad \text{(EQ 5)}$$

The global requirement R for SUD is the conjunction of requirements R1–R8, which must be decomposed into modular specifications. 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 is 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 in [18, p368], this property is far to strict, and will not in general be true of any reasonable client (e.g. after both r_i and g_i are set the environment can reset g_i before r_i can be set). What is needed is the weaker specification

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

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. It is sufficient to state the property only at the exact points at which r_i has just changed from zero to one. The complementary global requirement R5 is specified by

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

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 $(g_i = 1) \Rightarrow \Diamond_{\leq 6} (r_i = 0)$, which claims that the client must release the resource within six ticks of having gained access to it. It is impossible to guarantee a response to the grant variable being set if it is not kept set sufficiently long. A modular specification will 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.

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

The above property is satisfied if $(r_i = 0)$ holds (before the seventh tick) in response to the grant variable being set. Alternately, it will also be satisfied if g_i is reset before the seventh 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 $\Box(g_1 + g_2 + g_3 \leq 1)$ 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 $ak_i \stackrel{\text{def}}{=} (g_i = 0) \wedge \ominus(g_i = 1)$ 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 reset sufficiently long, which is specified by:

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

The requirement R6 given by $(r_i = 1) \Rightarrow \Diamond_{\leq 24}(g_i = 1)$ 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 and including the 7th clock tick rather than releasing it before the 7th tick as specified by R7).

It might be thought that the allocator can misuse the leniency specified by the right disjunct in the consequent of R15 by leaving g_j set to one for seven ticks of the clock (even after the client has released the resource). However, by R14, the allocator is obliged to reset g_i almost 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.

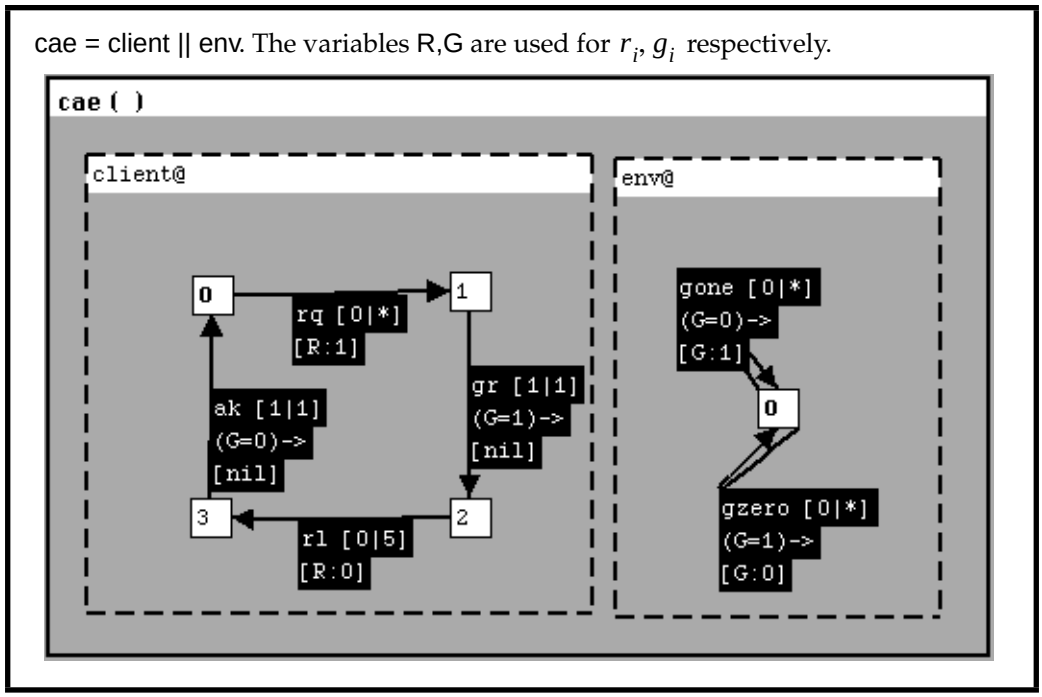
It is tedious but straightforward to confirm that $(R_A \wedge R_{C_1} \wedge R_{C_2} \wedge R_{C_3}) \rightarrow R$. A theorem prover exists for the propositional untimed fragment of RTTL, which is use-

ful in automating most of this check [18]. We must now check that each of the conjuncts in the antecedent is modularly valid.

4.3 Modular Verification

Modular specifications can be used to verify the resource allocator more efficiently by model-checking each component separately rather than the total system all at once. For example, to check the modular specifications of the clients it is sufficient to check the TTMchart shown in Figure 5. The two events gone and gzero

FIGURE 5. A generic client chart with its unconstrained environment



model the changes that can occur at any point in the environment. The size of the graph for checking the client module alone is 72 (states and edges), while the size of the graph for the total system is 1424 (Table 2).

While the reachability graph of a client module is an order of magnitude smaller than that of the total system, the allocator and its unconstrained environment is about the same size as the total system. However, the requirements R9 and R10 for the client constrain the behaviour of r_i. Thus, instead of an unconstrained environment, we may instead verify the allocator with the constrained environment shown in Figure 6. This reduces the size of the model to be checked to 410 which is substantially less than checking the total graph of size 1424 (Table 2).

In larger systems the modular approach substantially reduces the size of the reachability graph even using an unconstrained environment. For example, when the allocator has six clients, the size of the reachability graph of the total system is much larger than the allocator module with an unconstrained environment (Table 2). Constraining the environment of the allocator reduces the size of the model to be checked even more. The allocator can itself be decomposed into smaller modules if further model reduction is required.

FIGURE 6. The constrained environment of the allocator

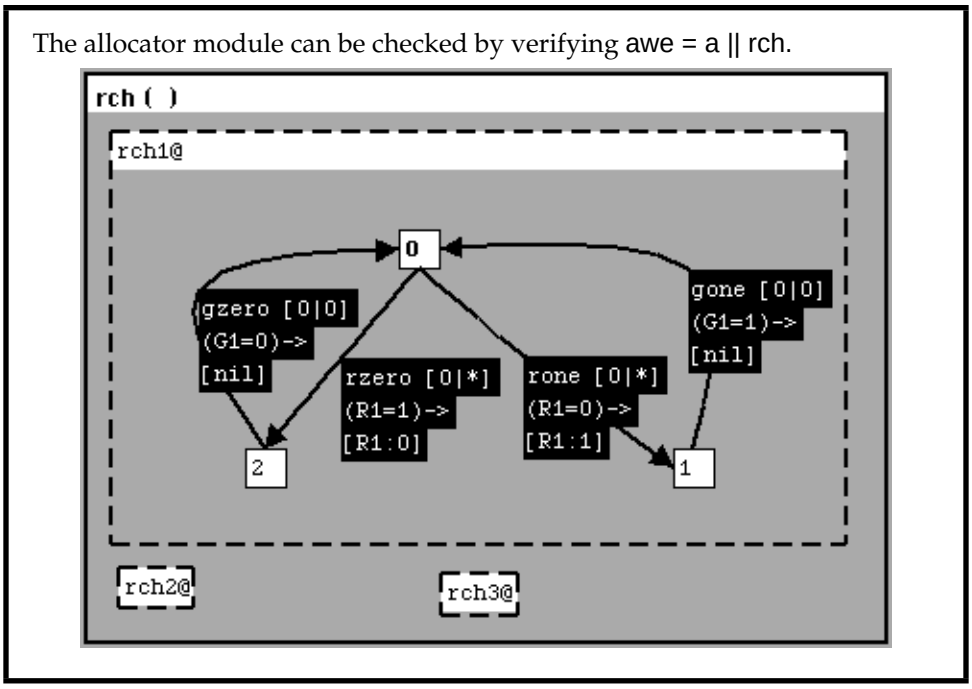


TABLE 2. Size of reachability graphs for the various TTMcharts

TTMchart	**Size of reachability graph = states + edges**
Total system of 3 clients and allocator (both charts in Figure 4 running in parallel).	**Size of graph:** 1424 Unique States = 200, Total States = 412, Total Edges = 1012 **Time to generate graph:** 1.7 minutes
Module of client and its unconstrained environment	**Size of graph:** 72 Unique States: 8, Total States = 20, Total Edges = 52.
Module of allocator for 3 clients, together with its environment constrained by the client requirements R9 and R10 (see Figure 6).	**Size of graph:** 410 Unique States = 76, Total States = 140, Total Edges = 270
Total system with six clients and allocator.	**Size of graph:** 86,124 Unique States = 5440, Total States = 18136, Total Edges = 67988
Module of allocator for 6 clients with its unconstrained environment.	**Size of graph:** 42,496 Unique States = 2752, Total States = 5312, Total Edges = 37184
Module of allocator for 6 clients with its constrained environment.	**Size of graph**: 22,944 Unique States = 2576, Total States = 4688, Total Edges = 18256

We note that one cannot really constrain the environment, as a module has no control over its environment. The constraints referred to above are really assumptions about the environment similar to traditional assumption/guarantee techniques. An assumption/guarantee specification for a concurrent program is a generalization of the pre/post-condition specification for a sequential program. It asserts that a module provides a guarantee G (e.g. the allocator requirements R_A)

if its environment satisfies an assumption E (e.g. the client requirements R9 & R10).

We have used single-assumption specifications, i.e. $[R_C \wedge (R_c \rightarrow R_A)] \rightarrow R$, which is trivially valid (because $R_A \wedge R_C \rightarrow R$ is valid). However, mutual-assumption specifications, e.g. $[(R_A \rightarrow R_C) \wedge (R_c \rightarrow R_A)] \rightarrow R$, in which each system guarantees to satisfies the other's environment assumption, are not in general valid for liveness properties. The techniques discussed in [1] can be used for mutual - assumption specifications in RTTL as well.

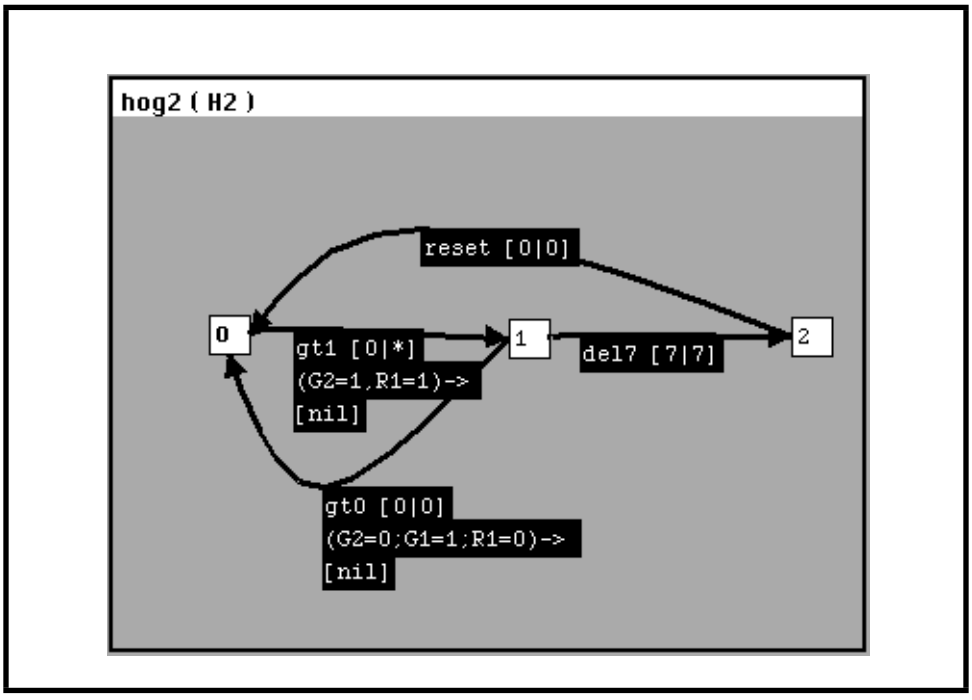
Watchdogs

The current verifier checks a small but important subset of RTTL properties. Those properties not included in the set treated, can be checked by attaching a *watchdog* to the module. A watchdog is a non-invasive observer of the system. It can read the various system variables, but does not write to or destroy them.

The TTMchart hog2 in Figure 7 is an example of a watchdog that detects when the subformula $\Diamond\Box_{<7}(g_2 = 1)$ in the requirement R15 becomes true (i.e. when client 2 rebels). The chart hog2 || hog3 (where hog3 detects when client 3 rebels) is composed in parallel with the allocator and its environment. By model-checking the resulting chart for property

$$(r_1 = 1) \Rightarrow \Diamond_{\leq 7}(g_1 = 1 \vee r_1 = 0 \vee h_2 = 2 \vee h_3 = 2) \qquad \text{(EQ 6)}$$

FIGURE 7. Watchdog called "hog2" to detect a rebellious client



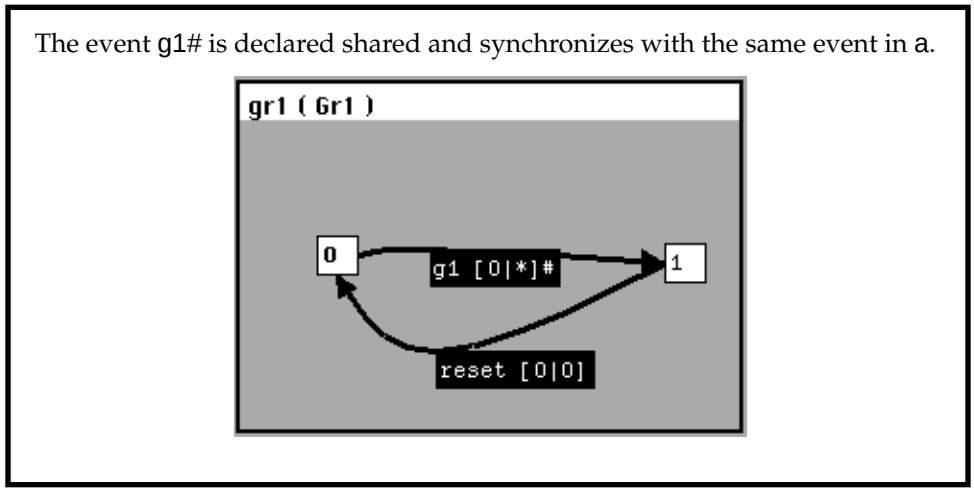
which is in a format that the verifier can check, we have thereby verified requirement R15. The activity variables h_2, h_3 correspond to the charts hog1 and hog2 respectively.

A disadvantage of adding a watchdog is that there is a corresponding increase in the size of the reachability graph that must be checked (although still smaller

than the size of the total graph). It is for this reason that the current verifier is being extended to verify arbitrary properties of real-time temporal logic [7].

The properties R9, R10 for the client, and R12, R13 for the allocator are almost in a format that the verifier can check. According to the definitions given in (EQ 5), changes in the release and grant variables must be detected by means of a watchdog. For example, to detect the condition gr_1 in the antecedent of R13, the watchdog in Figure 8 can be used. In the watchdog, the event `g1` which sets g_1 to

FIGURE 8. Watchdog to detect gr_i for requirement R13



one, is declared shared. Thus `g1` in the allocator of Figure 4 synchronizes with the corresponding component event `g1` in the watchdog. The property R13 can then be verified by checking

$$(\mathsf{Gr1} = 1) \Rightarrow (g_i = 1)\,\mathcal{W}\,(r_1 = 0 \wedge g_1 = 0) \qquad \text{(EQ 7)}$$

which is in a format that the verifier can deal with, where `Gr1` is the activity variable of the watchdog in Figure 8.

5.0 Conclusions

The main difficulty in modular specification is that modules are harder to specify than the total system. This is because the modular specification must take into account the environment (e.g. the rebellious clients in requirement R15).

It would be useful to have a tool that can check the validity of properties such as $R_A \wedge R_B \rightarrow R$ where R_A, R_B are the specifications of modules A and B, and R is the specification of the complete system. There is already a propositional theorem prover available for the untimed subset of RTTL, which can be extended to deal with timed properties.

The main advantage of modular specification and verification is that it significantly reduces the size of the state space that must be generated and checked. In general, the looser the coupling between the module and its environment the greater the efficiency of the modular check. The client module, which is very loosely coupled to its environment, can be checked in time an order of magnitude less than the total system. The allocator is more coupled to the environment (each client introduces one extra environmental variable), and hence the time to per-

form its check increases as more clients are added, although this modular check still takes substantially less time than checking the total system.

The use of assumption/guarantee specifications are useful for further reducing the size of the reachability graphs. Alternatively, the allocator can itself be decomposed into sub-modules.

Once the modular specification of the allocator is provided, the body can be developed and checked. If the body is changed, we need re-verify only the allocator module and not the complete system. For example, consider the alternative code in Figure 9 for the body of the allocator. The local variable *v* has been elimi-

FIGURE 9. Alternative code for the body of the allocator

```
loop forever do
  if r[i] = 1
     then do g[i] := 1; when r[i] = 1 then g[i] := 0 od
  i := (i mod 3) + 1
endloop
```

nated and the computation of the count variable *i* has been changed. This change can be verified by merely rechecking the new code against the modular specification of the allocator. There is no need to check the complete system.

The BUILD tool is written in Smalltalk and the VERIFY tool in Prolog. The current Prolog implementation is much slower than what can be achieved using more efficient languages. The VERIFY tool is currently being re-implemented in Smalltalk to allow model-checking of arbitrary branching time properties [7], which will eliminate the need for watchdogs in most cases. Arbitrary data types (including the ability to deal with the grant and request arrays such as *g* and *r*) will be supported.

The use of Smalltalk classes for representing data will provide rich flexible data definitions. However, the construction of reachability graphs will then not have the efficiency of a C language implementation restricted to efficient types for hashing (e.g. booleans, integers and unsigned 8 bit bytes). There may thus be a need to implement part of the verifier in C.

The modular methods discussed in this paper indicate that the StateTime toolset should be enhanced by allowing TTMcharts to have interface specifications. Modules can then be checked automatically for interface compatibility. Finally, the verifier should automatically build the environment (constrained where possible) in which the module must be checked.

6.0 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] 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.

[5] 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.

[6] 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.

[7] Gruden, C. "Automated Model Checking in the TTM/TCTL Framework." M.Sc., York University, Toronto, Canada, 1994 (to appear).

[8] Hall, A. "Seven Myths of Formal Methods." *IEEE Software*, Sep:11-19, 1990.

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

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

[11] 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.

[12] Holzmann, G. "Proving the Value of Formal Methods." In *FORTE'94 7th International Conference on Formal Description Techniques*, Berne, Switzerland, 1994 (to appear).

[13] 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.

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

[15] 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.

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

[17] 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).

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

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

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

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

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

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