Analyzing a Real-Time Program with Z

Jonathan Jacky *

Radiation Oncology, Box 356043
University of Washington
Seattle, WA 98195-6043

Abstract. Real-time behavior of a multi-tasking program running on a pre-emptive priority-based operating system is analyzed. The operating system and a collection of application tasks are modelled in Z. Real time is represented by an ordinary Z state variable. The model is adapted to a particular application by defining a state machine for each task and associating execution times with each state. The model is analyzed by exhaustive simulation with the SMV model checker. The state transitions described by Z operation schemas are implemented in the SMV programming language. Invariants, preconditions, and postconditions from the Z are translated to formulas in CTL, the SMV specification language. The SMV program is verified by checking these formulas. This detects coding errors in the SMV program and also reveals inconsistencies in the original Z where operation schemas are inconsistent with state invariants. The errors were corrected. Additional CTL formulas describe temporal properties that cannot be expressed directly in Z. The Z model is validated by checking an example SMV program with CTL formulas that confirm scheduling results from rate-monotonic analysis (RMA). Another application that does not satisfy the assumptions of RMA is analyzed, establishing that high-priority tasks cannot indefinitely delay low-priority tasks and real-time deadlines can be met.

1 Introduction

We wrote a control program for a radiation therapy machine, a safety-critical medical application [8, 11]. Our control program includes several concurrent tasks and meets real time deadlines. It runs on a commercial off-the-shelf (COTS) real time operating system.

We feel we should be able to answer questions such as these: Will each event be handled within its deadline? Is it possible for one high-priority task to indefinitely delay another low-priority task? In order to answer these questions, it is necessary to model the system as a collection of state transitions, and then analyze the model.

We chose ordinary Z [16] for our modelling notation. Many notations can describe state transitions; we use Z operation schemas for this. Moreover, Z state invariants provide essential information that cannot be expressed directly

* email jon@radonc.washington.edu

in notations that can only describe state transitions. The Z tool-kit provides the data types we need: sequences represent priority queues, relations (interpreted as finite state machines) represent the control structure of application tasks. Spivey modelled a multi-tasking operating system in Z [14]. However, he did not model the behavior of a complete system including application tasks, and did not model the passage of time or expiration of real-time deadlines.

We chose the SMV (Symbolic Model Verifier) [3, 13], a model checker, for our analysis tool. It performs completely automatic analysis of state transition systems. It can investigate behaviors that emerge when a collection of Z operation schemas works together. To check a Z specification using SMV, one implements the Z in the programming language of the checker. One also provides properties to check expressed in CTL (Computation Tree Logic), the checker's specification language. The checker performs all possible executions of the program, in effect performing an exhaustive simulation of the system. The checker verifies each property or outputs a counterexample (a trace of an execution that violates the property). An earlier experiment with Z and SMV was reported in [10].

Other formal notations provide built-in constructs for representing real time (several are reviewed in [2]), and there are special methods analyzing real-time scheduling [7, 12]. But sometimes the most general results can be obtained with the simplest notations and tools. Our results (section 8, table 2) show that ordinary Z and a general purpose model checker can support detailed analysis of a real-time program, providing quantitative answers to specific questions about performance.

2 Concurrency in Z

Before considering the tasking problem in detail, we represented our application in Z using the *interleaving* style proposed by Evans [4, 5]. All the data used by all tasks appears in the system state. All operations act on the same system state. The precondition of each operation schema is a guard: when the the precondition is satisfied, the operation is *enabled*. It is assumed that the underlying tasking system will provide *fairness*: every enabled operation will occur eventually. *Concurrency* becomes possible when more than one operation is enabled at the same time. In this situation it is assumed that all the enabled operations occur, but in nondeterministic order.

Up to a point, it is easy to design a multi-tasking system in Z using this style. One merely defines the state and the operations required by the application, taking care that the precondition of each operation causes it to become enabled when it is needed (examples based on our project appear in [8]).

However, difficulties can arise when the design is finally implemented on some real operating system. Operations are assigned to tasks which are scheduled by the operating system. It becomes necessary to write the code that is supposed to ensure fairness. For example, one task sets a semaphore to notify another task that new data are available. At this point, the operating system knows nothing of fairness. It deterministically selects the next task to run, based on task priorities

assigned by the programmer. If this machinery is not coded carefully, the system might not meet the fairness assumption. For example, high-priority tasks might be able to delay a low-priority task indefinitely.

To ensure that each event is handled in time, we need to solve a real-time scheduling problem: in some worst case situation, the sum of the execution times of several operations must be less than some interval. But what is the worst case situation, and which operations must be counted? Several of our application tasks are not periodic and they interact, so the answers to these questions are not obvious. Schedulability tests such as rate-monotonic analysis are not applicable [7, 12]. A particular scheduling problem from our application is posed in section 6 and analyzed in section 8.

3 Operating system and application tasks

We inferred our model of the operating system from the vendor's manual [17]. Tasks may be *ready* (to run) or *pending* (waiting for an event). Each task has a priority; for each priority, there is a queue of ready tasks in first-in, first-out (FIFO) order. The task that runs is at the head of the highest priority queue. A task which becomes ready can *pre-empt* a lower-priority task. A task becomes pending when it calls a function that waits for an event which has not yet occurred, and becomes ready again when the event occurs.

In our application, tasks read from devices attached to the controlled equipment and also from interprocess communication channels connected to other application tasks. Tasks become pending when they attempt to read from a device or an inter-task communication channel where input is not available. Pending tasks can be made to *time out*: they abandon the read operation if input has not appeared when a timeout interval expires. Tasks can also be made to simply wait for a timeout, in order to schedule periodic operations and give low-priority tasks an opportunity to run.

4 Z Model

In this section we present a generic model of a multi-tasking program with real-time deadlines and timeouts running on a pre-emptive priority-based operating system, interacting with an environment that provides events. We call this a *model* not a specification because it represents an already existing system and is intended to support analyses of that system.

We abstract away the application data; our model only represents control structure, tasking, synchronization, and the passage of time. We specialize the model for our application in section 5. The Z texts excerpted here [9] conform to the *Reference Manual* [16] and were type-checked [15].

4.1 Configuration

Our generic model can be specialized for any particular application by choosing values for the constants declared in this subsection (see section 5).

An application consists of a set of *tasks*. Each task executes in several *phases*. The sets of phases and tasks are fixed, but their actual contents depend on the application (this is indicated by the three dots in the otherwise formal text below). A task can only wait for an event at the end of a phase. In order to model pre-emption and the passage of time, it is necessary to represent the internal structure of the phases. Each phase occupies a contiguous range of machine addresses or *program counter* (PC) values. When a task executes, the program counter advances through a phase. Actually, execution may take different paths through a phase, depending on the values of data which we do not model here. The length of each phase given here can be taken to be the longest (worst case) path through the phase.

$$PC == \mathbb{N}$$

$$TASK ::= t_1 \mid t_2 \mid t_3 \mid \ldots$$

$$PHASE ::= p_1 \mid p_2 \mid p_3 \mid p_4 \mid p_5 \mid \ldots$$

$$\begin{array}{|l} phs : PC \pfun PHASE \\ start, end : PHASE \rightarrowtail PC \\ \hline \forall\, ph : PHASE \bullet \\ \quad start\ ph < end\ ph \land \\ \quad (start\ ph) \mathinner{..} (end\ ph) \subseteq \mathrm{dom}\ phs \land \\ \quad phs(\!|(start\ ph) \mathinner{..} (end\ ph)|\!) = \{ph\} \end{array}$$

The control structure of each task is given by the finite state machine *next* (where states correspond to phases). After the program counter reaches the end of a phase, the task begins executing at the beginning of another phase, determined by which event has occurred. Some phases send *signals* to a *destination* task (this models semaphores and other intertask communication). Input data only remains available during a *hold* time, after this deadline expires the data is lost. A timeout *interval* is associated with each phase that can time out. The *len* function returns the length of any phase (in program counter units, which are the same as clock ticks). Our *len*, *interval*, and *hold* determine the real-time behavior of the application. The ability to express arbitrarily complex timing behavior by defining interacting state machines distinguishes our model from other schedulability tests [7, 12] and makes exhaustive simulation necessary.

$$TIME == \mathbb{N}$$

$$EVENT ::= signal \mid data \mid timeout \mid expire$$

$$\begin{array}{|l} dest : PHASE \pfun TASK \\ len : PHASE \fun TIME \\ hold, interval : PHASE \pfun TIME \\ next : PHASE \fun EVENT \pfun PHASE \\ \hline len = (\lambda\, ph : PHASE \bullet end\ ph - start\ ph) \\ \mathrm{dom}\ hold = \{\, ph : PHASE \mid data \in \mathrm{dom}\ (next\ ph) \,\} \\ \mathrm{dom}\ interval = \{\, ph : PHASE \mid timeout \in \mathrm{dom}\ (next\ ph) \,\} \end{array}$$

Each task has a *priority*. Assignment of phases to tasks is determined by the phase initially executed by each task (a phase may be executed by more than one task).

$$PRIORITY == \mathbb{N}$$

$$\begin{array}{|l} init : TASK \longrightarrow PHASE \\ pri : TASK \longrightarrow PRIORITY \end{array}$$

4.2 System state

The operating system stores each task's *context*, represented here by its program counter. Tasks that are *ready* to run wait in a *queue*, one for each priority. The other tasks are *pending*, blocked at the end of some phase, waiting for an event. New *events* appear in the environment. They may be handled soon after they occur, or they may remain *unhandled* while the pertinent tasks are ready but unable to run. A key invariant holds that no pending tasks wait for these unhandled events. The *clock* indicates real time and a *timer* records when timeouts will expire for pending tasks. We also use the clock to model the expiration of real time *deadlines*.

RTSys

$$\begin{array}{|l} context : TASK \longrightarrow PC \\ ready, pending : \mathbb{P}\ TASK \\ queue : PRIORITY \longrightarrow \mathrm{iseq}\ TASK \\ \\ clock : TIME \\ events, unhandled : TASK \leftrightarrow EVENT \\ timer, deadline : TASK \pfun TIME \\ \hline \\ \mathrm{ran}\ context \subset \mathrm{dom}\ phs \\ \\ ready = TASK \setminus pending \\ \forall\, p : PRIORITY \bullet \mathrm{ran}\,(queue\ p) = \{\, t : ready \mid pri\ t = p \,\} \\ \\ \forall\, t : pending;\ ph : PHASE \mid ph = phs\ (context\ t) \bullet \\ \quad context\ t = end\ ph \land \neg\ (\exists\, e : \mathrm{dom}\,(next\ ph) \bullet (t, e) \in unhandled) \\ \\ \forall\, t : \mathrm{dom}\ deadline \bullet (t, data) \in unhandled \\ \forall\, t : \mathrm{dom}\ timer \bullet t \in pending \land timeout \in \mathrm{dom}\,(next\,(phs\,(context\ t))) \\ \\ \forall\, t : \mathrm{dom}\ timer \bullet clock \leq timer\ t \\ \forall\, t : \mathrm{dom}\ deadline \bullet clock \leq deadline\ t \end{array}$$

The system is *Running* when there is at least one ready task. The task at the head of the highest priority queue is *current* (running). Redundant state components name the program counter, phase and priority of the current task.

Running

$RTSys$
$pc : PC$
$phase : PHASE$
$current : TASK$
$priority : PRIORITY$

$ready \neq \varnothing$

$priority = max\,(pri(\!|ready|\!))$
$current = head(queue\,priority)$
$pc = context\,current$
$phase = phs\,pc$

In the initial state each task begins at the start of its first phase. The system is *Waiting* when there are no ready tasks. In the *NoEvent* state there are no new events in the environment. In the *NoTimeout* state no timeouts or deadlines have expired.

$$Init \mathrel{\widehat=} [\,Running \mid context = init\,;\,start\,]$$

$$Waiting \mathrel{\widehat=} [\,RTSys \mid ready = \varnothing\,]$$

$$NoEvent \mathrel{\widehat=} [\,RTSys \mid events = \varnothing\,]$$

NoTimeout

$RTSys$

$\forall\, t : \mathrm{dom}\, timer \bullet clock < timer\,t$
$\forall\, t : \mathrm{dom}\, deadline \bullet clock < deadline\,t$

4.3 Operations

There are thirteen state transitions, each modelled by a Z operation schema: *Wait*, *Compute*, *Continue*, *Switch*, *Block*, *Input*, *Timeout*, *Deadline*, *Expire*, *Defer*, *Enqueue*, *PreEmpt* and *Wakeup*. We call these *top-level* operations because none are included in the definitions of any other operations. The top-level operations define a state transition system which is complete and deterministic. Their preconditions are mutually exclusive and account for all states permitted by the invariant. Exactly one top-level operation is enabled in every possible state so the system cannot deadlock and we do not have to make any fairness assumptions.

Definitions of many top-level operations include these *building-block* operations: *Run*, *Phase*, *Pend*, *Event*, *Unhandled*, *Ready* and *Resume*. Building-block operations are not mutually exclusive; some top-level operations include more

than one building block. A *Run* operation occurs when there is at least one ready task and no events or timeouts occur. *Phase* is the specialization of *Run* that occurs when the program counter reaches the end of the phase. *Pend* is the specialization of *Phase* that occurs when no events are available for the current task. *Event* occurs when an event is available. *Unhandled* is the specialization of *Event* that occurs when the event is not handled. *Ready* is the specialization of *Event* that occurs when a pending task is waiting for the event. *Resume* is the specialization of *Ready* that occurs when the task that handles the event is (or becomes) current.

The top-level *Wait* operation occurs when no tasks are ready and no events occur. *Compute*, *Continue*, *Switch* and *Block* are specializations of *Run*. *Compute* advances the program counter of the current task, but not to the end of the phase. *Continue* is the specialization of *Phase* that occurs when the event needed by the current task has already occurred. *Switch* and *Block* are specializations of *Pend* that occur when another task is ready, or when no more tasks are ready, respectively. *Input*, *Timeout* and *Deadline* model the occurence of events in the environment. *Expire*, *Defer*, *Enqueue*, *PreEmpt* and *Wakeup* are specializations of *Event*. *Expire* models the expiration of a real-time deadline. *Defer* occurs when no pending task is waiting to handle the event. *Enqueue* is the specialization of *Ready* that occurs when the newly ready task does not have a higher priority than the current task. *PreEmpt* and *Wakeup* are specializations of *Resume* that occur when the newly ready task has a higher priority than the current task, or when there is no current task, respectively.

The *Wait* and *Compute* operations are of interest because they model the passage of time by advancing *clock*. They are the only operations that do so. Other operations are considered to take negligible time.

The *Wait* operation occurs when nothing else can happen: no tasks are ready and no events occur. The clock advances, but not past any timeout or deadline. Nothing else changes (indicated by the three dots in the otherwise formal text below).

Wait

$$\Delta RTSys$$

$$Waiting$$
$$NoEvent$$
$$NoTimeout$$

$$clock' > clock$$

$$context' = context$$
$$timer' = timer \land deadline' = deadline \land ready' = ready \land \ldots$$

Compute advances the program counter of the current task, but not past the end of the phase. The clock advances also, at the rate of one tick for each program counter step. Nothing else changes. *Compute* includes the *Run* building-block.

$$Run \mathrel{\widehat=} [\Delta RTSys; Running \mid NoEvent \land NoTimeout]$$

Compute

Run
$Running'$

$pc < end\ phase$

$pc < pc' \leq end\ phase$
$clock' = clock + (pc' - pc)$

$\{current\} \lhd context' = \{current\} \lhd context$
$timer' = timer \land deadline' = deadline \land ready' = ready \land \ldots$

It is helpful to collect formulas from related operations together. Table 1 is similar to the mode transition tables of the SCR notation [1]. It summarizes the top-level operations *Wait*, *Compute*, *Continue*, *Switch* and *Block*, along with their building-blocks *Run*, *Phase* and *Pend*. The *Run* building block is included in all the operations that appear below it in the table. Likewise, *Pend* is included in all operations below it. The second column in the table shows that the preconditions are mutually exclusive and cover all states where there are no inputs, events, or timeouts. A second table in [9] summarizes the other operations, whose preconditions cover the rest of the state space. There are 246 lines of Z in the entire model[2]

5 An application

We can specialize our model for any application by providing values for the constants declared in section 4.1. This example is based on our radiation therapy control program [8, 11].

There are four tasks: a *watchdog*, an interlock scanner *intlk*, and two controller tasks *ctlr*1 and *ctlr*2. The watchdog runs at highest priority, the interlock scanner is next highest, and the two controller tasks are lowest, at the same low priority. The watchdog and interlock tasks each execute just one phase, *synch* and *scan*, respectively. The controller tasks both execute the *poll* and *read* phases, starting with *poll*.

$$TASK ::= watchdog \mid intlk \mid ctlr1 \mid ctlr2$$

$$PHASE ::= synch \mid scan \mid poll \mid read$$

$$pri = \{watchdog \mapsto 3, intlk \mapsto 2, ctlr1 \mapsto 1, ctlr2 \mapsto 1\}$$

$$init == \{watchdog \mapsto synch, intlk \mapsto scan, ctlr1 \mapsto poll, ctlr2 \mapsto poll\}$$

The *watchdog* task runs periodically: it pends at *synch* for the *timeout* event. The *intlk* task only runs when it is signalled by another task: it pends at *scan* for the *signal* event. The *ctlr*1 and *ctlr*2 tasks each pend at *poll* and *read*. If *data* appears, they execute the *read* phase. If no data appears, a *timeout* event

[2] Nonblank lines output by running the Fuzz tool `-v` option [15] on [9].

Z operation	Precondition	Unchanged	Progress postcondition
Wait	*Waiting,* no input $NoEvent \wedge NoTimeout$	(All but *clock*)	$clock' > clock$
(Run)	*Running,* no input $NoEvent \wedge NoTimeout$		
Compute	$pc < end\,phase$	(All but $pc, clock$)	$pc' > pc$ $clock' = clock + (pc' - pc)$
Continue *(Phase)*	$pc = end\,phase$ $(\exists\, e : unhandled \ldots)$	*ready* *pending* *queue* *current* *priority*	*Running'* $e = (\mu\, e : unhandled \ldots)$ $unhandled' = unhandled \setminus \{(current, e)\}$ $phase' = next\,phase\,e$ $pc' = start\,phase'$ $deadline' = deadline \setminus \{(current, \ldots)\}$ $events' = events \cup \{(dest\,phase, signal)\}$
(Pend) *(Phase)*	$pc = end\,phase$ $\neg\, (\exists\, e : unhandled \ldots)$	*unhandled* *context*	$ready' = ready \setminus \{current\}$ $pending' = pending \cup \{current\}$ $queue'\,priority = tail\,(queue\,priority)$ $timer' = timer \oplus \{\ldots clock + interval\,phase\}$
Switch *(Pend)*	$ready \setminus \{current\} \neq \varnothing$		*Running'* $priority' = max\,(pri(\!\|ready'\|\!))$ $current' = head(queue\,priority')$ $pc' = context\,current$ $current' \neq current$ $priority' \leq priority$
Block *(Pend)*	$ready = \{current\}$		*Waiting'*

Table 1. Task state transitions: *Wait* and *Run* operations

will occur and they execute the *poll* phase again. At the end of its *synch* phase, the watchdog signals the *intlk* task, and at the end of their *read* phase, the two controller tasks signal the *intlk* task.

$$next\,synch = \{timeout \mapsto synch\}$$

$$next\,scan = \{signal \mapsto scan\}$$

$$next\,poll = next\,read = \{data \mapsto read, timeout \mapsto poll\}$$

$$dest = \{synch \mapsto intlk, read \mapsto intlk\}$$

$$interval = \{synch \mapsto period, poll \mapsto sample, read \mapsto sample\}$$

The watchdog task is periodic. The two controller tasks are recurrent: they run repeatedly, but not with any fixed period. Each of these tasks will pend for

some variable amount of time until data appears (nondeterministically) or the timeout expires. They will become unsynchronized with each other and with the watchdog. The interlock task runs whenever it is signalled by the periodic watchdog or by either recurrent controller task, and the interlock task can pre-empt or delay the controller tasks.

6 A real-time scheduling problem

Real-time deadlines arise in our example (section 5) because the information associated with a data event will be lost if the controller task does not begin to execute its read phase before the hold time expires. The controller task might miss the deadline because it cannot run while the higher priority watchdog and interlock tasks run. A controller task might also be delayed by the other controller task, which has the same priority.

Is it possible for a deadline to be missed? By applying reasoning similar to that of Liu and Layland [12], we can write some conditions which must be satisfied to prevent missing a deadline. The sum of the lengths of all the phases that might pre-empt a controller task must be less than the hold time. This includes the other controller task; moreover, the interlock task might run twice, signalled by both the watchdog task and the other controller, so this condition is $len\,synch + len\,read + 2 * len\,scan < hold$. Also, the period of the watchdog task must be long enough to prevent it from pre-empting a controller task repeatedly: $len\,read + 2 * len\,scan < period$. It seems obvious that these conditions are necessary to prevent missing a deadline, but are they sufficient? We should check our intuition with some analysis.

7 Analyzing the Z model using SMV

To analyze a Z model using the SMV model checker, one implements the Z in the SMV programming language and provides properties to check in CTL, the SMV specification language. The checker verifies each property or produces a counterexample (a trace of an execution that violates the property).

7.1 Writing the SMV program

The SMV program is, in effect, a simulation of a multi-tasking application running on a pre-emptive priority-based operating system. The Z texts presented in section 4, originally considered a model of that system, now serve as the *specification* for the simulation program. The specification could be implemented in any programming language, but there are two advantages to using SMV: the checker can verify that the program implements the specification, and the checker performs all possible executions of the program, achieving exhaustive simulation.

The SMV programming language [13] is much less expressive than Z. Its only data types are small integers (including booleans and enumerations). Our

specification cannot be implemented directly but must be simplified and made concrete.

To simplify, we abandon the generality of the model. We just implement one particular example, the application in section 5. Now we only need one priority queue that holds at most two controller tasks. The implementation is two-element array that indicates whether the two positions in the queue hold *ctlr*1, *ctlr*2, or neither.

To make the model concrete we represent all state variables as integers (including booleans and enumerations). The value of every variable must be kept small to limit the size of the state space so exhaustive simulation will be feasible. Program counters, clock and timers cannot be allowed to grow indefinitely. In our SMV program we reset the program counter to zero at the beginning of each phase so the program counters never grow larger than the length of the longest phase. Each phase is only a few units long, just long enough to admit the possibility of being pre-empted by higher priority tasks. We have a separate clock for each timeout and deadline, which remains at zero except when its timeout or deadline is counting up (several might be counting at once).

We defined SMV symbols that implement redundant state components such as *current* and predicates such as *Running*. This makes the SMV program shorter and helps it resemble the Z model more closely, without enlarging the state space.

Most of the implementation is not difficult because our Z model is already expressed in an operational style: the new value of every state variable after each transition appears by itself on one side of an equation. These equations are translated to SMV assignments. For each variable there is an SMV case statement. For each Z operation where that variable changes value, there is a case branch that assigns the new value, guarded by that operation's precondition.

The checker itself provides input to our SMV program nondeterministically. When the *ctlr*1 or *ctlr*2 task is pending, the checker may (or may not) provide a *data* event to that task at each execution step. The checker explores all possible execution sequences: the one where the *data* event occurs on the first execution step after *ctlr*1 pends at *poll*, the one where the *data* event occurs on the second execution step, the one where the *data* event never occurs so the *timeout* event occurs instead, etc. As a result, the checker considers all possible interleavings of the tasks.

Our SMV program [9] (excluding CTL formulas that express properties to be checked) is 423 (nonblank, noncomment) lines long. Fig. 1 shows excerpts that deal with *context watchdog*, *timer watchdog*, and *clock* (compare to the *Wait*, *Run* and *Compute* schemas in section 4.3). We used the smallest values for lengths of phases and intervals that still reveal the properties of interest (section 8). With these values, SMV reports that the program has 30062 reachable states in a space of more than 10^{23} states (2^{78}, or 78 state bits). SMV uses a data structure called a *binary decision diagram* (BDD) to represent the transition relation defined by a program [3]. It encodes the next-state relation implicitly defined by all thirteen top-level operation schemas described in section 4.3. SMV reports that the BDD for our program contains 24594 nodes.

The checker's performance depends on the ordering of variable declarations in the program and on several command line parameters such as cache size [13]. We adjusted these by trial and error to achieve acceptable performance (on a Hewlett-Packard J282 workstation with a PA8000 processor running at 180 MHz). Most CTL formulas can be verified in a few minutes, using about ten megabytes of memory.

7.2 Refinement

The development of the SMV program can be easily (but tediously) formalized as a *refinement*. Refinement is a method for establishing formally that one (concrete) specification is an implementation of another (more abstract) specification. Here we use refinement to show how expressions in SMV and formulas in CTL can be identified with expressions and predicates in Z.

The concrete specification (below) closely resembles our SMV program: the set of *ready* tasks is represented by an array of booleans *cready*, the tasks themselves are the array indices $CTASK1$, and the priority queue for the two controller tasks is the two-element array *cqueue*, where the first element is the head of the queue. The invariant shows how the current task is selected, based on the priorities of the ready tasks. The corresponding predicates in *Running* are $priority = max\,(pri(\!|ready|\!))$ and $current = head\,(queue\,priority)$.

$$BOOLEAN == \{0, 1\}$$

$$cnone == 0;\ cctlr1 == 1;\ cctlr2 == 2;\ cwatchdog == 3;\ cintlk == 4$$

$$CTASK == \{cnone, cctlr1, cctlr2, cwatchdog, cintlk\};$$

$$CTASK1 == CTASK \setminus \{cnone\}$$

CRTSys

$$ccurrent : CTASK$$
$$cready : CTASK1 \longrightarrow BOOLEAN$$
$$cqueue : \{1, 2\} \longrightarrow \{none, cctlr1, cctlr2\}$$
$$\ldots$$

$$ccurrent = cwatchdog \Leftrightarrow cready\,cwatchdog = 1$$
$$ccurrent = cintlk \Leftrightarrow cready\,cwatchdog = 0 \land cready\,cintlk = 1$$
$$ccurrent = cqueue\,1 \Leftrightarrow cready\,cwatchdog = 0 \land cready\,cintlk = 0$$
$$\ldots$$

Now we can relate the two specifications. The function *abs* associates each concrete task with its abstract counterpart, and the abstraction schema *Abs* relates the concrete state *CRTSys* to the abstract state defined in *RTSys* and *Running*.

$$abs == \{cctlr1 \mapsto ctlr1, cctlr2 \mapsto ctlr2, cwatchdog \mapsto watchdog, cintlk \mapsto intlk\}$$

```
MODULE main

VAR
   context: array 1 .. 4 of {0,1,2,3,4}; -- program counter indexed by task
   tclock: {0,1,2,3,4,5,6,7};   -- clock for watchdog timer
   timer: boolean;              -- true if watchdog timer counting

DEFINE
   watchdog := 3;               -- index into context array
   lsynch := 1;                  -- watchdog task, len synch in the Z

   Running :=  ready[watchdog] | ready[intlk] | ready[ctlr1] | ready[ctlr2];
   preRun := Running & NoInput & NoEvent & NoTimeout;
   preWait := Waiting & NoInput & NoEvent & NoTimeout;
   preCompute := preRun & InPhase;

ASSIGN
   init(context[watchdog]) := 0;   -- program counter at start of phase
   init(timer) := 0;               -- watchdog timer is not counting
   init(tclock) := 0;              -- clock is reset

   next(context[watchdog]) := case
        preCompute & current = watchdog
           & context[watchdog] < lsynch: context[watchdog]+1; -- Compute
        preResumeWD: 0;                                       -- Resume
        1: context[watchdog];                                 -- else no change
   esac;

   next(tclock) := case
        preWait & timer & tclock < period: tclock + 1;        -- Wait, tick
        preCompute & timer & tclock < period: tclock + 1;     -- Compute, tick
        preTimeoutWD: 0;                                      -- Timeout, reset
        1: tclock;                                            -- else no change
   esac;

   next(timer) := case
        prePend & current = watchdog: 1;                      -- (Pend), set
        preTimeoutWD: 0;                                      -- Timeout, reset
        1: timer;                                             -- else no change
   esac;
```

Fig. 1. Excerpts from SMV program

Abs

$RTSys$
$Running$
$CRTSys$

$ccurrent \neq cnone \Rightarrow current = abs\ ccurrent$
$ready = \{ t : CTASK1 \mid cready\ t = 1 \bullet abs\ t \}$

$\mathrm{dom}\ queue = \{1, 2, 3\}$
$queue\ 1 = (cqueue \upharpoonright CTASK1) \fatsemi abs$
$queue\ 2 = \textbf{if}\ cready\ cintlk = 1\ \textbf{then}\ \langle intlk \rangle\ \textbf{else}\ \langle\rangle$
$queue\ 3 = \textbf{if}\ cready\ cwatchdog = 1\ \textbf{then}\ \langle watchdog \rangle\ \textbf{else}\ \langle\rangle$
$\ldots$

We can define a concrete operation for each abstract operation. This fragment of the concrete *CPend* operation shows what happens when the controller task at the head of the priority queue becomes *pending*: the next array element advances to the head of the queue.

CPend

$\Delta CRTSys$

$(ccurrent \notin \{cctlr1, cctlr2\} \land cqueue' = cqueue) \lor$
$(ccurrent \in \{cctlr1, cctlr2\} \land cqueue'\ 1 = cqueue\ 2 \land cqueue'\ 2 = cnone)$
$\ldots$

The corresponding predicate in the abstract *Pend* operation is simply $queue'\ priority = tail\ (queue\ priority)$. The refinement laws in [16] can be used to check that *CPend* is a correct implementation of *Pend*, given the relations defined by *Abs*.

7.3 Analyzing the SMV program with CTL

Properties to check are expressed in CTL (Computation Tree Logic) [3]. CTL formulas are about SMV program *states* and *paths* (sequences of states that occur when the SMV program executes). CTL state formulas are like Z predicates. Path formulas provide temporal operators: if p is a state formula, `G` p means that condition p is true in all states on a path. `F` p means that p eventually becomes true. `X` p means that p is true in the next state. CTL also provides quantifiers: `A` (all paths) and `E` (some paths).

`AG` p expresses *safety*: p is invariant (is true on all paths, in all states). `AG(`*pre* `->` `AX` *post*`)` means that if *pre* is true in a state, then *post* is true in the next state. These two CTL formulas correspond to predicates in Z state schemas and operation schemas, respectively. Other CTL formulas can express behaviors that emerge when sequences of operations are executed. `EF` p expresses *liveness*: p can occur, p is reachable (is true on some paths, eventually). `AG AF` p expresses that condition p is cyclic or recurrent (on all paths p is always true eventually,

p occurs "infinitely often"). There are no built-in constructs corresponding to path formulas in ordinary Z, they can only be expressed after explicitly defining a next-state relation and sequences of states [6].

7.4 Verifying the SMV program

We verified our SMV program by the method of Atlee and Gannon [1]. For each Z operation, we wrote two kinds of CTL formulas: `EF` *pre* checks that the operation occurs and `AG(`*pre* `->` `AX` *post*`)` checks that it has the intended effect. It is necessary to check both formulas; if the `EF` liveness property is false, the `AG` correctness property will be vacuously true.

These CTL formulas are a machine-checkable formal specification for the SMV program. At first the checker found counterexamples to several formulas, which we traced to trivial coding errors. After we corrected the errors, all formulas were verified. This is a machine-checked proof that the SMV program correctly implements its CTL specification. The close resemblance between the CTL formulas and their Z counterparts, supported by the refinement arguments of section 7.2, suggest that the program correctly implements the Z specification as well.

We wrote 104 CTL formulas to verify the SMV program. The checker verified them in 24 minutes, using 10 megabytes.

7.5 Checking the Z specification

The preceding subsections establish that our SMV program is a correct implementation of the operation schemas in the Z specification. Therefore properties which are false in our program are not universally true in our specification. This means that checking our program can reveal errors in the Z. In particular, if a CTL property which is a translation of a (putative) Z invariant is not invariant in our SMV program, then the Z specification is inconsistent.

We translated the invariants of the *RTSys* and *Running* state schemas into CTL `AG` *inv* formulas. At first the checker found counterexamples to several formulas, revealing that the *RTSys* invariants were too strong[3]. After revising (weakening) the invariants to those shown in section 4.2, the checker verified all 34 of these formulas in one minute, using 9 megabytes.

Our SMV program is only one example implementation of our Z specification, so properties which are true in this program may not be universally true of our specification. There could be additional errors in our corrected Z specification which are not revealed by this example.

7.6 Validating the Z specification

We must show that the Z specification actually expresses the properties we intend. Checking the formula `!(EF` p`)` causes the checker to generate a counterexample showing the execution sequence from the initial state to a state that

[3] A reviewer of an early version of this paper also detected some of these errors.

satisfies p (see Table 2). This is a way of "running" the SMV program. Used in this way, the SMV program serves as a kind of animation of the Z specification. The counterexamples confirm that the intended behaviors occur.

If the Z specification is valid, our implementation should be able to reproduce examples of well-known results from real-time scheduling theory. Rate-monotonic analysis (RMA) applies in the special case where each task has a single phase, tasks do not signal each other, events occur on a strictly periodic schedule, and the deadline equals the period [7, 12]. A simple example with two tasks appears in [12]. We represented this example, and revised our SMV program to deliver events on a periodic schedule instead of nondeterministically. The program reproduced the results shown in Figs. 2a, 2b, and 2c in [12] and discussed in the accompanying text.

8 Analyzing a real-time scheduling problem

In our application (section 5), high-priority tasks must not be able to delay low-priority tasks indefinitely. All four tasks should be recurrent: each should execute repeatedly, though not necessarily with a fixed period. This property cannot be expressed directly in Z because each cycle involves the sequential execution of several operations. We wrote four CTL formulas to check recurrence: `AG AF` $current = watchdog$ etc. The checker verified all four in less than two minutes, using 9 megabytes.

The application will miss a real time deadline if a data event remains unhandled when its deadline expires. We checked `!(EF` p`)` formulas which express that this condition is unreachable. To check this, it is necessary to assign values to the lengths of all the phases and intervals. We made the phases as short as possible, while still allowing for pre-emption: $len\,synch = 1$ and $len\,scan = len\,read = len\,poll = 2$. By trial and error, we found $period = 10$ (the watchdog task period) and $hold = 12$ (the data hold time) were the shortest intervals where the deadlines could always be met (the CTL formula was verified). Reducing the value of *period* or *hold* by one makes it possible for the deadline to be missed[4].

When the data hold time is reduced to 11, the checker takes two minutes and 10 megabytes to generate a counterexample of 57 states that shows how the deadline is missed (Table 2): The read timeout for the *ctlr*2 task expires (state 28), but then a data event occurs anyway (state 30). After *ctlr*2 handles the timeout (state 40), it handles the data event also (state 43), and consequently signals the *intlk* task to run as well (state 47). All this delays *ctrl*1 so it misses its deadline (state 57). We did not anticipate that a data event would appear after a read timeout. In fact this is unlikely, but the checker shows the consequences of the SMV program we actually wrote, not what we expected.

[4] We kept $sample = 5$ (the controller task read timeout period).

St.	Clk	Operation	Task	Comment
1		(*Init*)	*watchdog*	Initial state
⋮		⋮	⋮	⋮
28		*Timeout*	*intlk*	*timeout* event for *ctlr*2 while *intlk* is current
29		*Enqueue*		*timeout* event queued for *ctlr*2
30		*Input*		*data* event for *ctlr*2
31		*Enqueue*		*data* event queued for *ctlr*2
32		*Switch*	*ctlr*1	*intlk* pends, *ctlr*1 becomes current
33		*Switch*	*ctlr*2	*ctlr*1 pends, *ctlr*2 becomes current
34		*PreEmpt*	*intlk*	*intlk* pre-empts *ctlr*2, handles signal from *ctlr*1
35		*Input*		*data* event for *ctlr*1
36	0	*Enqueue*		*data* event queued for *ctlr*1
37	1	*Compute*		*intlk* computes in *scan* phase
38	2	*Compute*		*intlk* computes in *scan* phase
39	2	*Switch*	*ctlr*2	*intlk* pends, *ctrl*2 becomes current
40	2	*Continue*		*ctrl*2 handles *timeout* event
41	3	*Compute*		*ctlr*2 computes in *poll* phase
42	4	*Compute*		*ctlr*2 computes in *poll* phase
43	4	*Continue*		*ctrl*2 handles *data* event
44	5	*Compute*		*ctlr*2 computes in *read* phase
45	6	*Compute*		*ctlr*2 computes in *read* phase
46	6	*Switch*	*ctlr*1	*ctlr*2 pends, *ctlr*1 becomes current
47	6	*PreEmpt*	*intlk*	*intlk* pre-empts *ctlr*1, handles signal from *ctlr*2
48	7	*Compute*		*intlk* computes in *scan* phase
49	8	*Compute*		*intlk* computes in *scan* phase
50	8	*Timeout*		*timeout* event for *watchdog*
51	8	*PreEmpt*	*watchdog*	*watchdog* pre-empts *intlk*, handles timeout
52	9	*Compute*		*watchdog* computes in *synch* phase
53	9	*Switch*	*intlk*	*watchdog* pends, *intlk* becomes current
54	9	*Defer*		*intlk* defers handling signal from *watchdog*
55	9	*Continue*		*intlk* handles signal from *watchdog*
56	10	*Compute*		*intlk* computes in *scan* phase
57	11	*Compute*		*ctlr*1 misses deadline

Table 2. Counterexample showing missed deadline

9 Conclusion

Ordinary Z and a general purpose model checker can support detailed analyses of a real-time program, providing quantitative answers to specific questions about performance. Exhaustive simulation with a model checker can be a practical alternative when other schedulability tests are not applicable. Our simulations used modest computing resources (two minutes, ten megabytes). Much larger simulations appear feasible.

The SMV programming language is so low-level that a formal specification

at the level of Z operation schemas is a practical necessity. It would not be reasonable to write an SMV program by intuition and debug it by trial and error. Our program was automatically verified by checking CTL formulas based on the Z. The verification quickly revealed coding errors that could have been quite difficult to detect and correct.

SMV can be quite useful for checking and exploring Z specifications. It can investigate behaviors that emerge when sequences of operations are executed. By providing exhaustive simulation and counterexample generation, it combines some of the advantages of theorem proving and animation.

References

1. Joanne M. Atlee and John Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.
2. Zhou Chaochen, C. A. R. Hoare, and Anders P. Ravn. A calculus of durations. *Information Processing Letters*, 40:269 – 276, 1991.
3. E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In J. W. De Bakker, W.-P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency*, pages 124–175, Noordwijkerhout, The Netherlands, 1993. REX School/Symposium, Springer-Verlag. Lecture Notes in Computer Science, vol. 803.
4. Andy S. Evans. Specifying and verifying concurrent systems using Z. In Maurice Naftalin, Tim Denvir, and Miquel Bertran, editors, *FME '94: Industrial Benefit of Formal Methods*, pages 366–380. Springer-Verlag, 1994. (Lecture Notes in Computer Science number 873).
5. Andy S. Evans. Visualizing concurrent Z specifications. In J. P. Bowen and J. A. Hall, editors, *Z User Workshop, Cambridge 1994*, pages 269–281. Springer-Verlag, 1994. (Workshops in Computer Science).
6. Andy S. Evans. An improved recipe for specifying reactive systems in Z. In Jonathan P. Bowen, Michael G. Hinchey, and David Till, editors, *ZUM '97: The Z Formal Specification Notation*, pages 275 – 294. Tenth International Conference of Z Users, Springer-Verlag, 1997. Lecture Notes in Computer Science 1212.
7. C. J. Fidge. Real-time schedulability tests for preemptive multitasking. *Real-Time Systems*, 14(1):61 – 93, 1998.
8. Jonathan Jacky. *The Way of Z: Practical Programming with Formal Methods.* Cambridge University Press, 1997.
9. Jonathan Jacky. Analyzing a real-time program with Z and SMV. Technical Report 98-06-01, Department of Radiation Oncology, University of Washington, Box 356043, Seattle, Washington 98195-6043, USA, June 1998.
10. Jonathan Jacky and Michael Patrick. Modelling, checking, and implementing a control program for a radiation therapy machine. In Rance Cleaveland and Daniel Jackson, editors, *AAS '97: Proceedings of the First ACM SIGPLAN Workshop on Automated Analysis of Software*, pages 25 – 32, 1997.
11. Jonathan Jacky, Jonathan Unger, Michael Patrick, David Reid, and Ruedi Risler. Experience with Z developing a control program for a radiation therapy machine. In Jonathan P. Bowen, Michael G. Hinchey, and David Till, editors, *ZUM '97: The*

Z Formal Specification Notation, pages 317 – 328. Tenth International Conference of Z Users, Springer-Verlag, 1997. Lecture Notes in Computer Science 1212.
12. C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46 – 61, 1973.
13. K. L. McMillan. The SMV system. Carnegie-Mellon University, February 2 1992. (draft).
14. J. M. Spivey. Specifying a real-time kernel. *IEEE Software*, 7(5):21–28, September 1990.
15. J. M. Spivey. *The FUZZ Manual*. J. M. Spivey Computing Science Consultancy, Oxford, January 1991. Second Printing.
16. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, New York, second edition, 1992.
17. Wind River Systems, Inc., Alameda, California. *VxWorks Programmer's Guide 5.3.1*, 1997.

This article was processed using the LaTeX macro package with LLNCS style