Z model of FreeRTOS

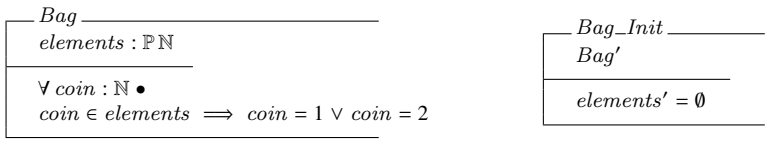
1 Z - Modelling Language

The Z notation is a formal specification language used for describing and modelling computing systems. Z notation was proposed by Abrial et al in 1974. Tools are available for simulation and validation of the model specified in Z. The Z notation is based upon set theory and mathematical logic. The mathematical logic is a first-order predicate calculus.

Mathematical objects and their properties can be collected together in *schemas* - patterns of declarations and constraints [2]. The schema language can be used to describe the state of a system, and the ways in which a state may change. The state of the system can be modelled in Z on a component basis in a bottom up fashion.

A schema specification modelling a component of the system state includes two parts - declaration part which specifies the fields or variables of the component (for example the declaration part of the schema Bag in Figure 1(a) specifies $elements$ as a set of natural numbers) and a constraint part which specifies the invariants on the component (for example the constraint part of the schema Bag in Figure 1 specifies that each member in the set $elements$ is either 1 or 2).

A state change in the system is modelled with a schema called *operation schema*. An operation schema specifies the state change w.r.t. the schema modelling the system state. The state change is specified as a relation between the pre-state and post-state of the schema modelling the system state. We call the



(a) (b)

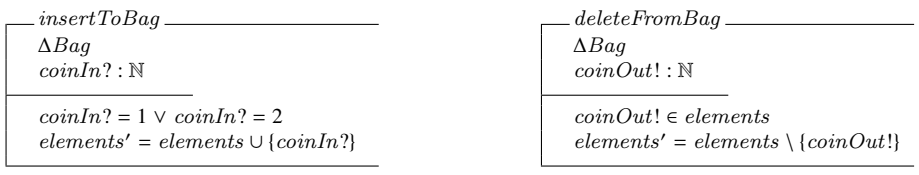
Figure 1: (a) Z schema modelling the object Bag and (b) schema to initialise Bag.

schema modelling the system state as the *operand schema* for the concerned operation schema.

The structure of an operation schema is similar to an operand schema where the declaration part declare the pre and post state of the operand schema and the constraint part specify the state change w.r.t. the operation. The state change is specified as a relation between *pre* and *post* states of the operand schema w.r.t. the concerned operation. A predicate defining the relationship between the pre-state and post state of a field is called a *before-after* predicate.

According to the convention, unprimed variables of the operand schema represent the pre state and the primed variables represent the post state w.r.t. the operation. If S is the name of the operand schema, then S and S' in the declaration part of an operation schema declare the pre and post state respectively.

A Z schema modelling a component in the system needs to have an initialization. This is done by an operation schema. Initialization gives a valuation to the fields of the operand schema in the initial state of the system. Initialization schema do not require a pre state declaration as there is no state before initializing the system. Figure 1(b) shows the initialization for the component schema Bag. It specifies that in the initial state the set $elements$ is empty.



(a) (b)

Figure 2: (a) Z schema specifying insertion operation for Bag and (b) Z schema specifying deletion operation for Bag.

The operation schema in Figure 2(a) specifies the insertion operation supported by Bag. There is a shortcut in Z to declare the pre and post state for an operation schema in a single step. If S is the name of the operand schema, then ΔS declares both the pre-state and post-state for the operation.

operation schema includes (in general) a pre-condition in addition to the before-after predicates in the constraint part. A pre-condition is a predicate on the pre-state (unprimed variables) and input. In $insertToBag$ the pre-condition demands that the number to be inserted is either 1 or 2.

According to the convention, field declaration in which a field name ending in ?/! represents input/output to the operation schema.

The before-after predicate of the operation schema in Figure 2 (a) specifies

that in the post state of the system, the set $elements$ will contain the new item ($coinIn?$) in addition to the elements present in the pre-state.

The post state of an operation is guaranteed to satisfy the before-after predicates if the operation is invoked within the precondition of the operation. The post state is undefined if the operation is invoked outside its precondition.

2 Refined Z Model M_2 for the portable layer of FreeRTOS.

The initial Z model M_1 is a deterministic model capturing the requirements of the scheduler related operations in the portable layer of FreeRTOS. Even though the model M_1 is very close to the existing implementation, we need to refine this model into an Z model say M_2 such that there is a one-to-one correspondence between the Z model and the existing implementation in terms of the data structures and APIs. Such a correspondence is mandatory for applying our theory of refinement.

The dissimilarities of the model M_1 with the existing implementation is given below.

1. The set of delayed tasks is modeled as a singe list in M_1 while it is implemented as two lists viz. *pxDelayedTaskList* and *pxOverflowDelayedTaskList* in the existing implementation.

2. The blocked tasks are modeled in M_1 as a priority queue and the task are maintained in the non-increasing order of priority values. On the other hand blocked lists are maintained in the non-decreasing order of *inverted priority* values. This is to share the same linked list operation for different priority queues.

We develop an Z model M_2 from M_1 to handle the above mentioned dissimilarities. The basic changes/additions made to the model M_1 to get the model M_2 is given below.

1. The set of delayed task is modeled as two task lists in M_2 viz. *delayedZ2* and *oDelayed*. In fact the single list *delayed* in M_1 is divided into two lists in M_2. The prefix of the list *delayed* in M_1 with value of time to awake less than or equal to *maxNumVal* is the list *delayedZ2* in M_2 and its suffix is the list *oDelayed* in M_2.

2. The value of time to awake is represented in the modulo ($maxNumVal$ + 1) system in *oDelayed* of M_2 while the corresponding time to awake value is ($maxNumVal$ + 1) more in the suffix of the list *delayed* in M_1.

3. The set of blocked tasks is modeled as a priority queue and the tasks are maintained in the non-decreasing order of *inverted priority* values. Even though the invariant for ensuring this order is equivalent to the corresponding invariant in M_1, writing it in the new form supports easy verification of the refinement conditions in this regard.

The heart of the portable layer of FreeRTOS is the scheduler related data structures and associated operations. FreeRTOS employs priority based scheduling policy and the user can configure the scheduler as *preemptive* or *non-preemptive*. Task is the pivotal object in FreeRTOS. A task can be in one of the states shown in Figure 3.

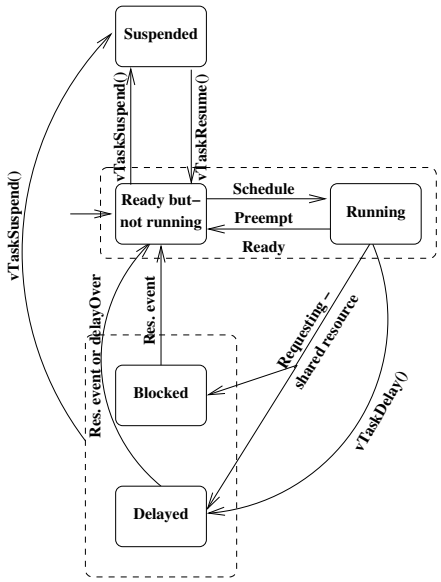


Figure 3: Different states for tasks in FreeRTOS.

The maximum time to wait for a resource is always an argument to the API for requesting the allocation of a shared resource. If the requested resource is not available, the requesting task will be added to the waiting queue of the resource as well as to the queue of delayed tasks. Such a blocked task will be moved back to ready either when the resource becomes available (*resource event*) or when the maximum time to wait expires (*delayOver*).

Following Z construct defines the type, *TASK* which is used to declare various tasks in the model and a free type *BOOL* with two values/constants in the domain - *TRUE* and *FALSE*.

$$[TASK]$$
$$BOOL ::= TRUE \mid FALSE$$

FreeRTOS uses a set of configurable macros which defines some system level properties like the maximum priority, type of the scheduler (preemptive or not) etc. The implementation of the FreeRTOS scheduler assumes that there is always a ready task to schedule. To ensure this a special task called *idle* is created when the scheduler is started. There are some useful operations which can be performed by the idle task like freeing the resources allotted to the deleted tasks.

Following Z construct defines the global constants *idle* and *null* modelling the idle task and special task which is assumed as the running task until the scheduler is started.

$$idle : TASK$$
$$null : TASK$$

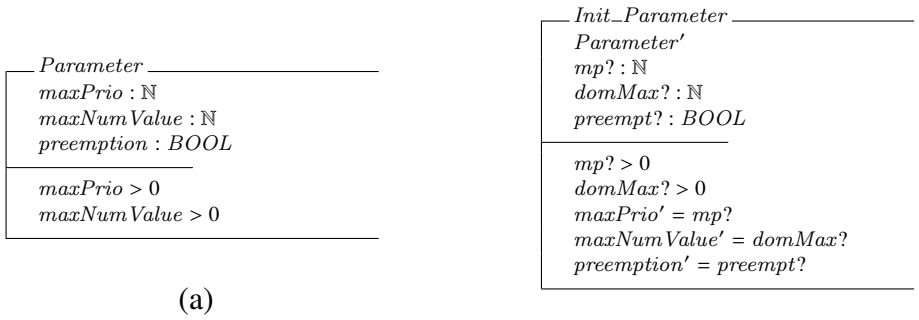


Figure 4: (a) Z schema,$Parameter$ modelling configurable macros in FreeRTOS and (b) Z schema to initialise $Parameter$.

We want to do a parameterized verification of FreeRTOS w.r.t. the constants defined in the files "FreeRTOSConfig.h" and "portmacro.h". Hence we model these constants as global variables which can be initialised to any value in the respective domain. Initial values for these variables are specified as arguments to the operation schema to initialise the system.

The schema in Figure 4(a) models the configurable macros in FreeRTOS. Where *maxPrio*, *maxNumValue* and *preemption* represent the maximum priority value, the maximum numeric value in the domain for the objects like system clock (*tickCount*) and the scheduling policy respectively in FreeRTOS. The value for *maxNumValue* can be initialised to one of the values given in the constraint section depending on the maximum value supported by the concerned type in the target machine. The schema in Figure 4(b) specifies the initialization for *Parameter*. The initial values are specified as parameters to this schema. No operation schema in

this model modifies any of these variables after initialisation and is assured by the predicates in the operation schemas.

In the existing implementation of FreeRTOS, a task can have priority in the closed interval [0, $maxPrio - 1$]. But for easy specification , a task is assumed to have priority in the closed interval [1, $maxPrio$].

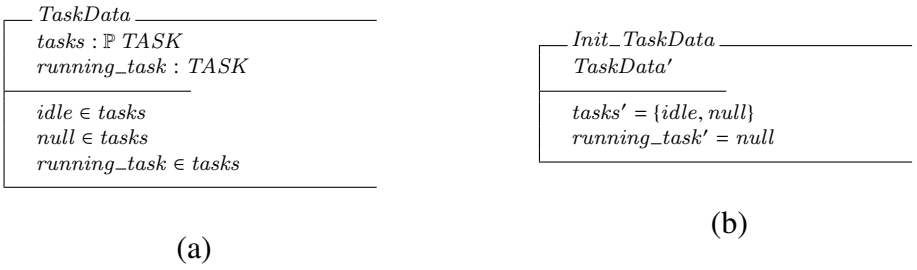


Figure 5: (a) Z schema, $TaskData$ modelling the task set in the system and (b) Z schema to initialise $TaskData$.

Schema in Figure 5(a) models the set of all tasks in the system. This schema is corresponding to the *Task Control Block* in FreeRTOS. In the declaration section, *tasks* represents the set of all tasks and *running_task* represents the task in execution. Constraint section specify that *idle*, *null* and *running_task* are valid tasks in the system. The initialization schema for *TaskData* in Figure 5(b) specifies that when the system is initialized, the set *tasks* is the set{idle,null} and *running_task* is *null*.

Figure 6 shows generic definition schemas which can be used to extract the sequence of first elements and sequence of second elements respectively from a given sequence of pairs ($TASK \times \mathbb{N}$). For example

$$seqFirst(\langle(task_1, 1), (task_2, 2), (task_3, 3)\rangle) = \langle task_1, task_2, task_3\rangle \text{ and}$$
$$seqSecond(\langle(task_1, 1), (task_2, 2), (task_3, 3) = \langle 1, 2, 3\rangle.$$

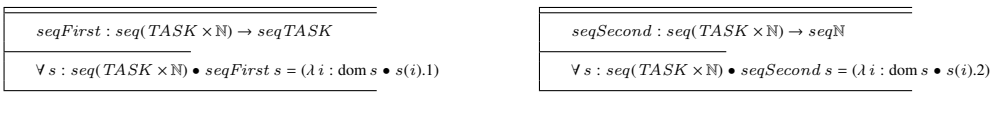


Figure 6: Generic definition schemas (a) to find a sequence of first elements from a given sequence of pairs ($TASK \times \mathbb{N}$) and (b) to find a sequence of second elements from a given sequence of pairs ($TASK \times \mathbb{N}$).

FreeRTOS maintains various task lists to implement the different states for tasks shown in Figure 3. The state $ready$ is implemented as a set of FIFO task lists indexed by priority. In FreeRTOS, running task is the last task in the highest index nonempty ready queue. The state $blocked$ is implemented as priority queue and the order is determined by priority. The state $delayedZ2$ is implemented as priority queue and the order is determined by the time to awake. The state $suspended$ is implemented as an unordered task list.

$ListData$

$ready$: seq (iseq $TASK$)
$blocked$: iseq $TASK$
$delayedZ2$: seq $TASK \times \mathbb{N}$
$oDelayed$: seq $TASK \times \mathbb{N}$
$suspended$: $\mathbb{P}\ TASK$

(ran $\frown$/(ran $ready$)) $\cap$ (ran $blocked$ $\cup$ ran $seqFirst(delayedZ2)$ $\cup$ ran $seqFirst(oDelayed)$ $\cup$ $suspended$) = $\emptyset$
$\forall\, i, j$: dom $ready$ | $i \neq j \bullet$ (ran $ready(i)$ $\cap$ ran $ready(j)$) = $\emptyset$
($suspended$ $\cap$ (ran $seqFirst(delayedZ2)$ $\cup$ ran $seqFirst(oDelayed)$)) = $\emptyset$
(ran $seqFirst(delayedZ2)$ $\cap$ ran $seqFirst(oDelayed)$) = $\emptyset$
$\forall\, i, j$: dom $delayedZ2$ | $(i < j) \bullet delayedZ2(i).2 \leq delayedZ2(j).2$
$\forall\, i, j$: dom $oDelayed$ | $(i < j) \bullet oDelayed(i).2 \leq oDelayed(j).2$
$\forall\, i, j$: dom $delayedZ2$ | $(i \neq j) \bullet delayedZ2(i).1 \neq delayedZ2(j).1$
$\forall\, i, j$: dom $oDelayed$ | $(i \neq j) \bullet oDelayed(i).1 \neq oDelayed(j).1$
dom $ready = 1 \cdot\cdot maxPrio$

Figure 7: Z schema,$ListData$ modelling the task lists in the system.

Schema in Figure 7 models different high level tasks lists in the system. The state $running$ is modelled using the field $running\_task$ in the schema $TaskData$. The set of ready tasks in the system, *ready* is modelled as a sequence of injective sequences. First injective sequence in *ready* is the sequence of tasks with priority 1, second sequence in *ready* is the sequence of tasks with priority 2 and so on. Each sequence is maintained in FIFO order.

In Z, $iseq$ represents a sequence in which no element appears more than once. Now the ready list is modelled very similar to the array of lists in FreeRTOS implementation.

Each resource is associated with one or two waiting queues of tasks. The state of a task is said to be *blocked* when it is present in any of these waiting queues. We use a single list named $blocked$ to model the task state - *blocked*. The task list, *blocked* is also modeled as an injective sequence of tasks. The sequence *blocked* will be maintained in the non-decreasing order of *inverted priority* values.

The delayed task lists, *delayedZ2* and *oDelayed* are modeled as sequences of pairs where the first element of a pair is a task t_i which is to be delayed and the second element is the *time to awake* for t_i. These sequences are maintained in the non-decreasing order of time to awake.

The set of suspended tasks, *suspended* is modeled as a set of tasks.

In Z, a sequence of type T is a function from the sub set of natural numbers to T. That is a sequence is a set of pairs of the form (i, x), where $i \in \mathbb{N}$ and $x \in T$ meaning that x is the i^{th} element in the sequence. Therefore the range of any sequence s of type T is a subset of elements of T present in s and its domain is the set of first n natural numbers $(1 \cdot\cdot n)$, where n is the cardinality of s.

In Z, $\frown/$ is the operator for concatenating a finite set of sequences of the same type. Let s be a sequence in Z, then $head(s)$ represents the first element in s and $tail(s)$ represents the sequence s' such that $s = head(s) \cdot s'$. Let x be a tuple of arity n in Z, then $x.i, 1 \leq i \leq n$ represents the ith element of x.

The constraint part of the schema in Figure 7 represents some interesting properties in FreeRTOS. For instance, the first predicate specifies that if a task is in *ready state*, it cannot be in any of the other states. Other constraints represent similar properties.

In the Z terminology the operator $\bullet$ has different meaning depending on the quantifier for bound variable [2].

$$\forall\, x : T \mid p \bullet q \iff \text{for all x in } T \text{ if x satisfies } p \text{ then } q$$

$$\exists\, x : T \mid p \bullet q \iff \text{there exists an x in } T \text{ such that x satisfies both } p \text{ and } q$$

Init_ListData

$ListData'$

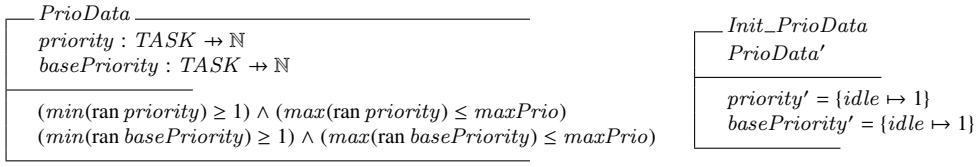
$reday'(1) = \langle idle \rangle$
$\forall\, i : \text{dom}\, ready \mid (i \neq 1) \bullet ready'(i) = \langle\rangle$
$blocked' = \langle\rangle$
$delayedZ2' = \langle\rangle$
$oDelayed' = \langle\rangle$
$suspended' = \emptyset$

Figure 8: Z schema to initialise $ListData$.

Schema in Figure 8 specifies initialisation for *ListData*.

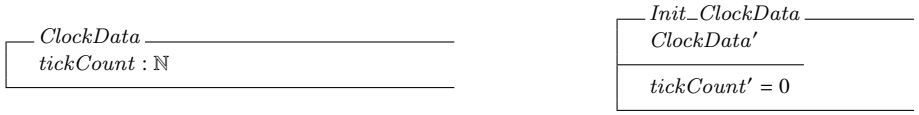
Schema in Figure 9(a) models the priorities of tasks in the system. Each task is having two priority values, the original priority and the inherited priority represented by the functions *basePriority* and *priority* respectively. Priority inheritance is a scheme used to reduce the effect of priority inversion in FreeRTOS [1]. These two functions are same if priority inheritance is not used. The value of a task under this function is its priority. Schema in Figure 9(b) specifies initialisation for *PrioData*.

The system clock is represented by an unsigned integer in FreeRTOS implementation. The value of this variable is initialised to 0 by the function to start



(a) (b)

Figure 9: (a) Z schema, $PrioData$ modelling the priorities of tasks in the system and (b) Z schema to initialise $PrioData$.



(a) (b)

Figure 10: (a) Z schema, $ClockData$ modelling the system clock in FreeRTOS and (b) Z schema to initialise $ClockData$.

the scheduler and is incremented in a function which is called from the ISR for servicing the tick interrupt. Figure 10 shows the schema and its initialisation for modelling the *system clock* in FreeRTOS.

Schema in Figure 11 combines the different component schemas to model the aspects of tasks in the system. According to the convention, there exists a single system level schema on which the operations are defined. The field *topReadyPriority* represents the highest priority among ready tasks and the field *schedulerRunning* represents the status of the scheduler (running or not).

In the constraint part, a number of interesting properties are specified. Examples include "running task is at the head of the highest priority nonempty ready sequence", "priority is defined for all valid tasks in the system", "time to awake for each delayed tasks is greater than the current value of clock" etc. Note that the specification of these properties refer to fields of different sub schemas defined above and hence can only be specified in a schema which combines such sub schemas.

There is an API to get the current clock value. This API enables the user to implement periodic tasks with the help of another API called *vTaskDelayUntil*. The application program can support only bounded numbers. Hence the clock value (*tickCount*) in the model is bounded by the variable *maxNumVal* which is assumed as the maximum numeric value in the selected domain. Note that the

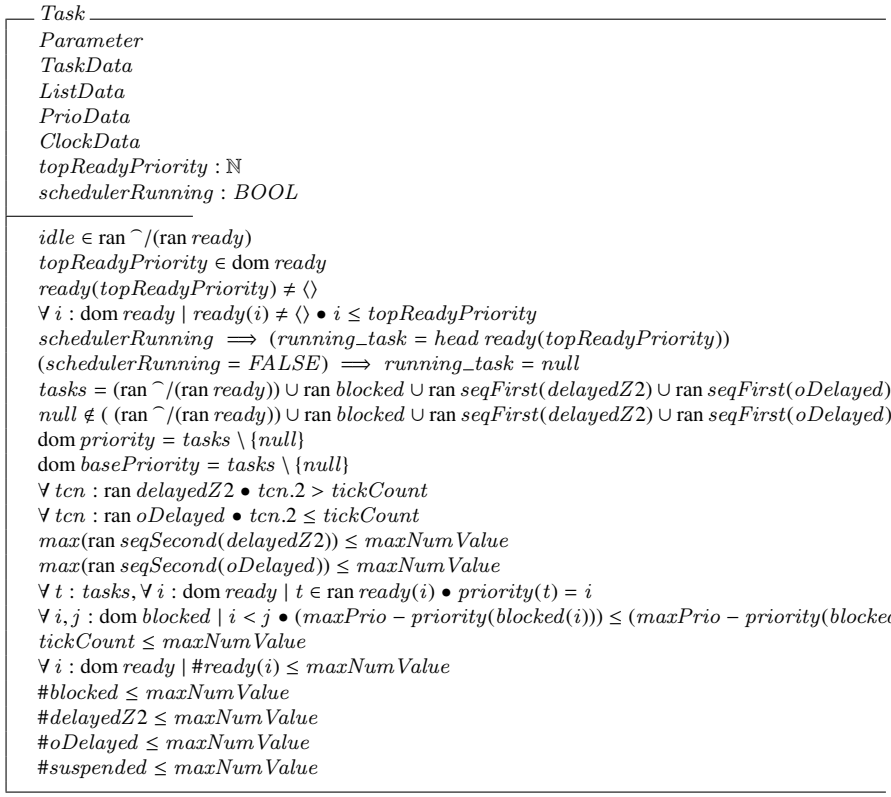


Figure 11: Z schema, $Task$ which combines sub schemas modelling task related data objects. When compared to the model M_1 the invariant regarding the ordering in blocked list is rewritten in accordance with the existing implementation FreeRTOS. But the new invariant is equivalent to the one in model M_1.

user can set the value of *maxNumVal* by an argument to initialise the system. In our Z model *tickCount* cycles in the interval [0, $maxNumVal$].

Even though the task lists are implemented as liked lists in FreeRTOS, there is a field called *uxNumberOfItems* in the list header which represents the number of nodes present in the list. Hence the maximum length of any task list is bounded by the maximum numeric value in the domain for this variable in the header. Therefore length of every task list in our model is bounded by *maxNumVal*.

Reference to a schema S_1 included in another schema S_2 will be expanded to the definition of S_1 similar to macro in programming languages like C.

The schema to initialise *Task* is given in Figure 12. The simulation tool for Z, *ProZ* needs a single initialisation schema with name *Init* to initialise the system. The *Init* in the model combines the initialisation schemas for sub schemas in the

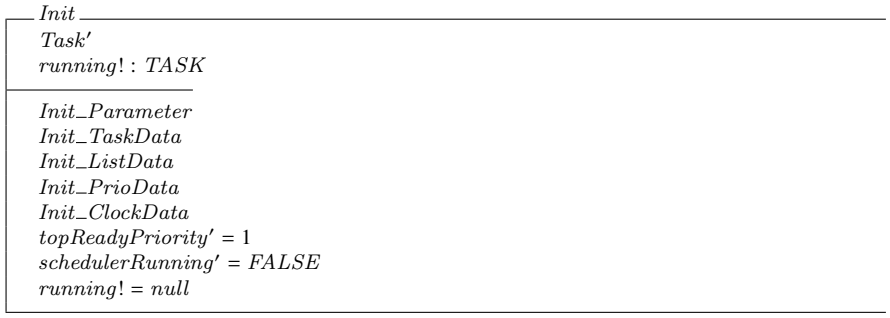


Figure 12: Schema to initialise $Task$..

model.

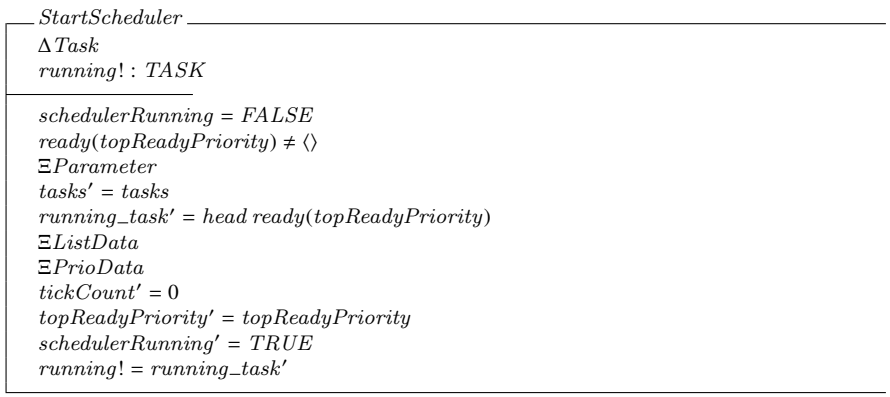


Figure 13: Operation schema $StartScheduler$ specifying $vTaskStartScheduler$ API in FreeRTOS.

FreeRTOS provides a function *vTaskStartScheduler* to activate the scheduler. Its functionality includes creating the idle task, initialising global resources etc. It schedules the longest waiting highest priority task. The convention is that the application creates a number of tasks and then calls *vTaskStartScheduler*, each task is assumed to be running in an infinite loop.

The APIs in FreeRTOS are specified as operation schemas in Z. The operation schemas agree on the type (input and output) with the corresponding API operation in FreeRTOS. The running task is a global variable (*pxCurrentTCB*) in the existing implementation of FreeRTOS. Thus every API operation returns the currently running task by this global variable. Other global resources like task

lists are also implemented as global variables. Each of the operation schema in our model includes an output field named *running!* which represents the currently running task in the post state of the corresponding operation.

Schema in Figure 13 specifies the API operation *vTaskStartScheduler* in FreeRTOS. Because of the requirement of the simulation tool *ProZ*, the initialisation of different data structures included in the definition of *vTaskStartScheduler* in FreeRTOS is captured in the system initialisation schema *Init*. If S is a schema in Z, then the expression ΞS in an operation schema is a short form for $x' = x$, where x is an arbitrary field declared in S.

An operation schema in Z should explicitly specify the relationship between the pre and post states for each field in the operand schema. Therefore if a field x do not changes its value under the operation, it must be specified as $x' = x$. Otherwise, x is allowed to take any value from its domain in the post state w.r.t. the operation.

$CreateTaskAndAddToReadyQueue_1$

$\Delta Task$
$taskIn? : TASK$
$prio? : \mathbb{N}$
$running! : TASK$

$schedulerRunning = FALSE$
$prio? \in \text{dom}\ ready$
$taskIn? \notin tasks$
$\#ready(prio?) < maxNumValue$
$\Xi Parameter$
$tasks' = tasks \cup \{taskIn?\}$
$running\_task' = running\_task$
$ready' = ready \oplus \{ (prio? \mapsto ready(prio?) \frown \langle taskIn? \rangle) \}$
$blocked' = blocked$
$delayedZ2' = delayedZ2$
$oDelayed' = oDelayed$
$suspended' = suspended$
$priority' = priority \oplus \{(taskIn? \mapsto prio?)\}$
$basePriority' = priority \oplus \{(taskIn? \mapsto prio?)\}$
$\Xi ClockData$
$topReadyPriority' = topReadyPriority$
$schedulerRunning' = schedulerRunning$
$running! = running\_task'$

Figure 14: Operation schema $CreateTaskAndAddToReadyQueue_1$ specifying $xTaskCreate$ API when the scheduler is not started. Here the running task is $null$ by an invariant in schema $Task$.

FreeRTOS provides a function, *xTaskCreate* to create a task. schemas of Figures 14, 15 and 16, specify the API operation - *xTaskCreate*. The functionality of *xTaskCreate* in FreeRTOS is captured in three operation schemas in the

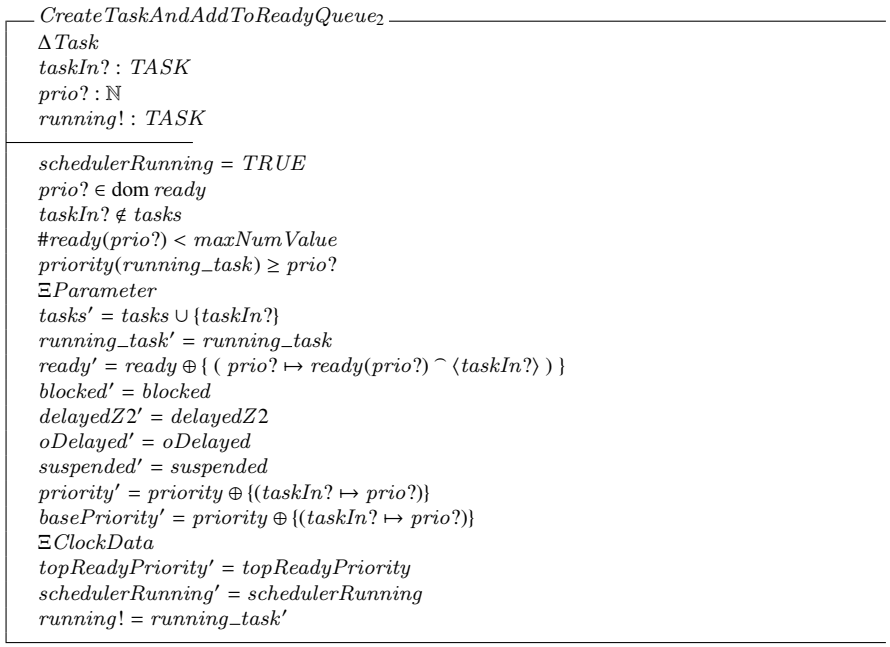


Figure 15: Operation schema $CreateTaskAndAddToReadyQueue_2$ specifying $xTaskCreate$ API when the task to be created has priority less than or equal to the priority of running task.

model. This is because of the semantics of the Z language. Note that the operation schemas of Figures 14, 15 and 16 have different preconditions and different post states depending on the respective precondition.

Suppose there are more than one schema modelling an operation in FreeRTOS, then these schemas will be distinguished by their preconditions. For example in the schemas of Figures 15 and 16, the priority of the task to be created with respect to the priority of running task distinguishes the schemas for modelling *xTaskCreate*. The schema in Figure 14 specifies *xTaskCreate* when the scheduler is not started.

The schema operator disjunction ($\vee$) in Z can be used to combine such schemas to get the operation schema specifying the corresponding function implementation in FreeRTOS. For example, $CreateTask = CreateTaskAndAddToReadyQueue_1 \vee CreateTaskAndAddToReadyQueue_2 \vee CreateTaskAndSchedule$ gives the operation schema corresponding to the operation *xTaskCreate* in FreeRTOS.

If the priority of task to be created is greater than priority of running task, then it goes to a higher index sequence in *ready*, which is now empty as specified by an invariant in the schema *Task*. In Z the operator $\oplus$ is used to override a relation

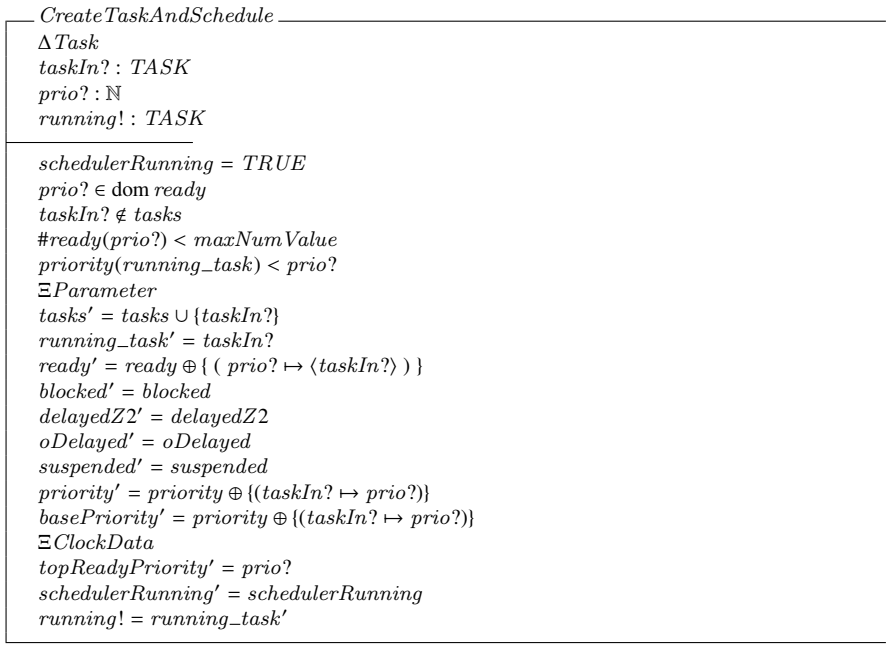


Figure 16: Operation schema $CreateTaskAndSchedule$ specifying $xTaskCreate$ API when the task to be created has priority greater than the priority of running task.

and it updates the relation by changing the images of those elements as specified in the second operand to this operator.

The task creation operation adds the new task to the set $tasks$ as well as to the ready queue and also adds the required information to the priority functions to represent the new task. In one case the operation schedules the newly created task and is determined by its precondition.

FreeRTOS provides an API, *vTaskDelete* to delete a task. The schemas of Figures 17 and 18 specify the deletion operation for running task.

The preconditions of the schema in Figures 17 specify that the sequence of ready task indexed by *topReadyPriority* contains at least one more task in addition to the running task. Hence this schema schedules the longest waiting task among the other tasks in this sequence.

The precondition of the schema in Figure 18 specifies that running task is the only task in the sequence indexed by *topReadyPriority* and there exists at least one nonempty sequence indexed by a number less than *topReadyPriority*. Hence this schema schedules the head of the next highest index nonempty sequence and updates *topReadyPriority* to this index.

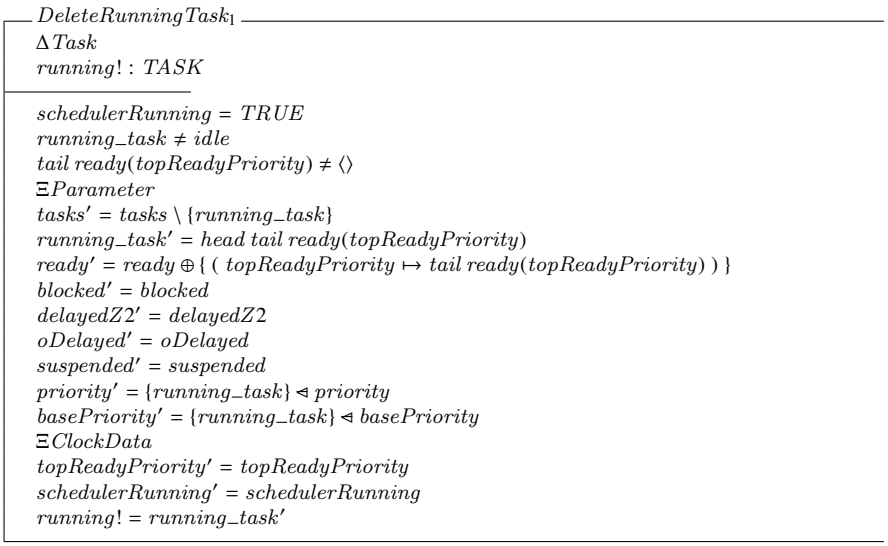


Figure 17: Operation schema $DeleteRunningTask_1$ specifying $vTaskDelete$ API when there exists at least one more ready task in the sequence corresponding to the priority of running task.

Let x be the name of a filed in the declaration part of an operation schema S such that x is not ending with ? (input) or ! (output). Then x represents a constraint on the state and is given in the constraint part of the schema S. For example $runnerUpPrty$ in schema $DeleteRunningTask_2$ represents the index of the second highest nonempty sequence in ready.

In both of the above cases the running task is removed from the ready queue and the set $tasks$ & priority functions are updated to effect the deletion.

In Z, $\lhd$ is the operator to remove those pairs from a binary relation with the first element present in the first operand to this operator. Similarly $\rhd$ is the operator to remove those pairs from a binary relation with the image (second element) present in the second operand to this operator. Removing one or more elements from a sequence will destroy its structure. For example if we remove the i^{th} element, then i is not mapped to any element. Z provides the operator $squash$ to restructure the resulting set of pairs to form a valid sequence by maintaining the relative order among the elements.

The operation schemas of Figure 19 specifies the operations to delete a task t_i when t_i is present in $ready$ list or in $blocked$ list. In both of these schemas the given task is removed from the respective task list and the set $tasks$ & priority functions are updated to effect the deletion.

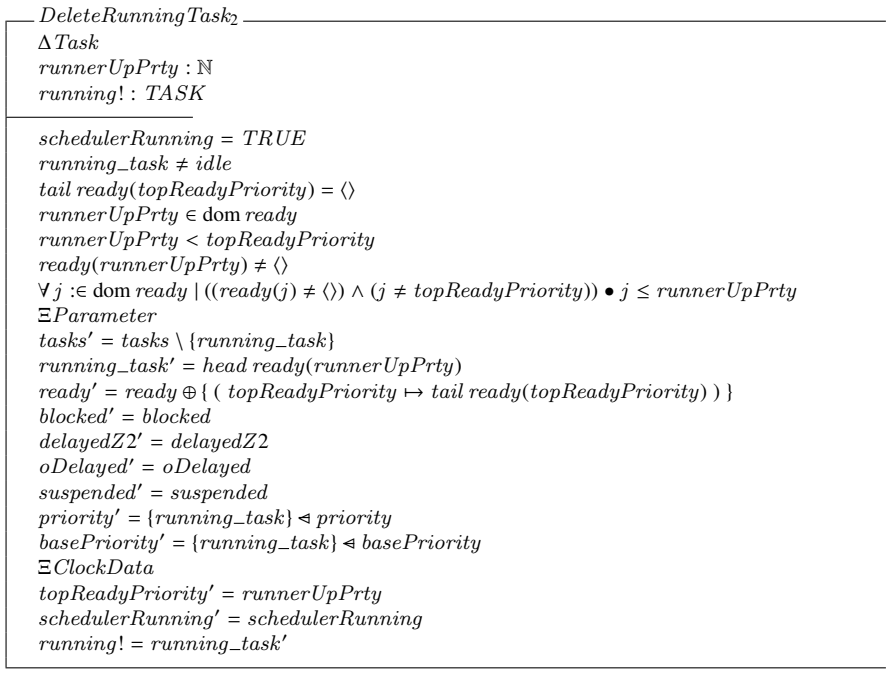


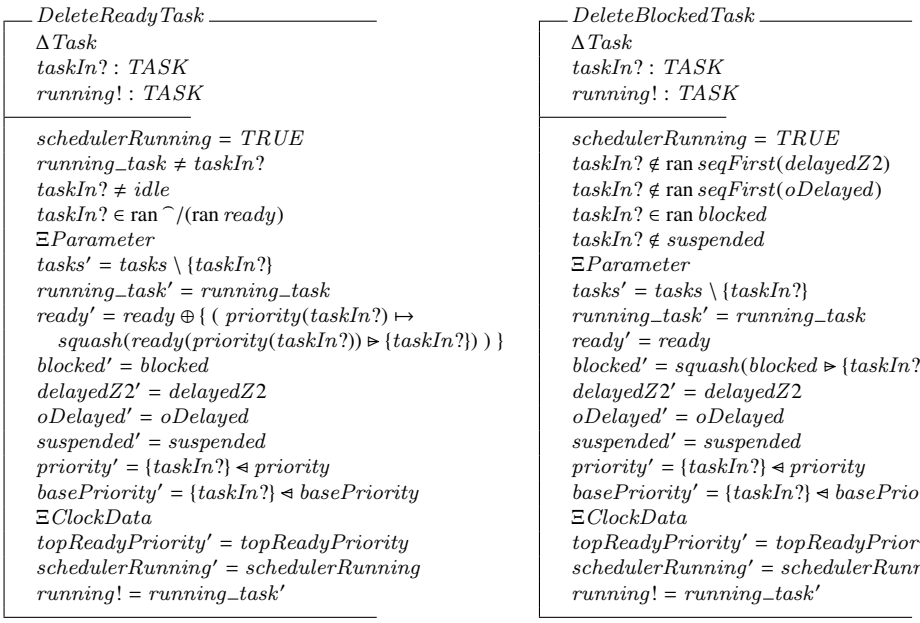
Figure 18: Operation schema $DeleteRunningTask_2$ specifying $vTaskDelete$ API when running task is the only task in the sequence corresponding to the priority of running task.

The operation schemas of Figure 20 specify the operations to delete a task t_i when t_i is present only in $delayedZ2$ list or in both $delayedZ2$ list and $blocked$ lists. The filter operator ($\upharpoonright$) in Z projects a given sequence into a new sequence in which all elements satisfies the constraint given as the second operand to $\upharpoonright$. In both of the schemas of Figure 20, the given task is removed from the respective task lists and the set $tasks$ & priority functions are updated to effect the deletion.

The operation schemas of Figure 21 specify the operations to delete a task t_i when t_i is present only in $oDelayed$ list or in both $oDelayed$ list and $blocked$ lists. Both of the schemas of Figure 21 remove the given task from the respective task lists and the set $tasks$ & priority functions are updated to effect the deletion.

The operation schemas of Figure 22 specify the operations to delete a task t_i when t_i is present only in $suspended$ list or in both $suspended$ list and $blocked$ lists. In both of the schemas of Figure 22, the given task is removed from the respective task lists and the set $tasks$ & priority functions are updated to effect the deletion.

In FreeRTOS if a blocking API is called by a task t_i with maximum value



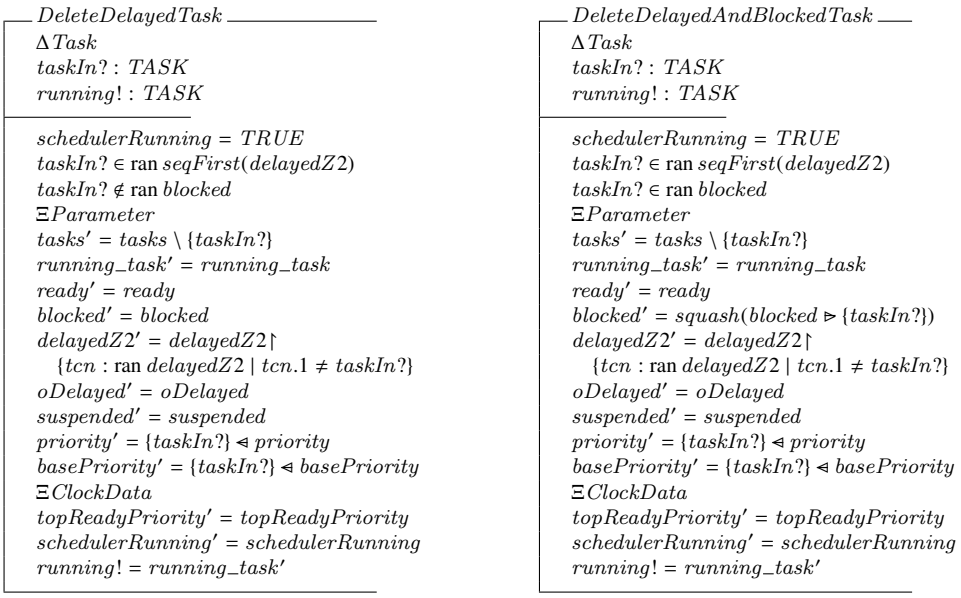
(a) (b)

Figure 19: Operation schemas $DeleteReadyTask$ and $DeleteBlockedTask$ specifying $vTaskDelete$ API when the task to be deleted is in *ready* or in *blocked* list respectively.

($maxNumValue$) for time to wait, then t_i is added to $suspended$ list instead of adding it to $delayedZ2$ list in addition to adding t_i to the waiting list ($blocked$) for the resource concerned. Thus a task can be in the $blocked$ list and $delayedZ2$ or $suspended$ list at the same time. The relationship among the task lists is governed by the invariants in the schema $ListData$ which disallow a task to be in some of these lists when it is present in some other list.

In FreeRTOS if a blocking API is called by a task t_i with maximum value ($maxNumValue$) for time to wait, then t_i is added to $suspended$ list instead of adding it to $delayedZ2$ list in addition to adding t_i to the waiting list ($blocked$) for the resource concerned. Thus a task can be in the $blocked$ list and $delayedZ2$ or $suspended$ list at the same time. The relationship among the task lists is governed by the invariants in the schema $ListData$ which disallow a task to be in some of these lists when it is present in some other list.

FreeRTOS provides two APIs for delaying a task. The function *vTaskDelayUntil* delays a task for a given (as an argument) number of clock ticks w.r.t. the previous wake up time which is given as another argument. On the other hand the function *vTaskDelay* delays a task for a given (as the argument) number of clock



(a) (b)

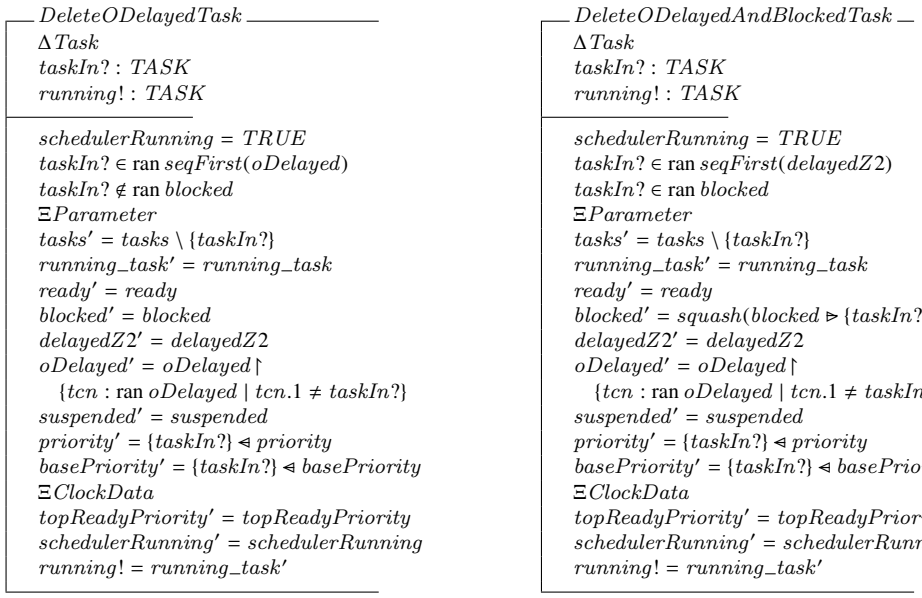
Figure 20: Operation schemas $DeleteDelayedTask$ and $DeleteDelayedAndBlockedTask$ specifying $vTaskDelete$ API when the task to be deleted is in only *delayedZ2* list or in both *delayedZ2* list and *blocked* list respectively.

ticks w.r.t. the system clock. That is in the first case the delay time is relative to the previous wake up time given as an argument and in the second case, the delay is absolute.

he system clock is defined as an unsigned integer in the existing implementation. Hence the system clock is bounded by the maximum possible value for unsigned integer. Therefore the clock value (*tickCount*) is bounded in the model M_1. The value is bounded by the a parameter called *maxNumVal*. The user can supply the value for this variable as a parameter to initialise the system.

The operation to update the value of *tickCount* performs a modulo ($maxNumVal$+ 1) increment so that its value cycles in the interval [0, $maxNumVal$]. That is the value of *tickCount* is reset to zero when the current value is *maxNumVal*. This is called *clock overflow*.

The function *vTaskDelayUntil* takes two arguments - *delay period* and *previous wake time*. The purpose is to delay the running task for *delay period* number of clock ticks from *previous wake time*. The wake-up time for the task is the sum of *delay period* and *previous wake time*. Depending on the arguments the function



(a) (b)

Figure 21: Operation schemas $DeleteODelayedTask$ and $DeleteODelayedAndBlockedTask$ specifying $vTaskDelete$ API when the task to be deleted is in only *oDelayed* list or in both *oDelayed* list and *blocked* list respectively.

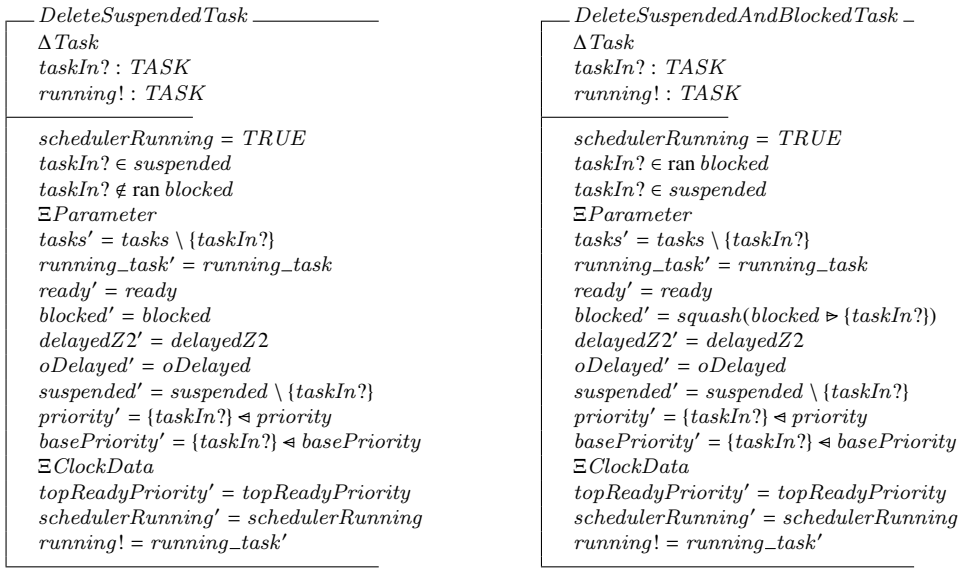
vTaskDelayUntil needs to consider the following two cases.

1. No clock overflow happened after *previous wake time*. In this case both *previous wake time* and *tickCount* are in the same time window ([0, $maxNumVal$]).

2. Clock overflow happened after *previous wake time*. In this case *previous wake time* and *tickCount* are in different time windows.

Figure 23 shows the possible values for *time to awake* when *previous wake time* and *tickCount* are in the same time window. Figure 24 shows the possible values for *time to awake* when *previous wake time* and *tickCount* are in different time windows.

The operation schemas of Figures 25 specify the operation $vTaskDelayUntil$ when the previous wake time and requested wake time are in the same *time window* ($prevWakeTime? \leq tickCount$) . According to a precondition the requested wake time is already elapsed and hence no state change is required in this case.



(a) (b)

Figure 22: Operation schemas $DeleteSuspendedTask$ and $DeleteSuspendedAndBlockedTask$ specifying $vTaskDelete$ API when the task to be deleted is in only *suspended* list or in both *suspended* list and *blocked* list respectively.

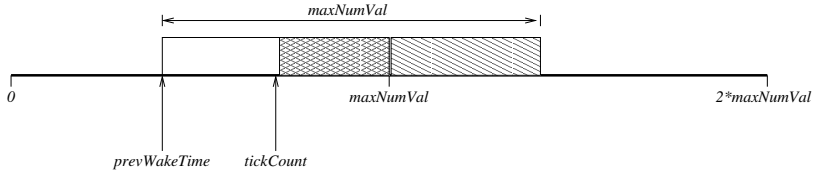


Figure 23: Possible values for wake-up time when *previous wake time* and *tick-Count* are in the same time window. The running task need to be delayed iff the condition (($prevWakeTime \leq tickCount$) $\wedge$ ($tickCount <$ ($prevWakeTime$ + $delayPeriod$))) evaluates to *true*.

The operation schemas of Figures 26 specify the operation $vTaskDelayUntil$ when the previous wake time and requested wake time are in different *time windows* (value of *tickCount* overflowed after the previous wake time). According to a precondition the requested wake time was in the previous time window which is already elapsed and hence no state change is required in this case.

The operation schemas of Figures 27 specify the operation $vTaskDelayUntil$

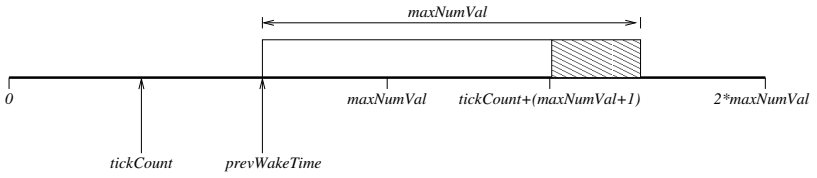


Figure 24: Possible values for wake-up time when *previous wake time* and *tickCount* are in the different time windows.The running task need to be delayed iff the condition (($prevWakeTime > tickCount$) $\wedge$ (($tickCount$ + ($maxNumVal$ + 1)) < ($prevWakeTime$ + $delayPeriod$))) evaluates to ***true***.

$TaskDelayUntil_1$

$\Delta Task$
$prevWakeTime? : \mathbb{N}$
$delay? : \mathbb{N}$
$running! : TASK$
$prevWakeTime! : \mathbb{N}$

$schedulerRunning = TRUE$
$delay? \leq maxNumValue$
$prevWakeTime? \leq maxNumValue$
$prevWakeTime? \leq tickCount$
$(prevWakeTime? + delay?) \leq tickCount$
$\Xi Parameter$
$\Xi TaskData$
$\Xi ListData$
$\Xi PrioData$
$\Xi ClockData$
$topReadyPriority' = topReadyPriority$
$schedulerRunning' = schedulerRunning$
$running! = running\_task'$
$prevWakeTime! = prevWakeTime? + delay?$

Figure 25: Operation schema $TaskDelayUntil_1$ specifying $vTaskDelayUntil$ API when the previous wake time and requested wake time are in the same *time window*. According to a precondition the requested wake time is already elapsed and hence no state change is required.

when the previous wake time and requested wake time are in different *time windows* (value of *tickCount* overflowed after the previous wake time). According to a precondition the requested wake time in the current time window is already elapsed and hence no state change is required. The value of previous wake time is updated using modulo (maxNumValue + 1) addition so that the new value is in the current window for the clock (*tickCount*).

The operation schemas of Figures 28 specify the operation $vTaskDelayUntil$

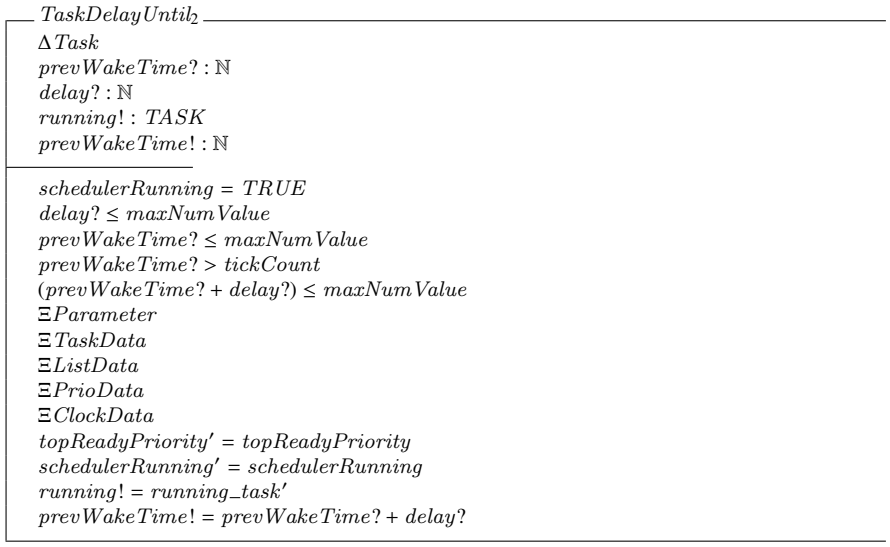


Figure 26: Operation schema $TaskDelayUntil_2$ specifying $vTaskDelayUntil$ API when the previous wake time and requested wake time are in different *time windows* (value of *tickCount* overflowed). According to a precondition the requested wake time was in the previous time window which is already elapsed and hence no state change is required.

$TaskDelayUntil_3$

$\Delta Task$
$prevWakeTime? : \mathbb{N}$
$delay? : \mathbb{N}$
$running! : TASK$
$prevWakeTime! : \mathbb{N}$

$schedulerRunning = TRUE$
$delay? \leq maxNumValue$
$prevWakeTime? \leq maxNumValue$
$prevWakeTime? > tickCount$
$(prevWakeTime? + delay?) > maxNumValue$
$(prevWakeTime? + delay?) \leq (tickCount + (maxNumValue + 1))$
$\Xi Parameter$
$\Xi TaskData$
$\Xi ListData$
$\Xi PrioData$
$\Xi ClockData$
$topReadyPriority' = topReadyPriority$
$schedulerRunning' = schedulerRunning$
$running! = running\_task'$
$prevWakeTime! = prevWakeTime? + delay? - (maxNumValue + 1)$

Figure 27: Operation schema $TaskDelayUntil_3$ specifying $vTaskDelayUntil$ API when the previous wake time and requested wake time are in different *time windows* (value of *tickCount* overflowed). According to a precondition the requested wake time in the current time window is already elapsed and hence no state change is required. The value of previous wake time is updated using modulo (maxNumValue + 1) addition so that the new value is in the current window for the clock (*tickCount*).

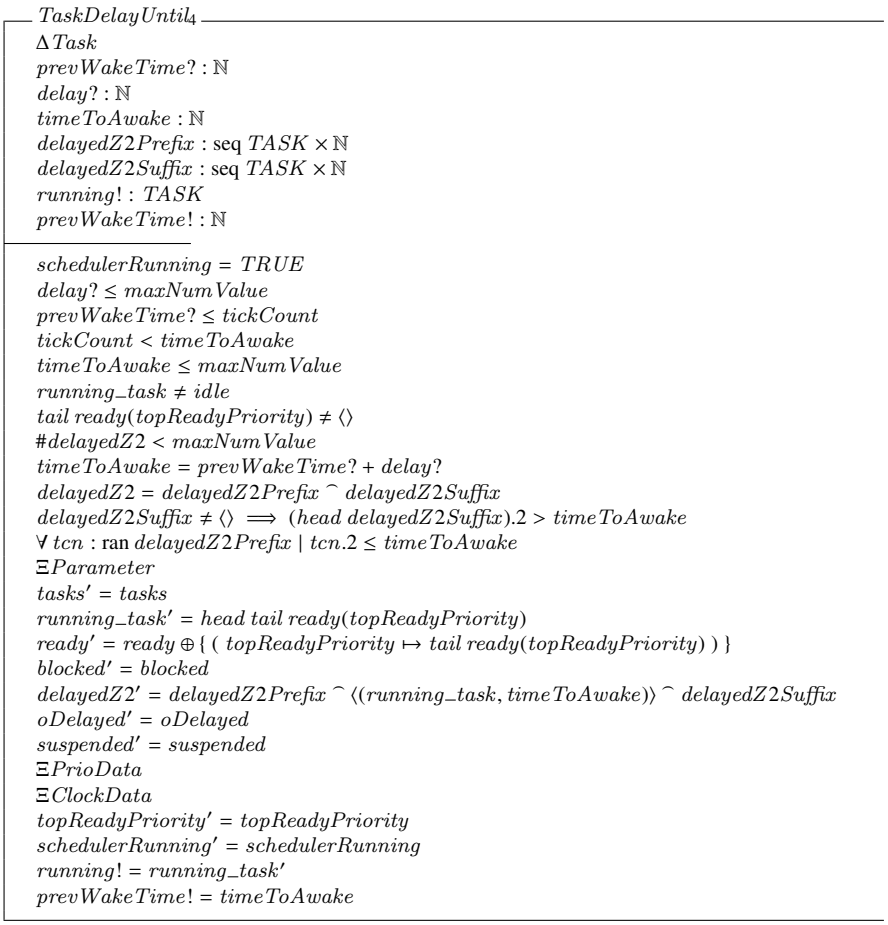


Figure 28: Operation schema $TaskDelayUntil_4$ specifying $vTaskDelayUntil$ API when the previous wake time and requested wake time are in the same *time window*. According to precondition there exists at least one more ready task in the sequence corresponding to the priority of running task and the new value of time to awake is within the current time window and is greater than the value of *tickCount*.

when the previous wake time and requested wake time are in the same *time window* ($prevWakeTime? \leq tickCount$) . According to precondition there exists at least one more ready task in the sequence corresponding to the priority of running task and the new value of time to awake is within the current time window and is greater than the value of *tickCount*.

The field *delayedZ2Prefix* of the schema in Figure 28 represents the maximal prefix of the sequence *delayedZ2* where value of *time to awake* for each task is

less than or equal to the value of *time to awake* for the task to be delayed. The field *delayedZ2Suffix* represents the remaining part of the sequence *delayedZ2*.

The list $delayedZ2$ is updated such that the modification restores its invariant given in the schema $ListData$.

The operation schemas of Figures 29 specify the operation $vTaskDelayUntil$ when the previous wake time and requested wake time are in the same *time window* ($prevWakeTime? \leq tickCount$) . According to precondition running task is the only task in the sequence corresponding to the priority of running task and the new value of time to awake is within the current time window and is greater than the value of *tickCount*.

The operation schemas of Figures 30 specify the operation $vTaskDelayUntil$ when the previous wake time and requested wake time are in the same *time window* ($prevWakeTime? \leq tickCount$) . According to precondition there exists at least one more ready task in the sequence corresponding to the priority of running task. The value of time to awake is updated using modulo (maxNumValue + 1) addition. The value of ($prevWakeTime?$ + $delay?$) is greater than *maxNumVal* and hence the task need to be added to the list *oDelayed*.

The operation schemas of Figures 31 specify the operation $vTaskDelayUntil$ when the previous wake time and requested wake time are in the same *time window* ($prevWakeTime? \leq tickCount$) . According to precondition running task is the only task in the sequence corresponding to the priority of running task. The value of time to awake is updated using modulo (maxNumValue + 1) addition. The value of ($prevWakeTime?$ + $delay?$) is greater than *maxNumVal* and hence the task need to be added to the list *oDelayed*.

The operation schemas of Figures 32 specify the operation $vTaskDelayUntil$ when the previous wake time and requested wake time are in different *time windows* (value of *tickCount* overflowed). According to a precondition there exists at least one more ready task in is the sequence corresponding to the priority of running task. The value of time to wake is updated using modulo (maxNumValue + 1) addition so that the new value is in the current time window for the clock (*tickCount*).

The operation schemas of Figures 33 specify the operation $vTaskDelayUntil$ when the previous wake time and requested wake time are in different *time windows* (value of *tickCount* overflowed). According to a precondition running task is the only task in the sequence corresponding to the priority of running task. The value of time to wake is updated using modulo (maxNumValue + 1) addition so that the new value is in the current time window for the clock (*tickCount*).

The operation schemas of Figures 35 specify the operation $vTaskDelay$ when the value of ($tickCount$ + $delay?$) is less than or equal to *maxNumVal*. According to a precondition there exists at least one more ready task in the sequence corresponding to the priority of running task. The running task is added to the

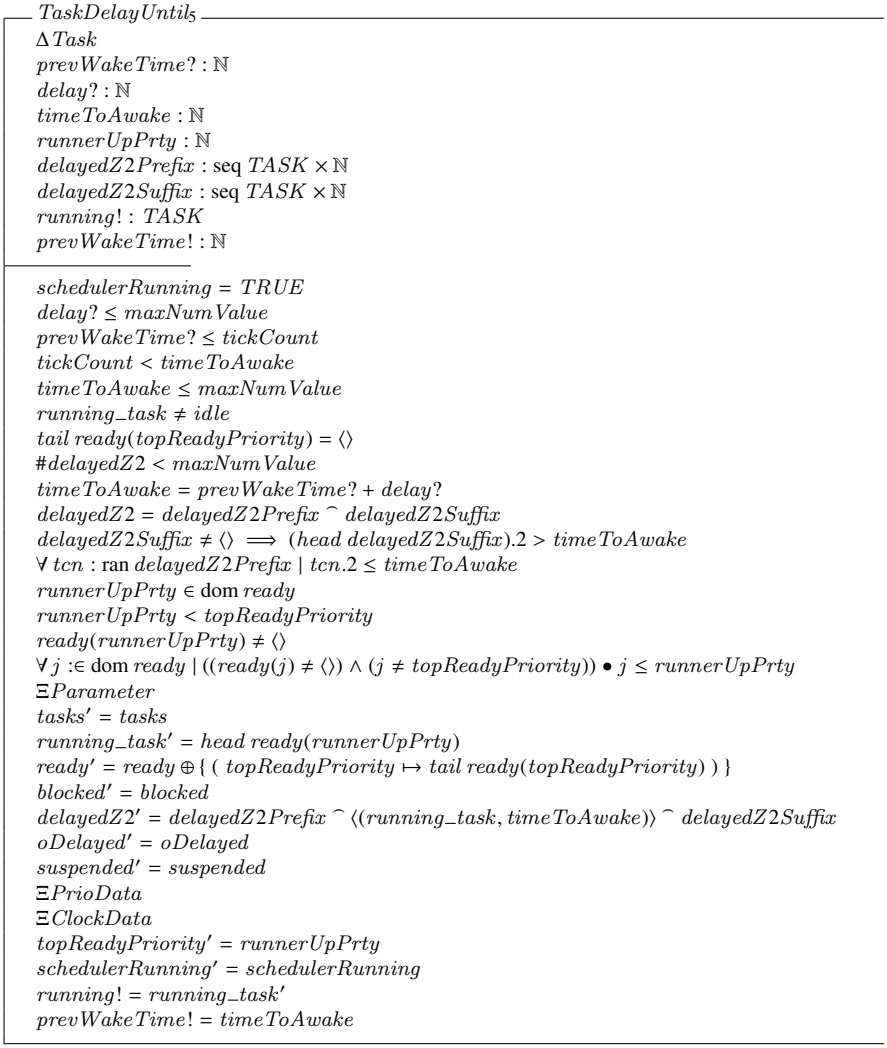


Figure 29: Operation schema $TaskDelayUntil_5$ specifying $vTaskDelayUntil$ API when the previous wake time and requested wake time are in the same *time window*. According to precondition running task is the only task in the sequence corresponding to the priority of running task and the new value of time to awake is within the current time window and is greater than the value of *tickCount*.

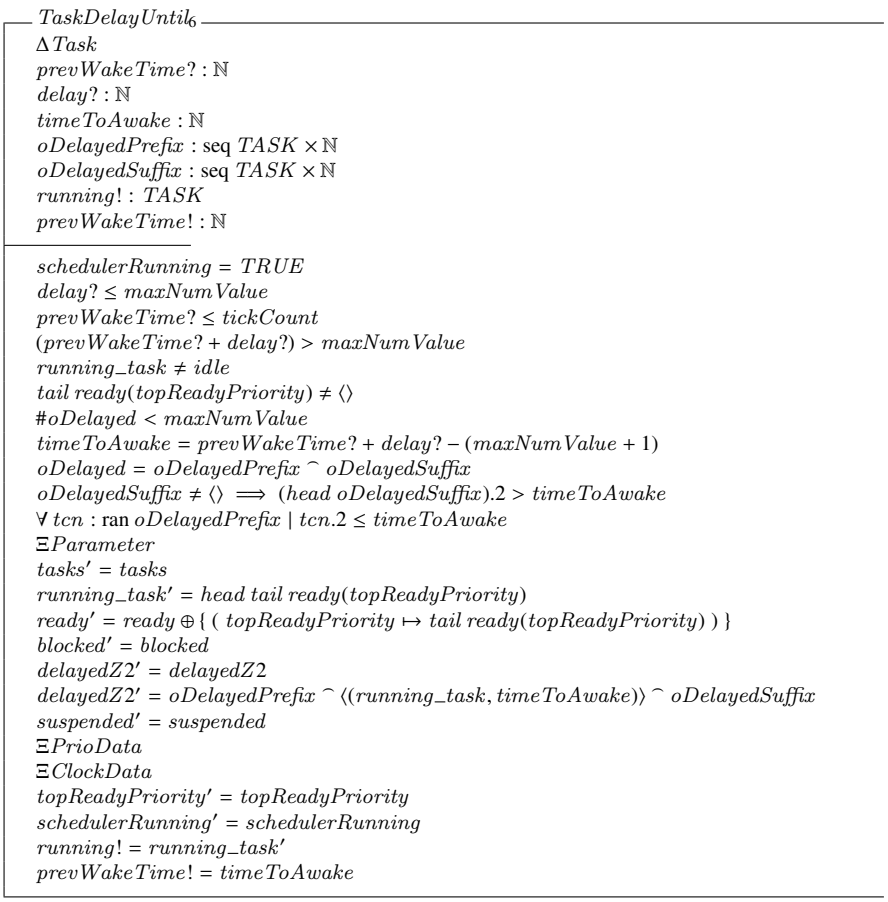


Figure 30: Operation schema $TaskDelayUntil_6$ specifying $vTaskDelayUntil$ API when the previous wake time and requested wake time are in the same *time window*. According to precondition there exists at least one more ready task in the sequence corresponding to the priority of running task. The value of time to awake is updated using modulo (maxNumValue + 1) addition.

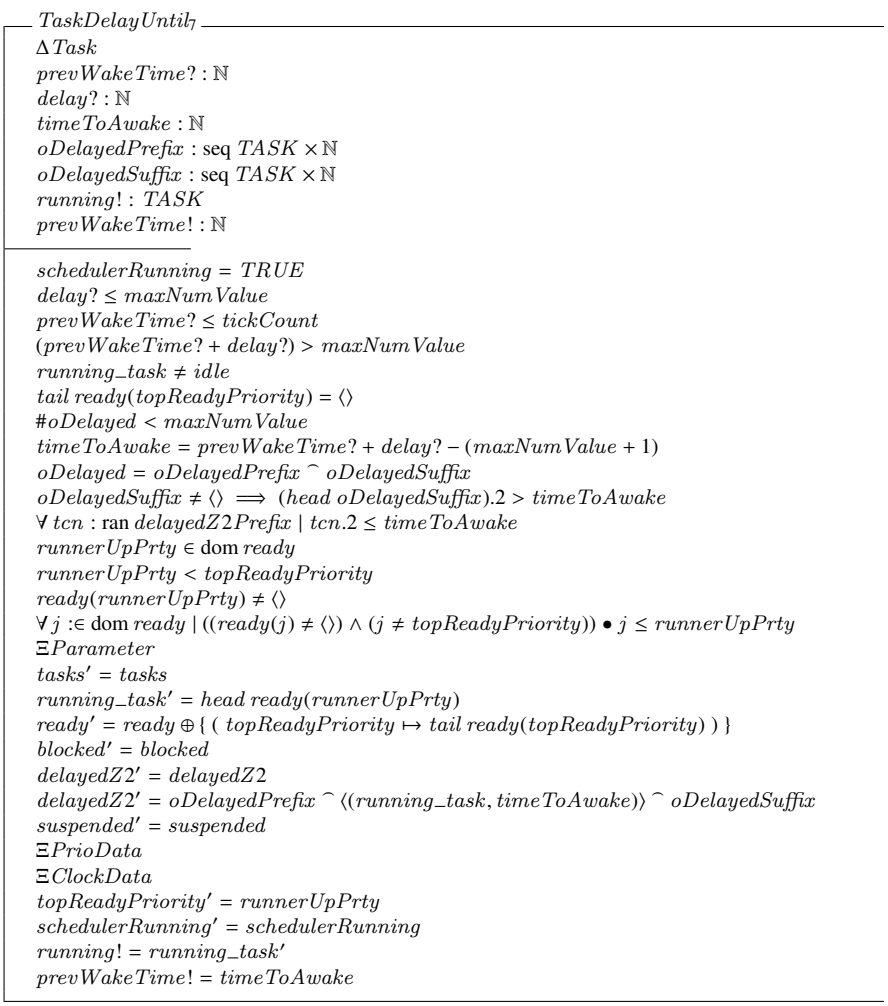


Figure 31: Operation schema $TaskDelayUntil_7$ specifying $vTaskDelayUntil$ API when the previous wake time and requested wake time are in the same *time window*. According to precondition running task is the only task in the sequence corresponding to the priority of running task. The value of time to awake is updated using modulo (maxNumValue + 1) addition.

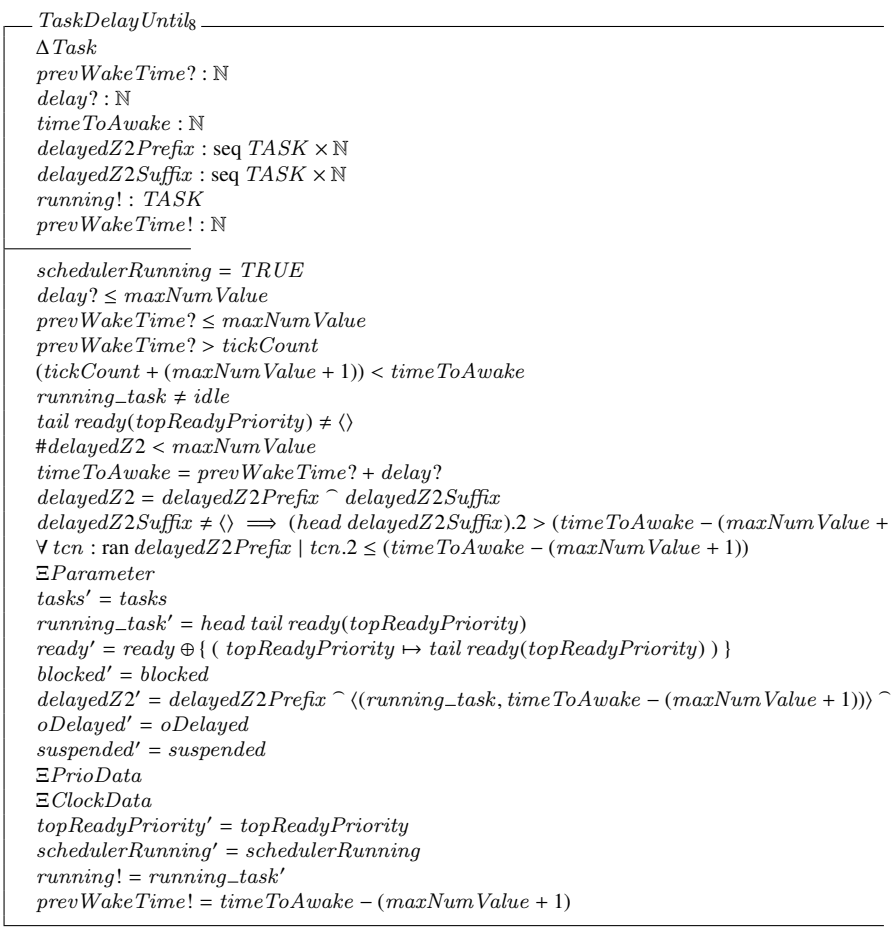


Figure 32: Operation schema $TaskDelayUntil_8$ specifying $vTaskDelayUntil$ API when the previous wake time and requested wake time are in different *time windows* (value of *tickCount* overflowed). According to a precondition there exists at least one more ready task in is the sequence corresponding to the priority of running task. The value of time to wake is updated using modulo (maxNumValue + 1) addition so that the new value is in the current time window for the clock (*tickCount*).

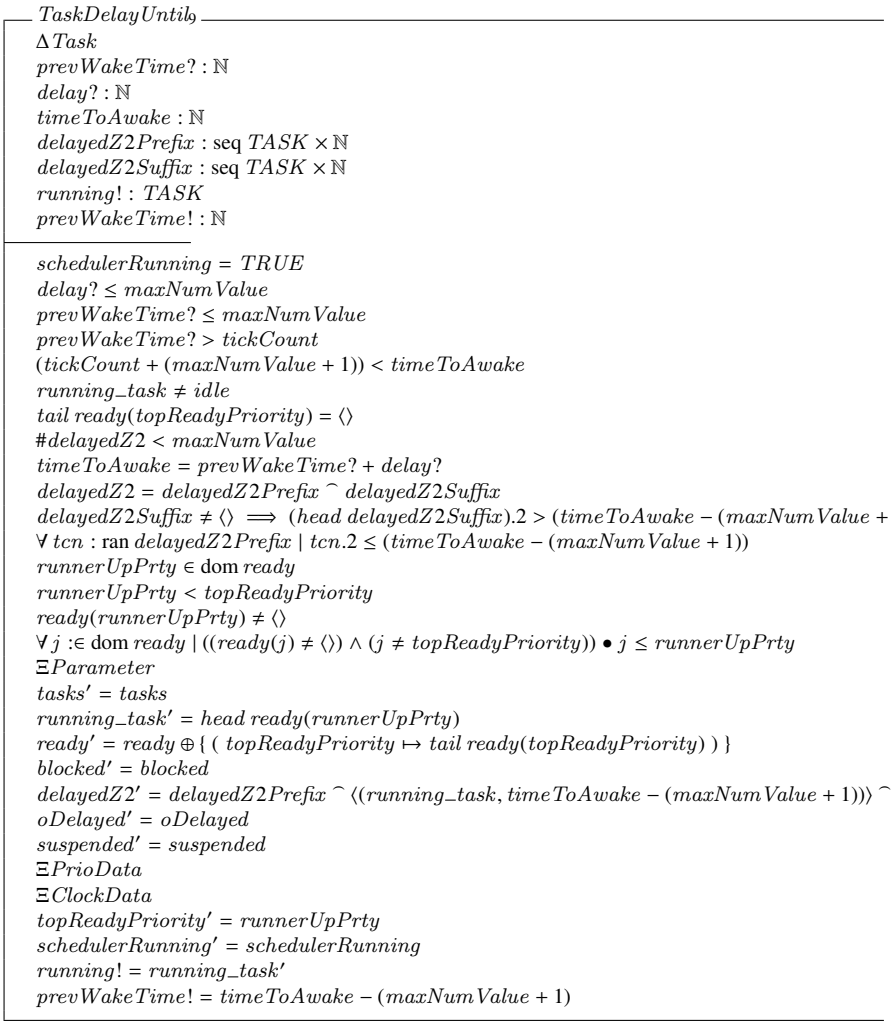


Figure 33: Operation schema $TaskDelayUntil_8$ specifying $vTaskDelayUntil$ API when the previous wake time and requested wake time are in different *time windows* (value of *tickCount* overflowed). According to a precondition running task is the only task in the sequence corresponding to the priority of running task. The value of time to wake is updated using modulo (maxNumValue + 1) addition so that the new value is in the current time window for the clock (*tickCount*).

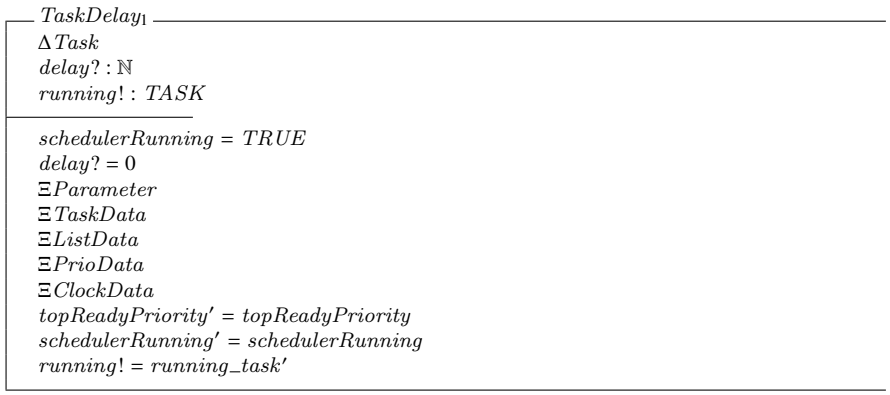


Figure 34: Operation schema $TaskDelay_1$ specifying $vTaskDelay$ API when the delay period is 0. Hence no state change is required.

list *delayedZ2* such that the post state of the operation respects the invariants on *delayedZ2*.

The operation schemas of Figures 36 specify the operation $vTaskDelay$ when the value of ($tickCount$ + $delay$?) is greater than *maxNumVal*. According to a precondition there exists at least one more ready task in the sequence corresponding to the priority of running task. The running task is added to the list *oDelayed* such that the post state of the operation respects the invariants on *oDelayed*. The value of time to awake is computed using modulo ($maxNumVal$ + 1) addition.

The operation schemas of Figures 35 specify the operation $vTaskDelay$ when the value of ($tickCount$ + $delay$?) is less than or equal to *maxNumVal*. According to a precondition running task is the only task in the sequence corresponding to the priority of running task. The running task is added to the list *delayedZ2* such that the post state of the operation respects the invariants on *delayedZ2*.

The operation schemas of Figures 38 specify the operation $vTaskDelay$ when the value of ($tickCount$ + $delay$?) is greater than *maxNumVal*. According to a precondition running task is the only task in the sequence corresponding to the priority of running task. The running task is added to the list *oDelayed* such that the post state of the operation respects the invariants on *oDelayed*. The value of time to awake is computed using modulo ($maxNumVal$ + 1) addition.

In FreeRTOS, there is a function called *vTaskIncrementTick* to increment the system clock. This function is automatically invoked when a tick interrupt occurs. This function increments the system clock by one and moves all delayed tasks with wake up time same as the new clock value to ready queue. It also performs a context switch if it moves a task of priority higher than the priority of running

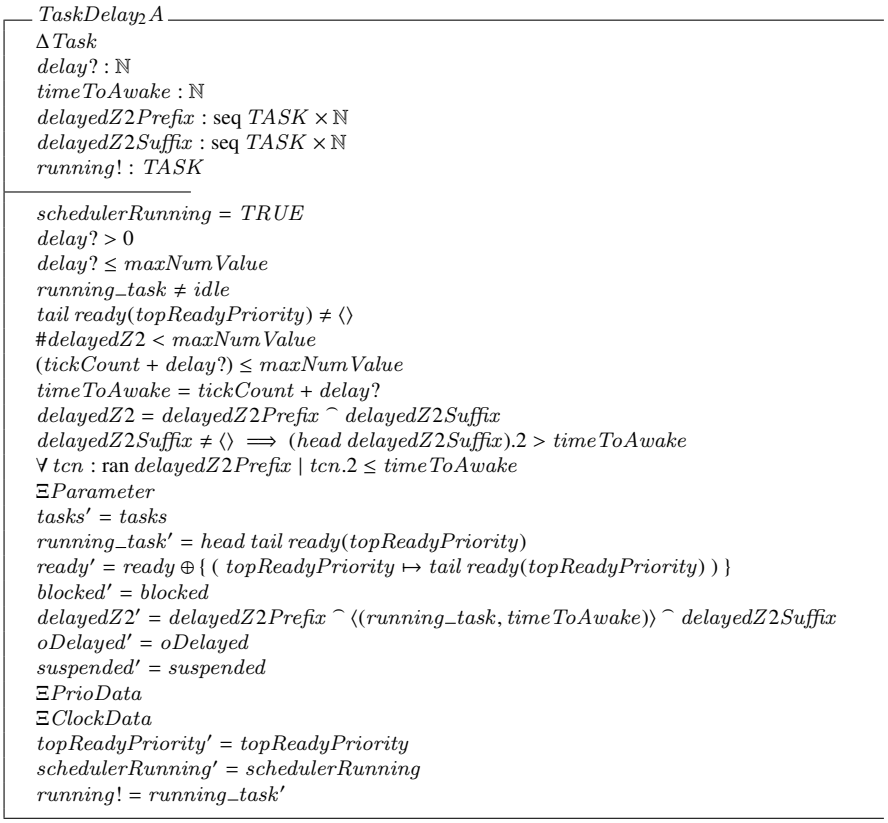


Figure 35: Operation schema $TaskDelay_2A$ specifying $vTaskDelay$ API when the value of ($tickCount$ + $delay$?) is less than or equal to *maxNumVal*. According to a precondition there exists at least one more ready task in the sequence corresponding to the priority of running task.

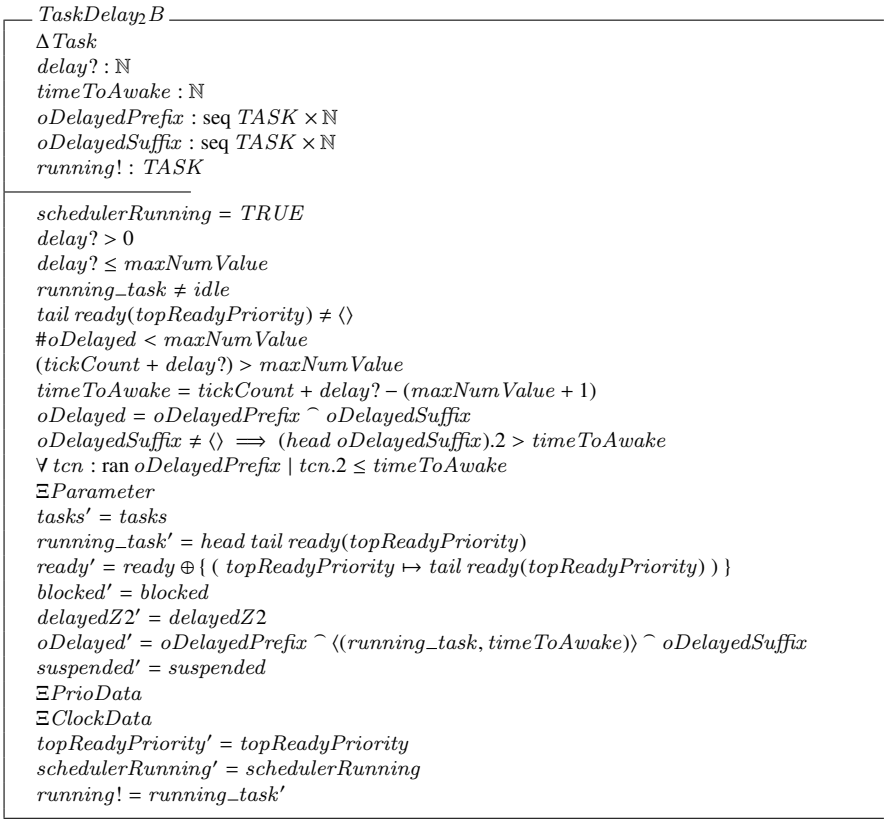


Figure 36: Operation schema $TaskDelay_2B$ specifying $vTaskDelay$ API when the value of ($tickCount + delay?$) is greater than *maxNumVal*. According to a precondition there exists at least one more ready task in the sequence corresponding to the priority of running task.

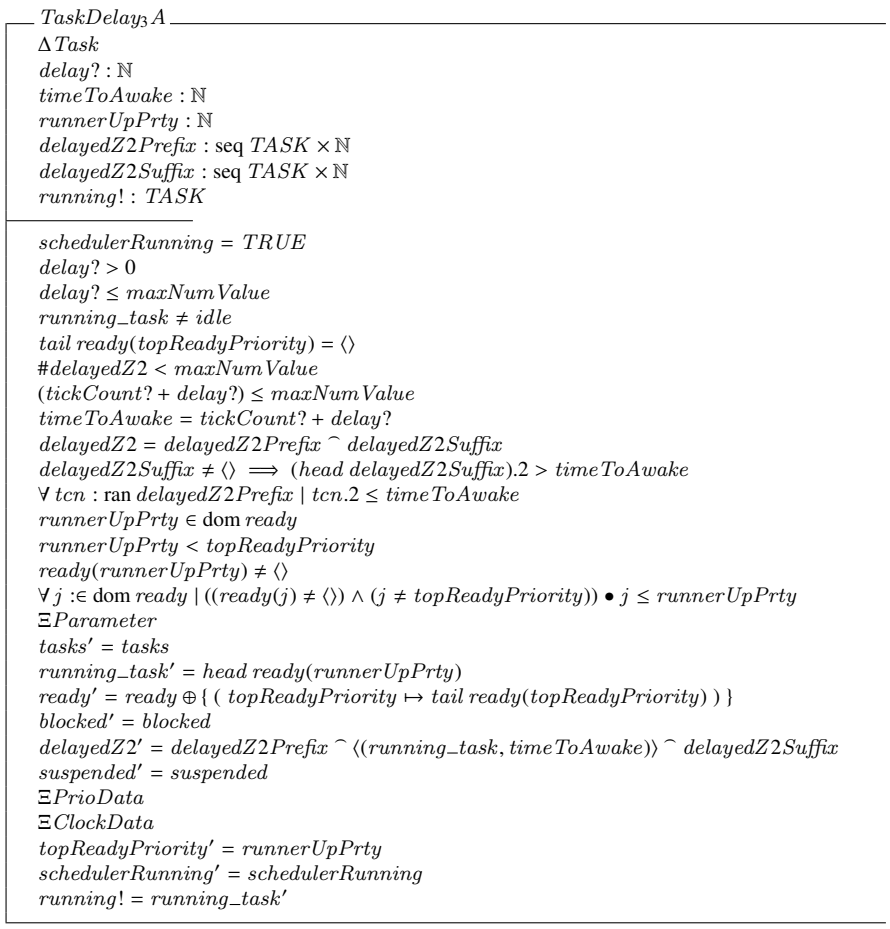


Figure 37: Operation schema $TaskDelay_3 A$ specifying $vTaskDelay$ API when the value of ($tickCount$ + $delay$?) is less than or equal to *maxNumVal*. According to a precondition running task is the only task in the sequence corresponding to the priority of running task.

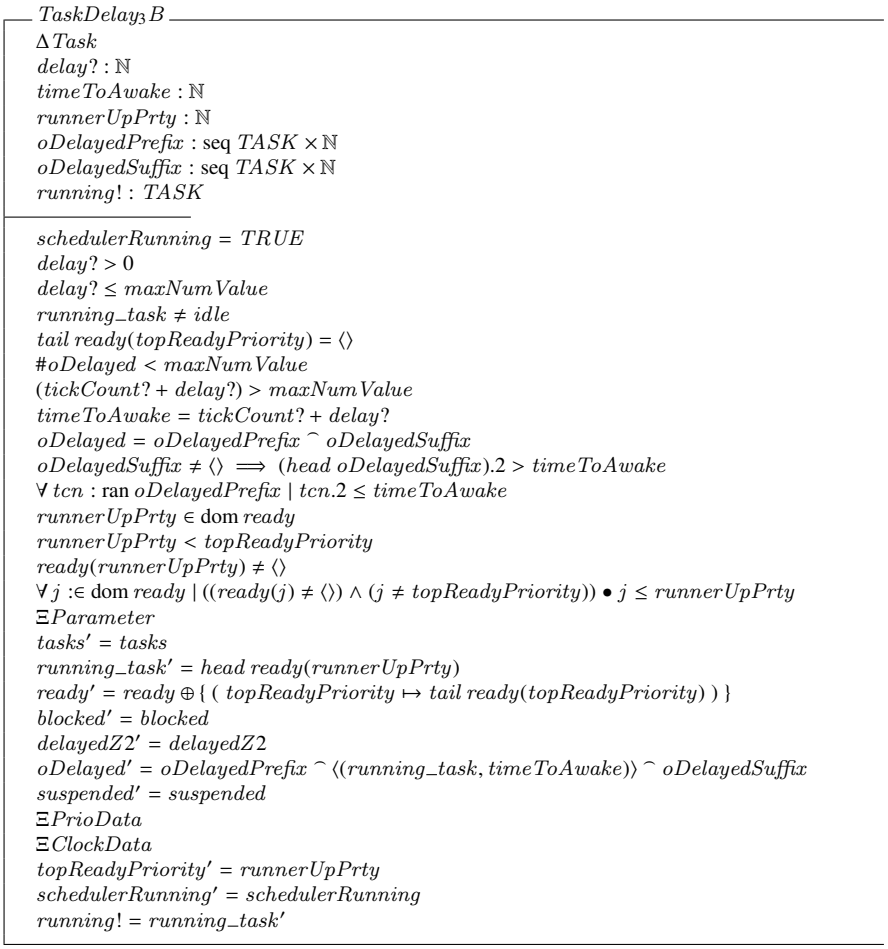


Figure 38: Operation schema $TaskDelay_3B$ specifying $vTaskDelay$ API when the value of ($tickCount$ + $delay$?) is greater than *maxNumVal*. According to a precondition running task is the only task in the sequence corresponding to the priority of running task.

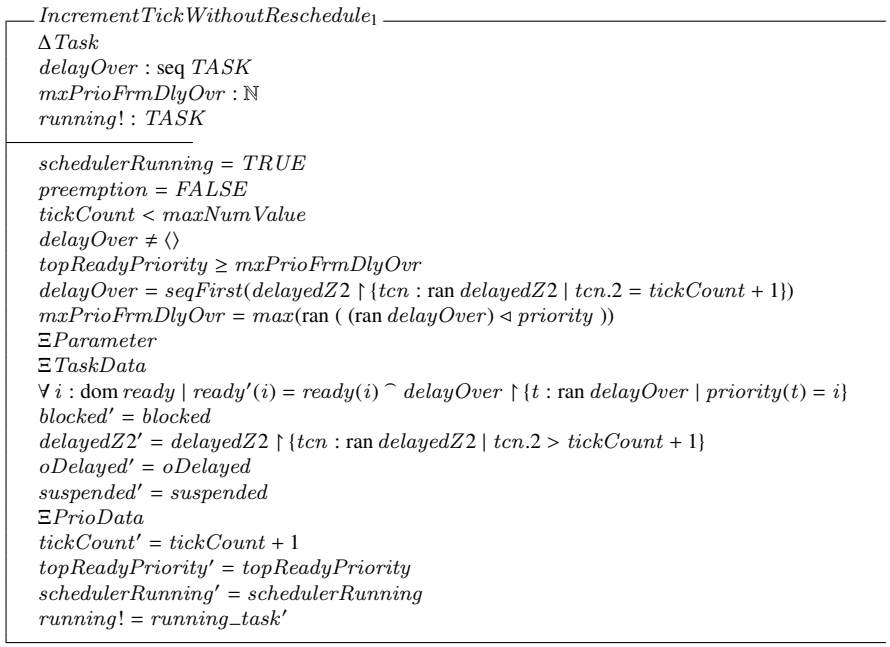


Figure 39: Operation schema $IncrementTickWithoutReschedule_1$ specifying $vTaskIncrementTick$ API when (1) the scheduler is running in the non preemptive mode, (2) each task moving from state *delayed* to state *ready* has priority less than priority of running task and (3) current value of the clock is less than *maxNumValue*.

task from the state *delayedZ2* to *ready*.

In the initial Z schema M_1 the clock was not bounded. But the clock is bounded by a constant (configurable macro) in the existing implementation of FreeRTOS. The refined Z model captures this (bounding the clock) by using an extra delayed list called *oDelayed*. Hence the clock increment is modular in M_2 w.r.t. the maximum clock value (same as $maxNumValue$). The operations schemas modelling clock increment will update the list *delayedZ2* or *oDelayed* depending on the current value of clock.

The schemas of Figures 39 and 40 specify *vTaskIncrementTick* for the situation which do not require to do a context switch. Both of these schemas assert that the new value of the clock is within the permissible limit and hence need to process the list *delayedZ2*. These schemas assume that the system is running in the *non preemptive* mode and is asserted as a precondition. Then context switch is required only if the clock update moves one or more task of priority higher than running task's priority from *delayedZ2* to *ready*.

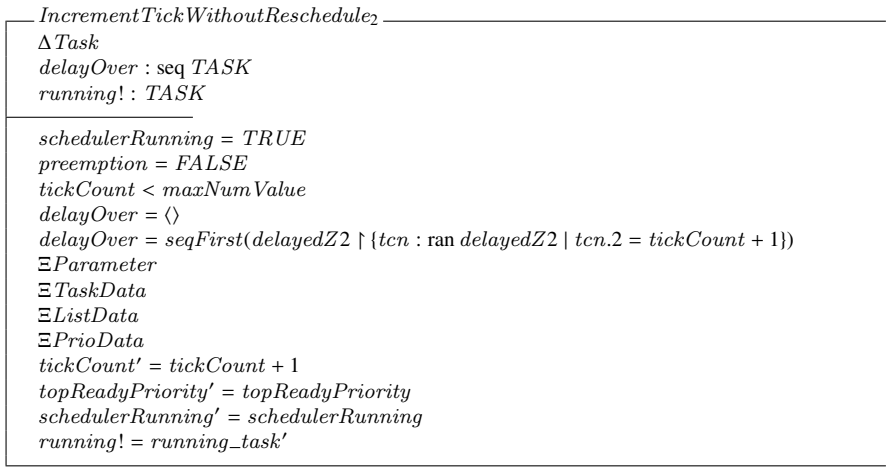


Figure 40: Operation schema $IncrementTickWithoutReschedule_2$ specifying $vTaskIncrementTick$ API when (1) the scheduler is running in the non preemptive mode, (2) no task needs to be moved from state *delayed* to state *ready* and (3) current value of the clock is less than *maxNumValue*.

The field *mxPrioFrmDlyOvr* of the schema in Figure 39 represents the priority of the highest priority task moving from state *delayedZ2* to the state *ready*.

The precondition of the schema in Figure 39 asserts that there are some tasks which need to be moved from *delayedZ2* to *ready*. The precondition also asserts that none of these task has priority higher than the priority of running task. This schema increments the clock value by one.

The precondition of the schema in Figure 40 asserts that no task needs to be moved from *delayedZ2* to *ready*. This schema just increments the clock value by one.

The schemas of Figures 41 and 42 specify *vTaskIncrementTick* for the situation which do not require to do a context switch. Both of these schemas assert that the current value of the clock is the maximum possible value and hence need to reset the clock to 0. This also demand interchanging the delayed lists. The state change of tasks from delayed to ready is decided by the state of the list *oDelayed*. These schemas assume that the system is running in the *non preemptive* mode and is asserted as a precondition. Then context switch is required only if the clock update moves one or more task of priority higher than running task's priority from *oDelayed* to *ready*.

The precondition of the schema in Figure 41 asserts that there are some tasks which need to be moved from *oDelayed* to *ready*. The precondition also asserts that none of these task has priority higher than the priority of running task. This

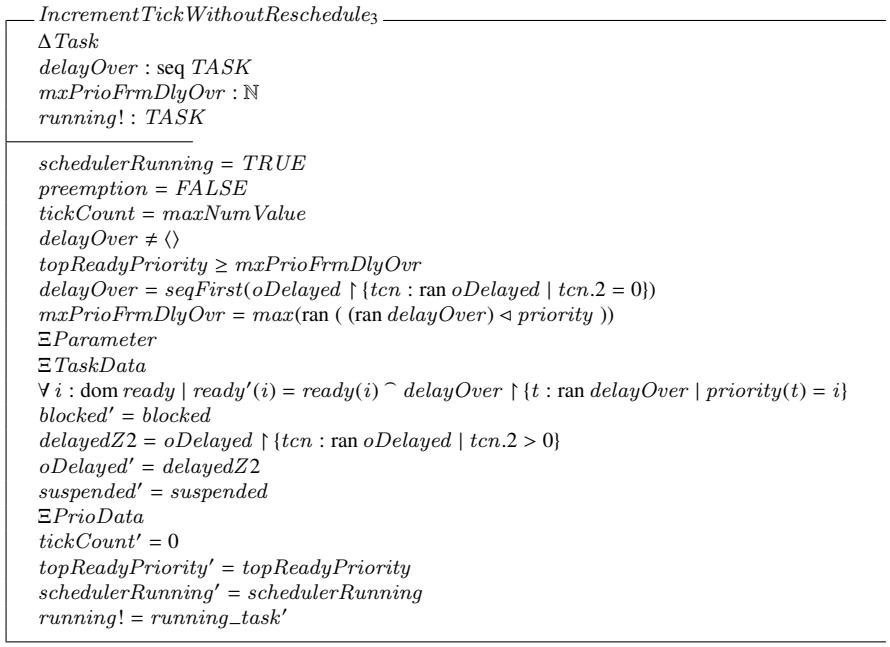


Figure 41: Operation schema $IncrementTickWithoutReschedule_3$ specifying $vTaskIncrementTick$ API when (1) the scheduler is running in the non preemptive mode, (2) each task moving from state *delayed* to state *ready* has priority less than priority of running task and (3) current value of the clock is equal to *maxNumValue*.

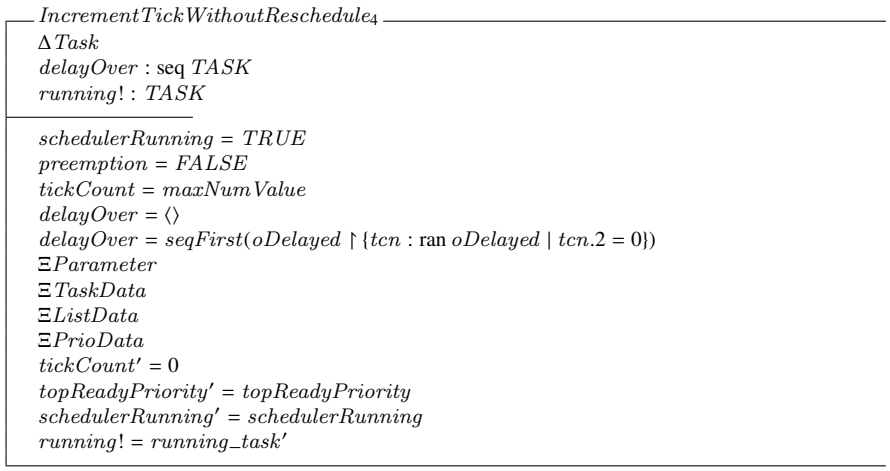


Figure 42: Operation schema $IncrementTickWithoutReschedule_2$ specifying $vTaskIncrementTick$ API when (1) the scheduler is running in the non preemptive mode, (2) no task needs to be moved from state *delayed* to state *ready* and (3) current value of the clock is equal to *maxNumValue*.

schema increments the clock value by one.

The precondition of the schema in Figure 42 asserts that no task needs to be moved from *oDelayed* to *ready*. This schema just increments the clock value by one.

The operator $\lhd$ in Z is the domain restriction operator. This operator removes those pairs from the binary relation (given as the second argument to $\lhd$) with first element not present in the first argument to $\lhd$.

The schemas of Figures 43 and 44 specify *vTaskIncrementTick* for a situation which demands a context switch. These schemas require to do a context switch because of *preemptive* mode of operation as asserted by a precondition. Both of these schemas assert that the new value of the clock is within the permissible limit and hence need to process the list *delayedZ2*.

The precondition of the schema in Figure 43 asserts that there are some tasks which need to be moved from *delayedZ2* to *ready*. The precondition also asserts that none of these task has priority higher than the priority of running task. This schema updates the state of the task lists *ready* and *delayedZ2* w.r.t. the required movement of tasks from *delayedZ2* to *ready*.

The precondition of the schema in Figure 44 asserts that no task needs to be moved from *delayedZ2* to *ready*.

The schemas of Figures 43 and 44 increment the clock value by one and ro-

Figure 43: Operation schema $IncrementTickAndReschedule_1$ specifying $vTaskIncrementTick$ API when (1) the scheduler is running in the preemptive mode, (2) each task moving from state *delayed* to state *ready* has priority less than or equal to priority of running task, (3) there exists at least one more ready task in the sequence corresponding to the priority of running task and (4) current value of the clock is less than *maxNumValue*.

$IncrementTickAndReschedule_2$

$\Delta Task$
$delayOver$: seq $TASK$
$running! : TASK$

$schedulerRunning = TRUE$
$preemption = TRUE$
$tickCount < maxNumValue$
$delayOver = \langle\rangle$
$delayOver = seqFirst(delayedZ2 \upharpoonright \{tcn : \text{ran } delayedZ2 \mid tcn.2 = tickCount + 1\})$
$tail\ ready(topReadyPriority) \neq \langle\rangle$
$\Xi Parameter$
$tasks' = tasks$
$running\_task' = head\ tail\ ready(topReadyPriority)$
$ready' = ready \oplus \{ (\ topReadyPriority \mapsto tail\ ready(topReadyPriority) \frown \langle running\_task \rangle\ ) \}$
$blocked' = blocked$
$delayedZ2' = delayedZ2$
$oDelayed' = oDelayed$
$suspended' = suspended$
$\Xi PrioData$
$tickCount' = tickCount + 1$
$topReadyPriority' = topReadyPriority$
$schedulerRunning' = schedulerRunning$
$running! = running\_task'$

Figure 44: Operation schema $IncrementTickAndReschedule_2$ specifying $vTaskIncrementTick$ API when (1) the scheduler is running in the preemptive mode, (2) no task needs to be moved from state *delayed* to state *ready*, (3) there exists at least one more ready task in the sequence corresponding to the priority of running task and (4) current value of the clock is less than *maxNumValue*.

tate the *ready* list at index $topReadyPriority$ to effect the preemption. Note that *topReadyPriority* is the priority of the running task as defined by an invariant in the schema *Task*.

The schemas of Figures 45 and 46 specify *vTaskIncrementTick* for a situation which demands a context switch. These schemas requires to do a context switch because of *preemptive* mode of operation as asserted by a precondition. Both of these schemas assert that the value of the clock overflows and hence need to process the list *oDelayed*.

The precondition of the schema in Figure 45 asserts that there are some tasks which need to be moved from *oDelayed* to *ready*. The precondition also asserts that none of these task has priority higher than the priority of running task. This schema updates the state of the task lists *ready*, *delayedZ2* and *oDelayed* w.r.t. the required movement of tasks from *oDelayed* to *ready*.

The precondition of the schema in Figure 46 asserts that no task needs to be moved from *oDelayed* to *ready*.

The schemas of Figures 45 and 46 reset the clock value to 0 and rotate the *ready* list at index $topReadyPriority$ to effect the preemption. Note that *topReadyPriority* is the priority of the running task as defined by an invariant in the schema *Task*.

The schemas of Figures 47 and 48 specify *vTaskIncrementTick* for a situation which demands a context switch because it needs to move one or more task of priority higher than running task's priority from *delayedZ2* to *ready*.

The precondition of the schema in Figure 47 asserts that the new value of the clock is within the permissible limit and hence need to process the list *delayedZ2*. This increments the clock value by one and updates the state of the task lists *ready* and *delayedZ2* w.r.t. the required movement of tasks from *delayedZ2* to *ready*. It also updates $topReadyPriority$ to the priority of the newly scheduled task.

The precondition of the schema in Figure 48 asserts that the new value of the clock overflows and hence need to process the list *oDelayed*. This resets the clock value to 0 and updates the state of the task lists *ready*, *delayedZ2* and *oDelayed* w.r.t. the required movement of tasks from *oDelayed* to *ready*.

The schemas of Figures 47 and 48 schedules the longest waiting highest priority tasked moved from the delayedZ2 state to ready state. These schemas also update $topReadyPriority$ to the priority of the newly scheduled task.

FreeRTOS provides an API *vTaskPrioritySet* to change the priority of an existing task in the system. A task can change the priority of itself or it can change the priority of other task. Changing priority may demand a context switch.

The schema in Figure 49 specifies the function *vTaskPrioritySet* in FreeRTOS when the given task holds an inherited priority (different priority values for the functions *priority* and *basePriority*). In this case only the function *basePriority* is changed to update the priority of the given task.

Figure 45: Operation schema $IncrementTickAndReschedule_3$ specifying $vTaskIncrementTick$ API when (1) the scheduler is running in the preemptive mode, (2) each task moving from state *delayed* to state *ready* has priority less than or equal to priority of running task, (3) there exists at least one more ready task in the sequence corresponding to the priority of running task and (4) current value of the clock is equal to *maxNumValue*.

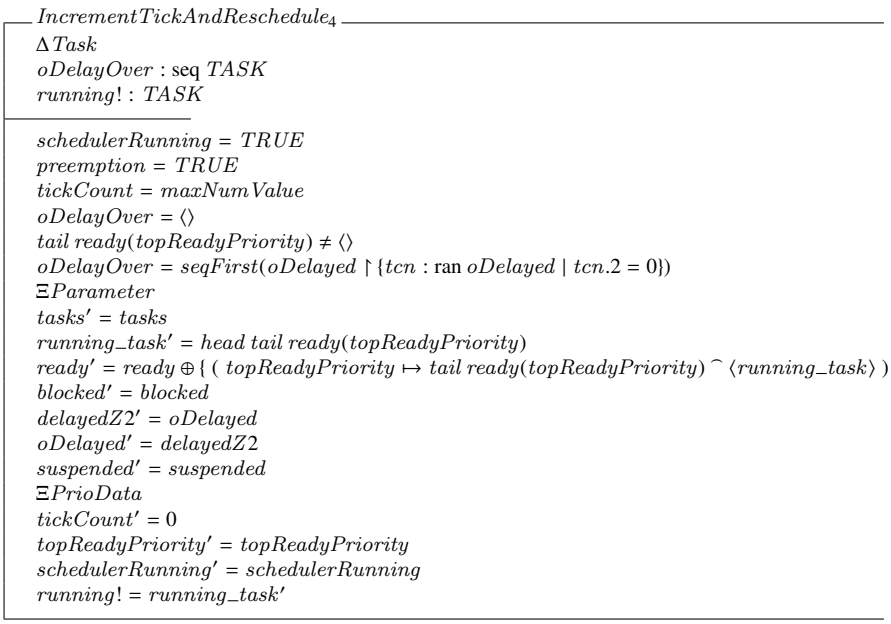


Figure 46: Operation schema $IncrementTickAndReschedule_4$ specifying $vTaskIncrementTick$ API when (1) the scheduler is running in the preemptive mode, (2) no task needs to be moved from state *delayed* to state *ready*, (3) there exists at least one more ready task in the sequence corresponding to the priority of running task and (4) current value of the clock is equal to *maxNumValue*.

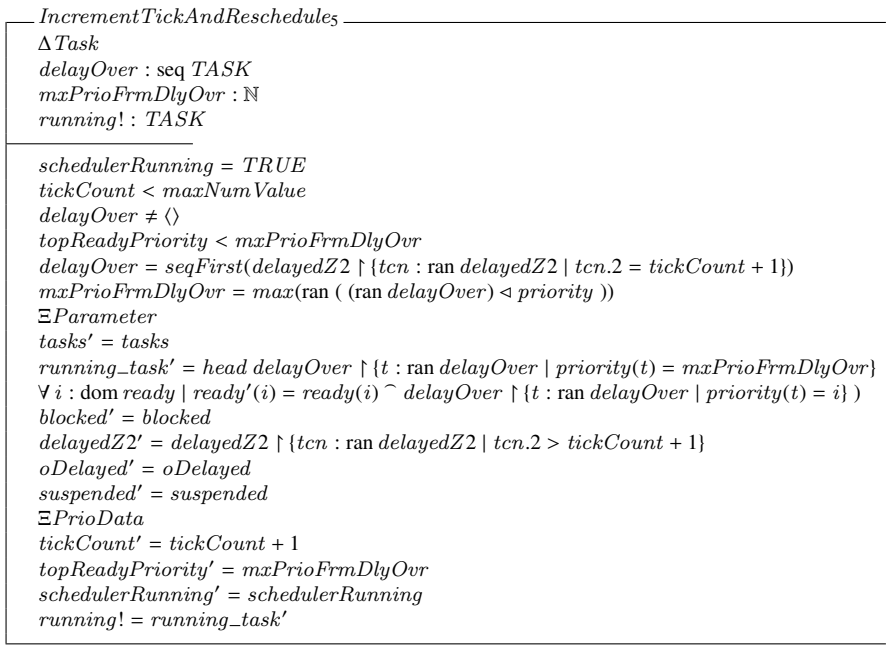


Figure 47: Operation schema $IncrementTickAndReschedule_5$ specifying $vTaskIncrementTick$ API when (1) the scheduler is running in the preemptive mode, (2) some of the tasks moving from state *delayed* to state *ready* has priority higher than the priority of running task and (3) current value of the clock is less than *maxNumValue*.

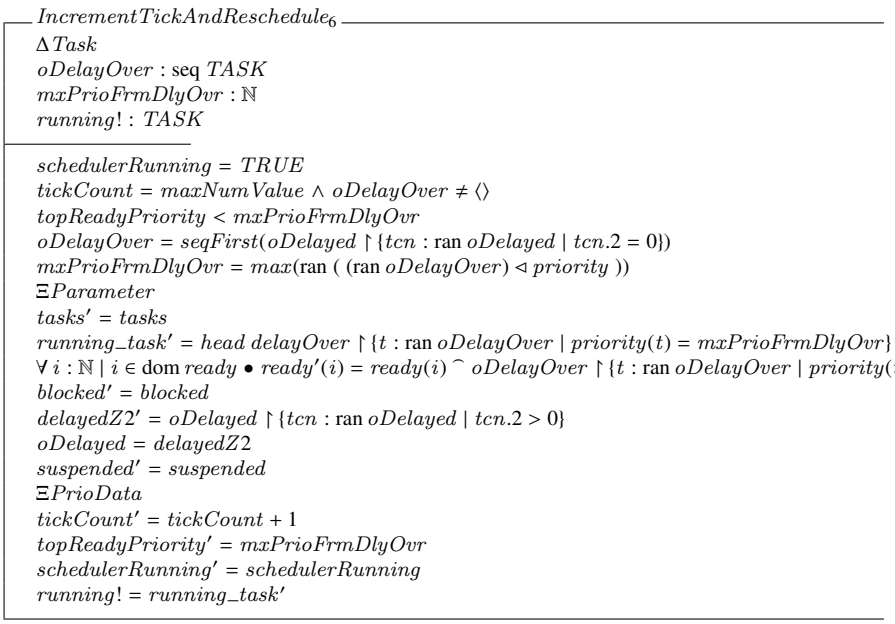


Figure 48: Operation schema $IncrementTickAndReschedule_6$ specifying $vTaskIncrementTick$ API when (1) the scheduler is running in the preemptive mode, (2) some of the tasks moving from state *delayed* to state *ready* have priority higher than the priority of running task and (3) current value of the clock is equal to *maxNumValue*.

$SetPriorityWhenInherited$

$\Delta Task$
$taskIn? : TASK$
$newPriority? : \mathbb{N}$
$running! : TASK$

$newPriority? \in \mathrm{dom}\, ready$
$schedulerRunning = TRUE$
$taskIn? \in tasks$
$basePriority(taskIn?) \neq newPriority?$
$priority(taskIn?) \neq basePriority(taskIn?)$
$\Xi Parameter$
$\Xi TaskData$
$\Xi ListData$
$priority' = priority$
$basePriority' = basePriority \oplus \{(taskIn? \mapsto newPriority?)\}$
$\Xi ClockData$
$topReadyPriority' = topReadyPriority$
$schedulerRunning' = schedulerRunning$
$running! = running\_task'$

Figure 49: Operation schema $SetPriorityWhenInherited$ specifying $vTaskPrioritySet$ when the tasks holds inherited priority.

$SetPriority_1$

$\Delta Task$
$taskIn? : TASK$
$newPriority? : \mathbb{N}$
$running! : TASK$

$newPriority? \in \mathrm{dom}\, ready$
$schedulerRunning = TRUE$
$priority(taskIn?) = basePriority(taskIn?)$
$taskIn? = running\_task$
$newPriority? > topReadyPriority$
$\Xi Parameter$
$\Xi TaskData$
$ready' = ready \oplus \{ (\ priority(taskIn?) \mapsto tail\ ready(priority(taskIn?))\ ), (\ newPriority? \mapsto \langle taskIn? \rangle\ )\ \}$
$blocked' = blocked$
$delayedZ2' = delayedZ2$
$oDelayed' = oDelayed$
$suspended' = suspended$
$priority' = priority \oplus \{(taskIn? \mapsto newPriority?)\}$
$basePriority' = basePriority \oplus \{(taskIn? \mapsto newPriority?)\}$
$\Xi ClockData$
$topReadyPriority' = newPriority?$
$schedulerRunning' = schedulerRunning$
$running! = running\_task'$

Figure 50: Operation schema $SetPriority_1$ specifying $vTaskPrioritySet$ for increasing the priority of running task.

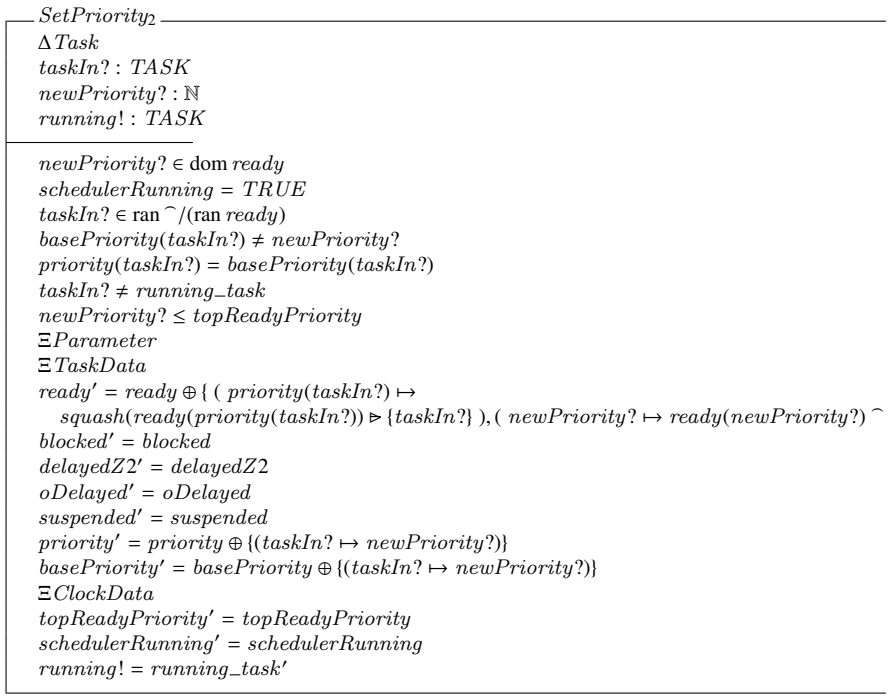


Figure 51: Operation schema $SetPriority_2$ specifying $vTaskPrioritySet$ for changing the priority of ready task to a value less than or equal to the priority of running task.

The schema in Figure 50 specifies the priority increase for running task. It updates the ready list at index equals to the priority of the given task (running task in this case) and the priority functions w.r.t. the required priority change.

The schema in Figure 51 specifies the function *vTaskPrioritySet* in FreeRTOS to change the priority of a ready task which is not running. One of the precondition specifies that the new priority is less than or equal to the priority of running task and hence no context switch is required. It updates the ready list at index equals to the priority of the given task and the priority functions w.r.t. the required priority change.

The schema in Figure 52 specifies the priority change for task whose state is not ready. The priority functions are updated in accordance with the required priority change.

The schema in Figure 53 specifies the function *vTaskPrioritySet* in FreeRTOS to decrease the priority of running task. The preconditions specify that running task is the only task in the ready list at the index $topReadyPriority$ and the new

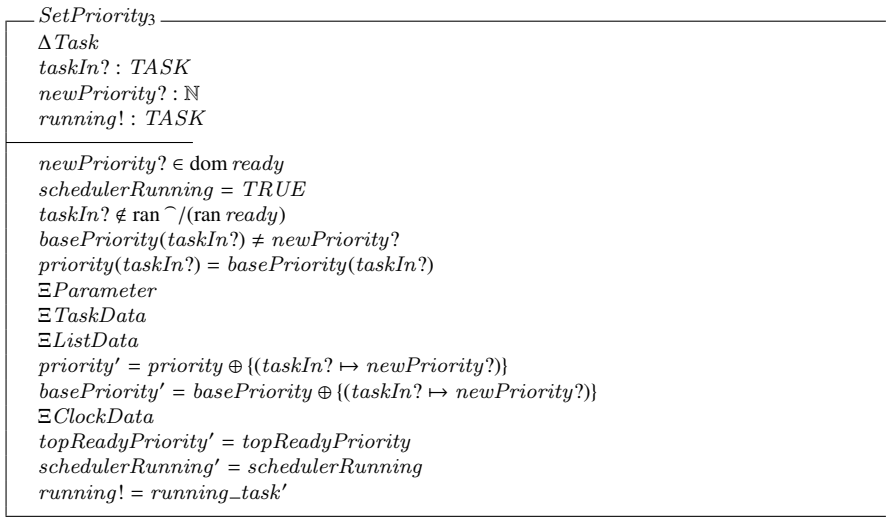


Figure 52: Operation schema $SetPriority_3$ specifying $vTaskPrioritySet$ for changing the priority of task which is not ready.

priority for running task is higher than the priorities of any other ready task. Hence no context switch is required. It updates the ready list at index equals to the priority of the given task and the priority functions w.r.t. the required priority change.

The schema in Figure 54 specifies the priority decrease for running task. One of the precondition specifies that the ready sequence at index $topReadyPriority$ contains at least one more task in addition to the running task and hence it needs to do a context switch. This schema schedules the longest waiting task among the other tasks in this sequence. The ready lists and priority functions are updated in accordance with the required priority change.

The schema in Figure 55 specifies the function *vTaskPrioritySet* in FreeRTOS to increase the priority of ready task which is not running. One of the precondition specifies that the new priority for the given ready task is higher than that of running task and hence the given task needs to be scheduled. The ready list and the priority functions are updated w.r.t. the required priority change and also updates $topReadyPriority$ to the priority of the newly scheduled task.

The schema in Figure 56 specifies the priority decrease for running task. Precondition in this case specifies that there is no other task in the ready list of running task. Hence this schema schedules the head of the next highest nonempty ready list. Such a nonempty list is assumed in the precondition. The ready list and the priority functions are updated w.r.t. the required priority change and also updates

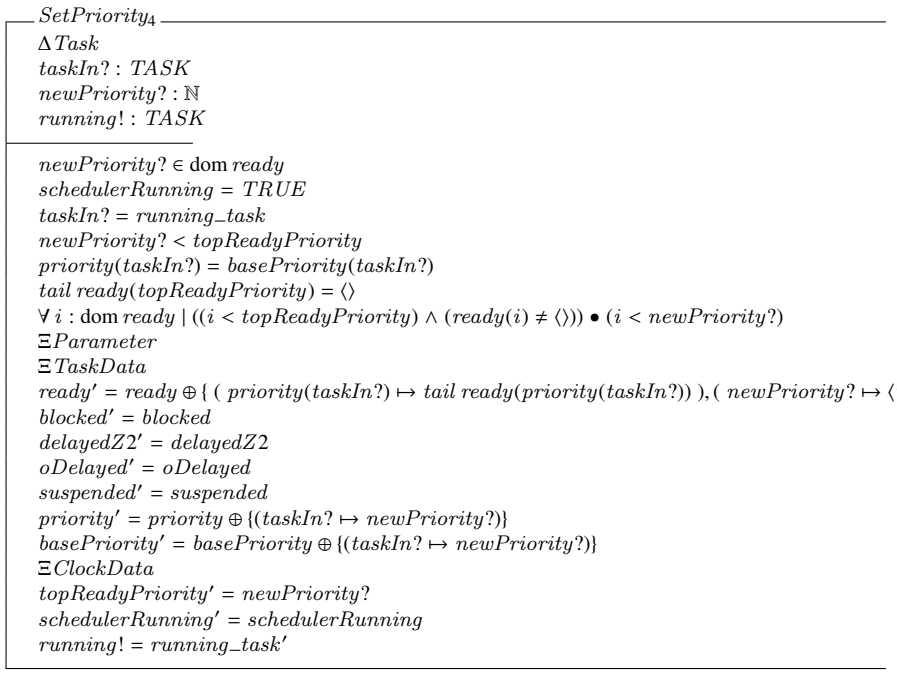


Figure 53: Operation schema $SetPriority_4$ specifying $vTaskPrioritySet$ for changing the priority of running task when (1) running task is the only task in the sequence corresponding to the priority of running task and (2) the new priority is higher than the priority of any other ready task.

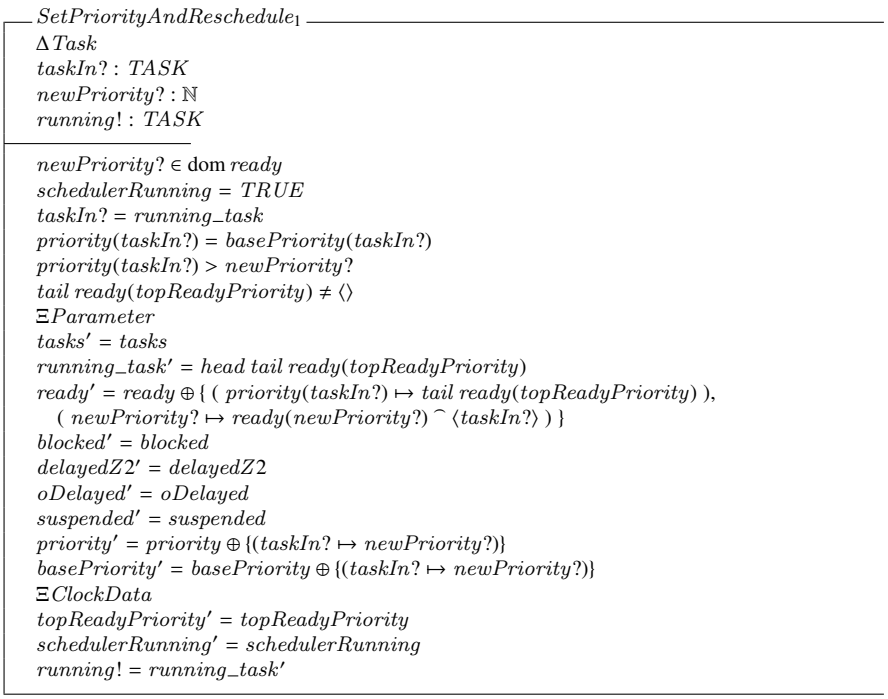


Figure 54: Operation schema $SetPriorityAndReschedule_1$ specifying $vTaskPrioritySet$ for decreasing the priority of running task when there exists at least one more task in the sequence corresponding to the priority of running task.

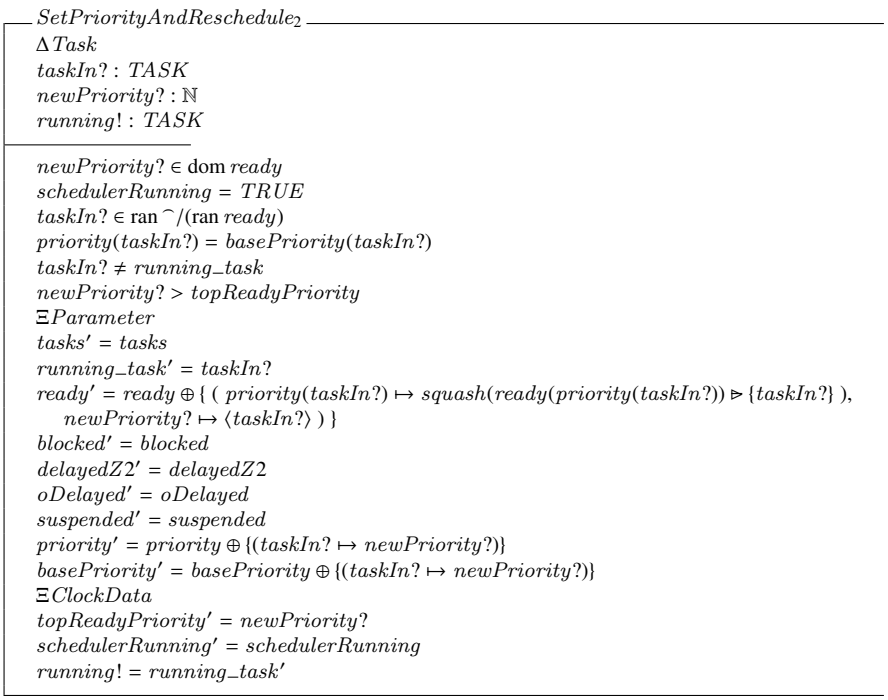


Figure 55: Operation schema $SetPriorityAndReschedule_2$ specifying $vTaskPrioritySet$ for increasing the priority of a ready task to a value higher than the priority of running task.

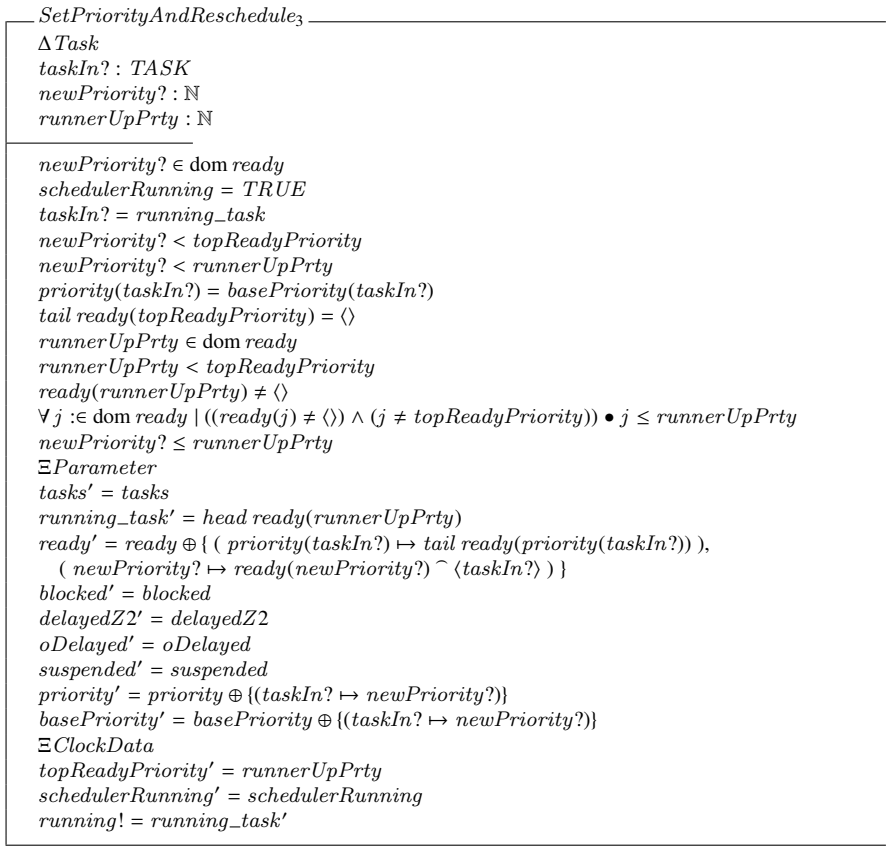


Figure 56: Operation schema $SetPriorityAndReschedule_3$ specifying $vTaskPrioritySet$ for decreasing the priority of running task when running task is the only task in the sequence corresponding to the priority of running task.

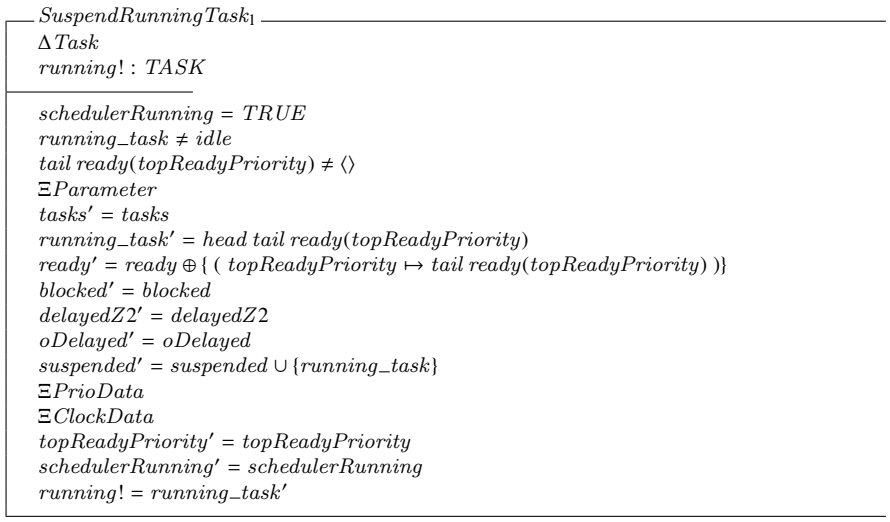


Figure 57: Operation schema $SuspendRunningTask_1$ specifying $vTaskSuspend$ API when there exists at least one more task in the sequence corresponding to the priority of running task.

$topReadyPriority$ to the priority of the newly scheduled task.

FreeRTOS provides an API *vTaskSuspend* to suspend a task. The task to be suspended will be moved from the present list to the suspended list. FreeRTOS also provides an API *vTaskResume* to resume a suspended task, which changes the state of suspended task to *ready*. Our Z model allows a state change to *suspended* only if the task to be suspended is either *running* or *ready to run*. In the other cases the task to be suspended will be either *delayed* or *blocked*. Suspend and resume operations on the *delayed/blocked* task can be virtually performed by setting and clearing a boolean flag in the task data structure while it is *delayed/blocked*.

The operation schemas of Figures 57,58 and 59 specify the function for suspending a ready task.

Precondition of the schema in Figure 57 specifies that there is one or more tasks in the ready list of running task and it schedules the oldest task among such tasks. The task lists *suspended* and *ready* are updated as required.

Precondition of the schema in Figure 58 specifies that there is no other task in the ready list of running task. Therefore this schema schedules the head of the next highest nonempty ready list and updates $topReadyPriority$ to the priority of newly scheduled task. Such a nonempty list is assumed in the precondition. The task lists *suspended* and *ready* are updated as required.

The operation schema in Figure 59 specifies the function for suspending ready

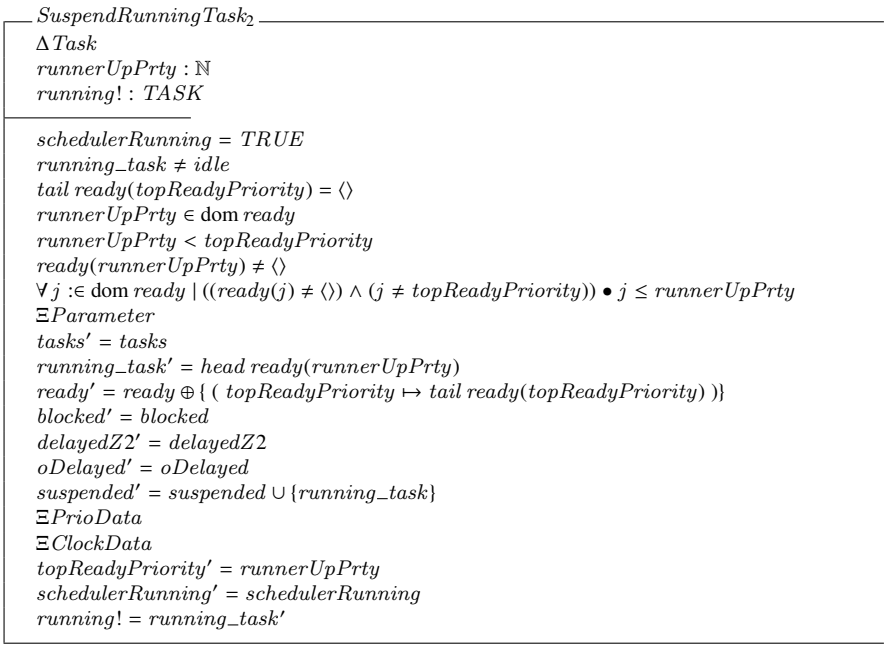


Figure 58: Operation schema $SuspendRunningTask_2$ specifying $vTaskSuspend$ API when running task is the only task in the sequence corresponding to the priority of running task.

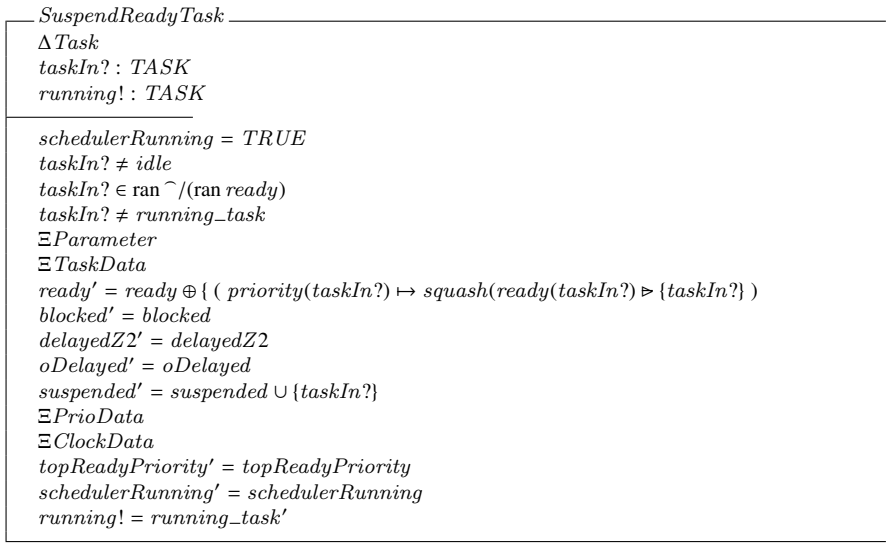


Figure 59: Operation schema $SuspendReadyTask$ specifying $vTaskSuspend$ API when task to be suspended in ready list but not the running task.

task which is not running. This schema updates the *suspended* and *ready* lists as required.

The operation schema in Figure 60 specifies the function *vTaskResume* in FreeRTOS. According to a precondition the priority of the task to be resumed is less than or equal to the priority of running task and hence context switch is not required. These schemas updates the *suspended* and *ready* lists as required.

The operation schema in Figure 61 specifies the function *vTaskResume* in FreeRTOS. In this case, a precondition specifies that the priority of the task to be resumed is greater than the priority of running task and hence context switch is required. This schema updates the *suspended* and *ready* lists as required and schedules the resumed task.

The operation schema in Figure 62 specifies the function *xTaskGetTickCount* in FreeRTOS.

References

[1] Richard Barry. **U**sing the FreeRTOS Real Time Kernel – A Practical Guide. 2010.

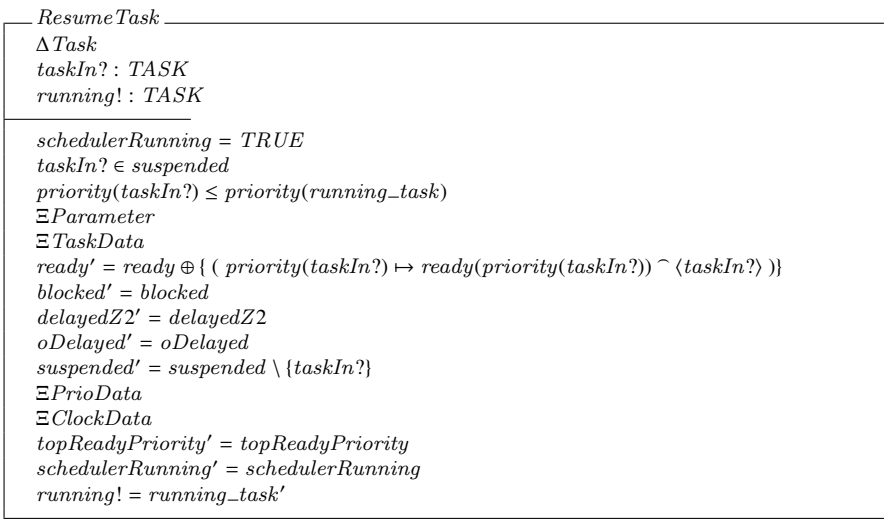


Figure 60: Operation schema $ResumeTask$ specifying $vTaskResume$ API when task to be resumed has priority less than or equal to the priority of running task.

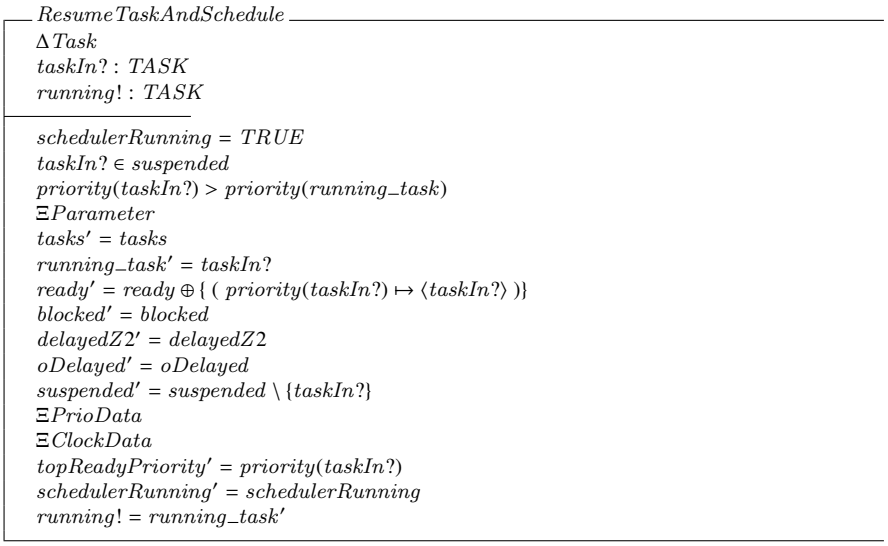


Figure 61: Operation schema $ResumeTaskAndSchedule$ specifying $vTaskResume$ API when task to be resumed has priority greater than the priority of running task.

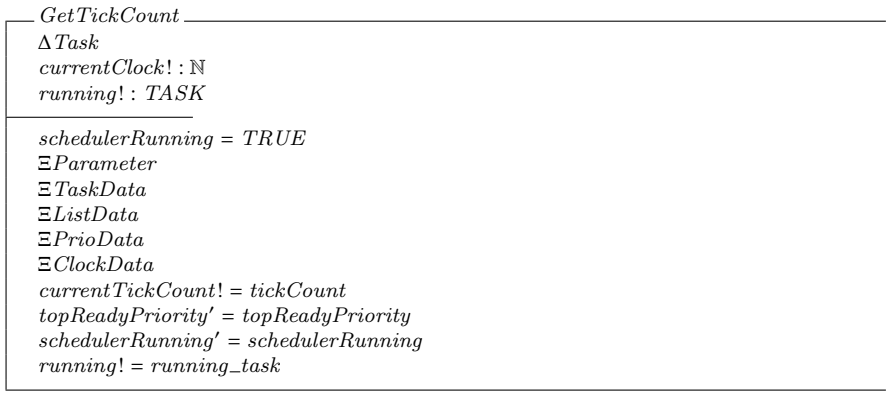


Figure 62: Operation schema $GetTickCount$ specifying $xTaskGetTickCount$ API.

[2] Jim Woodcock and Jim Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, 1996.