Formalizing FreeRTOS: First Steps

Conference Paper · August 2009

DOI: 10.1007/978-3-642-10452-7_8 · Source: DBLP

3 authors, including:

David Déharbe
ClearSy System Engineering
114 PUBLICATIONS **940** CITATIONS

SEE PROFILE

Anamaria Martins Moreira
Federal University of Rio de Janeiro
62 PUBLICATIONS **222** CITATIONS

SEE PROFILE

Formalizing FreeRTOS: First Steps

D. Déharbe, S. Galvão, and A. Martins Moreira

Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte
Natal, RN, Brazil

Abstract. This paper presents the current state of the formal development of FreeRTOS, a real-time operating system. The goal of this effort is to address a scientific challenge and is realized within the scope of the Grand Challenge on Verified Software. The development is realized with the B method. A model of the main functionalities of the FreeRTOS is now available and can be a starting point to establish an agreed formal specification of FreeRTOS that can be used by the research community.

1 Introduction

Computer Science is a fairly young discipline, but has a dramatic impact on our society and lifestyle. The pervasive nature of computing has given rise to a very large number of sub-areas and has fragmented the efforts of the research community. It seems now a good time for the community to pause and reflect to define scientific challenges that provide the opportunity for these different sub-areas to share and combine knowledge, efforts and results to achieve groundbreaking results that attend to existing needs of our societies.

One such initiative has been undertaken by the Brazilian Computer Society [1] and has identified five grand challenges. One such challenge is concerned with the *technological development of quality: dependable, scalable and ubiquitous systems*. Formal methods have shown to be a successful approach to build dependable systems. They are currently employed in applications requiring a high level of safety and integrity.

The work presented in this paper represents a small step in the direction of this challenge, but it more specifically addresses another one: the *International Grand Challenge on Verified Software* [2]. One of the activities associated to this challenge consists in setting up case studies of increasing complexity to measure and compare existing approaches to build verified software, to identify their weaknesses and how they can be improved. It is in that context that real-time operating system FreeRTOS has been proposed as a case study [3].

FreeRTOS is mainly written in C, with some parts in assembly language. It is available as a library of types and functions to build real-time, multi-tasking, embedded software. FreeRTOS is an interesting case study for many reasons. First, it has a large community of users and its verification would have a strong impact. Second, although FreeRTOS has a relatively large number of functions, its source code has medium size. Third, it is easily available, as it is open source

and it is well documented. Finally, and most importantly, modeling and verifying the kernel of an operating system is scientifically challenging [4]: for instance, the code includes many complex pointer-based operations.

The verification of a software aims at showing that it is free from errors (or to find some errors). In the context of this paper, we are interested in design errors, resulting in a discrepancy between the system behavior and its requirements. In the case of FreeRTOS, the requirements are distributed throughout the documentation, are expressed in natural language and are therefore not adequate as a source for a formal verification effort. The first step towards verifying FreeRTOS is thus to build a functional specification of its intended behavior. The goal of this paper is to present the current state of the model of FreeRTOS, which covers a significant, and essential, subset of the available functionalities. This model is available for researchers interested in contributing to the challenge of the verification of FreeRTOS.

Several formalisms are candidates to specify the functional requirements of *software*. In this work, we have chosen the B method [5,6]: it provides not only a notation, but also a framework for the verification of a specification and its refinement towards an implementation. It is similar to other well-known formal specification notations, such as VDM [7] and Z [8]. One important criterion to choose the B method is that it has a solid tool support for all the development stages.

Overview of the paper. Sections 2 and 3 lay the ground for this paper by presenting respectively the main features of FreeRTOS and the B method. Section 4 then enumerates and describes the functionalities of FreeRTOS that have been selected for modeling. The resulting functional specification is presented in Section 5. Section 6 draws conclusions and presents future work.

2 FreeRTOS

FreeRTOS is a simple, easy-to-use real-time operating system. Its source code is written in C and assembly. It is open source and has little more than 2,200 lines of code. FreeRTOS has been officially ported to most architectures for embedded systems, such as 8051, PIC, ARM and Zilog's Z80.

One key assumption of FreeRTOS is that the target system has a single processing unit. FreeRTOS provides the following services: task management, inter-task communication and synchronization, memory management, real-time events, and control of input and output devices. These services are provided as a library of types and functions that needs to be linked to the compiled code of the application being developed. Typically, this code is divided in two parts: the first one contains the code of the tasks that are going to be executed during the operation of the system, while the second contains the code responsible for the system initialization; namely, registering the tasks and starting the scheduler. Consequently, the run of an application built with FreeRTOS starts with a boot phase, to set up the different tasks and communication channels, followed by an

application execution phase, starting when the scheduler is activated: from that moment on, tasks are scheduled and executed.

2.1 Task management

Tasks form the basic computation unit in multi-tasking applications. A task has a *state*, that may be one of `running`, `ready`, `suspended` and `blocked`, a *priority*, an integer value ranging from zero up to a maximum value defined at compile time, and the *execution context* storing the call stack and the register values when the task is not executing.

Task scheduling is based on priorities. The scheduler always chooses one task with the highest priority among those `ready` tasks. A direct consequence of that policy is that the priority of the running task is always greater than or equal to that of all `ready` tasks.

Scheduling also equally shares the processing time between tasks with the same priority. Thus, if there are two or more tasks having the highest priority among the `ready` tasks, they shall equally share the processing time.

Finally, FreeRTOS automatically creates a system task, called the *idle task*, that has the lowest possible priority. This task guarantees that the processor is always executing some task and also executes some administration duties of the operating system, such as memory management.

2.2 Communication and synchronization

FreeRTOS provides message-passing communication facilities. Tasks may post messages to queues and read messages from queues. Queues have a fixed, limited capacity, defined when the task is created. Message-passing is blocking: whenever a task wants to read from an empty queue or to write to a full queue, it is blocked. There are however facilities to associate delays to queue access, or to make non-blocking accesses.

FreeRTOS also provides semaphores as a task synchronization primitive. Semaphores are actually implemented as queues with capacity one, with the convention that the semaphore is taken when the queue is empty and it is free when the queue is full. FreeRTOS also provides counting semaphores, to control the access to a resource by a maximum number of tasks. To avoid the priority inversion problem, FreeRTOS also provides mutexes with priority inheritance.

3 The B method

The B method is a model-driven design methodology to build software components reliably, in the sense that the programs produced are guaranteed to implement the corresponding functional specification. The B method consists in the following steps: first, a functional specification of the requirements, or part thereof, is developed. In B, this initial specification is called a *machine*. This initial specification is then subject to different kinds of analysis, including type

checking and theorem proving, to establish that it is implementable and that all executions are safe, in the sense that they may not reach an invalid state.

Once the specification has been built, it is used as the starting point of a series of refinements, each *refinement* resulting in an artifact providing a new model of the system. In B, refinements are usually constructed by the modeler, although automatic refinement support is now also possible [9]. Each such refinement may capture new functional requirements or introduce a more concrete description of the system, by introducing an algorithmic development or an implementable data representation. The former is called an *horizontal* refinement and the latter a *vertical* refinement [10]. Eventually, a sequence of horizontal and then vertical refinements shall lead to a fully algorithmic artifact, called an *implementation* in the B method. Such modules may then be translated to source code for programs in imperative languages such as C, Ada or Java.

The theoretical underpinnings of the B method are first-order logic, integer arithmetic, set theory, substitution calculus, and refinement theory. The different modules are written in a language called *abstract machine notation* (or AMN). An AMN module is divided into sections, each section being responsible for defining an aspect of the specification, e.g. parameters, basic types, constant values, state variables, initial states and transitions.

As an illustration, Figure 1 contains a module, called *Kernel*, specifying a system which allows to include new tasks up to a maximum number of ten. The **MACHINE** section identifies the nature (a functional specification) and the name of the module. The section **SETS** introduces a new type of entities, namely $TASK$. At this level, no further detail is provided on how this entity is going to be implemented. The **VARIABLES** section enumerates the name of the different state variables. Here, the state is composed of a single variable, named *tasks*. The **INVARIANT** section defines the possible values of the state variables: it defines their types and other restrictions that shall reflect the functional requirements of the system. Next, the **INITIALISATION** section provides a definition of the set of possible initial states of the system. Last is the **OPERATIONS** section, which is where the different types of events that the system may execute and the corresponding state transitions are defined. The example has a single operation that takes a task as parameter and adds it to *tasks*. In B, operations may have parameters (passed by value), results, and may change the value of the state variables. Operations are defined in a language called the *generalized substitution language*. The constructs of this language are syntactically similar to that of imperative programming languages, and semantically, they are predicate transformers.

In the B method, a machine must be verified to satisfy the correctness criteria stating that it is implementable, that all the states that are reachable are valid states (i.e. they satisfy the condition expressed in the invariant clause), and that all expressions appearing in the specification are well-defined. This verification consists in checking proof obligations that are automatically generated from the text of the machine. The proof obligations are formulas of first-order logic and the user is responsible for proving that they are valid.

MACHINE	**INVARIANT**	$task\_add(task) =$
$Kernel$	$tasks \in \mathbb{P}(TASK) \wedge$	**PRE**
SETS	$\mathbf{card}(tasks) \leq 10$	$task \in TASK \wedge task \notin tasks \wedge$
$TASK$	**INITIALISATION**	$\mathbf{card}(tasks) < 10$
VARIABLES	$tasks := \emptyset$	**THEN**
$tasks$	**OPERATIONS**	$tasks := tasks \cup \{task\}$
		END

Fig. 1. Functional specification of a simple task management system.

Consider the example of Figure 1. To guarantee that the machine is implementable, one needs to prove the satisfiability of the different constraints of the model. In the case of this example, one needs to show the validity of the existential quantification of the invariant:

$$\exists tasks \bullet tasks \in \mathbb{P}(TASK) \wedge \mathbf{card}(tasks) \leq 10.$$

To guarantee the correctness criterion stating that all reachable states are valid, one must check that each operation preserves the invariant: if the operation is applied to a state satisfying the invariant, and if the pre-condition of the operation is satisfied, then the resulting state must also be valid. The following formula, generated automatically by the proof obligation generator, states this property:

$$\begin{array}{l} tasks \in \mathbb{P}(TASK) \wedge \mathbf{card}(tasks) \leq 10 \quad \wedge \\ task \in TASK \wedge task \in tasks \wedge \mathbf{card}(task) < 10 \Rightarrow \\ (tasks \cup \{task\}) \in \mathbb{P}(TASK) \wedge \mathbf{card}(tasks \cup \{task\}) \leq 10. \end{array}$$

Finally, it is necessary to show that all the expressions occurring in the specification are well-defined. In the case of the example, one must show that $tasks$ is a finite set in every context where the expression $\mathbf{card}(tasks)$ is evaluated.

The proof of these verification conditions is performed either by automatic theorem provers, or manually, by issuing commands to an interactive theorem prover. Typically, the automatic theorem prover manages to discharge a significant part of the verification conditions. The remaining conditions are either valid and the user must be able to build a proof of their validity, or are not valid. In the former case, it might happen that the user cannot build the proof, as the prover is inherently incomplete; he has then the choice of including additional rules or to check the condition manually and take responsibility for the verdict. In the latter case, the specification has some error and must be corrected or the formula cannot be proved. In the case of an erroneous specification, the information provided by the interactive theorem prover is often helpful to locate the error. Eventually, the user shall reach a point where all verification conditions have been proved and the refinement process may be initiated.

Note that the functional model may also be used to derive manually an implementation in a programming language. Moreover the functional specification may also be used as a reference to generate tests [11] of the implementation.

An example of refinement is presented in Figure 2. The state variable *tasks* is no longer a set of tasks but a sequence of tasks. Sequences are pre-defined in AMN and the operators **ran** and $\rightarrow$ return respectively the contents of the sequence (as a set) and addition of an element to the end of the sequence. The B method also defines a set of verification conditions which, when proved, guarantee that the refinement is correct with respect to the initial specification.

REFINEMENT *Kernel_r*
REFINES *Kernel*
VARIABLES *tasks_r*
INVARIANT $tasks\_r \in \mathbf{seq}(TASK) \wedge \mathbf{ran}(tasks\_r) = tasks$
INITIALISATION $tasks\_r := []$
OPERATIONS
$task\_add(task) =$
BEGIN
$tasks\_r := task \rightarrow tasks\_r$
END

Fig. 2. Refinement of the machine *Kernel* (Figure 1).

4 Overview of the modeling

Modeling a complex system with the B method may be facilitated by taking into account the following remarks:

1. Parts of the functional requirements may be abstracted in the initial specification. Such requirements may be introduced later, by means of horizontal refinements, or by extending the specification. In order to adopt this approach, one must first plan a sequence of incremental modeling steps, each introducing additional entities and functionalities of FreeRTOS. Such steps are described in Section 4.3.
2. When requirements do not present interdependency, they may specified in different modules. These modules will then be combined using the composition mechanism of the B method (e.g inclusion, vision, etc.) The modular structure of the model is presented in Section 5.

In system development projects, (formal) specifications are usually performed in the initial stages. In the experience reported in this paper, the system already exists, its functionalities have been identified and implemented. The presented model is the result of the analysis of the documentation of the system as well as of the source code of its implementation.

Based on an informal analysis of FreeRTOS documentation and source code, we planned an incremental construction of the model. Such increments are presented in section 4.3.

The main classes of entities provided by FreeRTOS are tasks, message queues, co-routines, semaphores and mutexes, and each such class has an associated set

of functions. However in the case of FreeRTOS, semaphores are nothing more than specialized message queues. Therefore, to build a first functional model of FreeRTOS, two basic kinds of entities were initially chosen for formalization: tasks and message queues, which form the basic mechanism for task communication and synchronization.

4.1 Tasks

Functions manipulating tasks can be divided into the functions that manage the tasks themselves and those that control the scheduler.

The task management functions that we have modeled are task creation (`xTaskCreate`), task destruction (`xTaskDelete`), an accessor to get the priority of a task (`uxTaskPriorityGet`), task suspension (`vTaskSuspend`), resumption of a suspended task, taking it to a `ready` state (`vTaskResume`), changing the priority of a task (`vTaskPrioritySet`), interruption of a task for a given time period, starting from the moment the function was called (`vTaskDelay`), or from the moment the task was resumed (`vTaskDelayUntil`).

With respect to the scheduling aspects, we have modeled functions to: access the currently executing task (`xTaskGetCurrentTaskHandle`), access to the state of the schedule, which may be `executing`, `suspended` or `uninitialized` (`xTaskGetSchedulerState`), get the number of existing tasks (`uxTaskGetNumberOfTasks`), get the time elapsed since the scheduler was initialized (`xTaskGetTickCount`), initiate the scheduler and start the so-called *idle* task (`vTaskStartScheduler`), finalize the activities of the scheduler and put it back in the uninitialized state, deleting all the entities created (`vTaskEndScheduler`), suspend the scheduler (`vTaskSuspendAll`), and resume the scheduler (`xTaskResumeAll`).

4.2 Message queues

We have modeled the following functions related to message queues: construction of a new, empty, queue (`xQueueCreate`), sending a message to a queue (`xQueueSend`), sending a message to the back of a queue (`xQueueSendToBack`) or to the front of a queue (`xQueueSendToFront`), to retrieve a message from the front of a queue (`xQueueReceive`), to read a message from the front of a queue, without removing it (`xQueuePeek`), and to delete a message queue. The presented model does not take into account the (fixed) capacity of the queues, resulting in non-deterministic models of these functions.

4.3 Increments in the model

Once the basic funcionalities of the system have been identified (namely, tasks and message queues), we identified modeling steps such that they could be defined in an incremental fashion:

1. **Basic model:** In this first step, we considered mainly the behavior of the functions related to the state of the tasks and the transitions between such states. The notion of priority was, at this level, left abstract. Also, the state of the scheduler was defined as well as the concept of elementary timing events called *ticks* in FreeRTOS. Message queues were also modeled, and their size was left abstract. In order to be able to abstract notions such as queue size and message priority, operations depending on these were defined non-deterministically.
2. **Priority**: In this second step, task priority was effectively taken into account in the model. The main consequence is that functions resulting in the scheduling of a new task were refined into more deterministic versions.
3. **Mutexes**: The third stage will consist in specifying this mechanism, that allows synchronizing tasks without provoking priority inversions.
4. **Queue size:** The fourth step shall consist in removing non-determinism related to queue sizes and actually specify the behavior related to the requirements with respect to full or empty queues.
5. **Addition of non-elementary entities:** We have already mentioned that a semaphore can be viewed as a message queue. Modeling semaphores and the related requirements will be performed last, by using the definitions already available for message queues.

5 The functional model

This section presents the first two steps to build the model of FreeRTOS as described in Section 4. The resulting model thus includes tasks, message queues, scheduling, and priorities. All the models presented in this section have been developed and verified using Atelier B 4.0 [12].

Even though only parts of the requirements are considered, the estimated size of the resulting model seemed large enough to consider a modular structure of the specification. The components of this structure are:

- The *Config* machine contains auxiliary definitions of constants that need to be be instantiated when building an application on top of FreeRTOS; for instance, the number of priority levels is configurable. Our model simply defines the domain of these constants.
- The *Types* machine declares the types of the different entities of the model of the system such as tasks, queues, messages, return codes.
- The *Task* machine defines the state variables modeling the tasks in the system, as well as the corresponding elementary functions.
- The *Queue* machine has a role similar to the *Task* machine, but related to message queues.
- The *Scheduler* machine is a very simple machine that justs maintains the current state of the scheduler.
- The machine *FreeRTOSBasic* includes an instance of the three machines *Task*, *Queue* and *Scheduler* and defines models of elementary message passing functions.

– Finally, the machine *FreeRTOS* includes an instance of *FreeRTOSBasic* and defines models of high-level message-passing functions. The intermediate machine *FreeRTOSBasic* is necessary as B does not allow operations in a machine to refer to operations in the same machine.

5.1 Tasks

The machine *Task* defines the entities and operations related to tasks (see excerpts in Figure 1). Several modeling approaches are possible: using disjoint sets, or using a state function mapping tasks to an enumerated set. We chose the former, as it allows expressing the invariant using simple sets and is thus easier to analyze using one of the available theorem provers.

The state variable *active* indicates whether the operating system is active or not (i.e. it is in the initialization phase of an instance of the system). Variable *tasks* represents all the created tasks. In addition, *running*, *ready*, *blocked* and *suspended* represent respectively the currently scheduled task, the set of tasks ready to be scheduled, the set of blocked tasks and the set of suspended tasks. Finally, variable *idle* represents the idle system task.

VARIABLES
$active, tasks, blocked, running, ready, suspended, idle$
INVARIANT
$active \in BOOL \land tasks \in \mathbb{F}(TASK) \land running \in TASK \land idle \in TASK$
$\land\, blocked \in \mathbb{F}(TASK) \land ready \in \mathbb{F}(TASK) \land suspended \in \mathbb{F}(TASK)$

The invariant includes also constraints to model several requirements: (1) a task may be in a single state at any time; (2) while the scheduler has not been activated, tasks are ready to execute; when the scheduler has been activated, (3) the idle task is always either ready to execute or executing, and (4) there is always a running task.

$$
\begin{array}{l}
blocked \subseteq tasks \land ready \subseteq tasks \land suspended \subseteq tasks \land \\
ready \cap blocked = \emptyset \land blocked \cap suspended = \emptyset \land suspended \cap ready = \emptyset \land \\
(active = FALSE \Rightarrow tasks = ready) \land \\
(active = TRUE \Rightarrow (idle = running \lor idle \in ready) \land \\
\qquad\qquad\qquad\quad running \notin (blocked \cup ready \cup suspended) \land \\
\qquad\qquad\qquad\quad tasks = \{running\} \cup suspended \cup blocked \cup ready)
\end{array}
$$

In addition, the *Task* machine contains basic operations that model the elementary changes to the system state with respect to tasks and the scheduler. Such operations are used to specify the functions of FreeRTOS related to tasks. There is a total of twelve such elementary functions, from which two will be presented here[1].

The creation of a new task is specified by the operation *t_create*. This operation can only be applied when the scheduler has not been initialized. It takes

[1] The full models and the corresponding interactive proofs are freely available at `http://code.google.com/p/freertosb/source/browse`.

as parameter the priority of the task, which will be taken into account in a refinement, and creates and returns a new task entity which is initially ready to execute:

$$
\begin{array}{l}
result \longleftarrow t\_create(priority) = \\
\textbf{PRE} \\
\quad priority \in PRIORITY \wedge \\
\quad active = FALSE \\
\textbf{THEN} \\
\quad \textbf{ANY } task \quad \textbf{WHERE} \\
\qquad task \in TASK \wedge task \notin tasks \\
\textbf{THEN} \\
\quad tasks := \{task\} \cup tasks \parallel \\
\quad ready := \{task\} \cup ready \parallel \\
\quad result := task \\
\textbf{END} \\
\textbf{END};
\end{array}
$$

The second operation shown here is *t_startScheduler* that specifies the state transition when the scheduler is activated. This corresponds to the change from the initialisation phase to the execution phase of a FreeRTOS application. In that phase, the system task *idle* is created and the scheduler chooses a task for execution. Again, recall that the priority has not been taken into account at this level of abstraction and is the subject of a further refinement. It is here left non-deterministic:

$$
\begin{array}{l}
t\_startScheduler = \\
\textbf{PRE} \\
\quad active = FALSE \\
\textbf{THEN} \\
\quad active := TRUE \parallel \\
\quad blocked, suspended := \emptyset, \emptyset \parallel \\
\quad \textbf{ANY } idle\_task \textbf{ WHERE} \\
\qquad idle\_task \in TASK \wedge \\
\qquad idle\_task \notin tasks \\
\quad \textbf{THEN} \\
\qquad tasks := \{idle\_task\} \cup tasks \parallel \\
\qquad idle := idle\_task \parallel \\
\qquad \textbf{ANY } task \textbf{ WHERE} \\
\qquad\quad task \in ready \cup \{idle\_task\} \\
\qquad \textbf{THEN} \\
\qquad\quad running := task \parallel \\
\qquad\quad ready := (ready \cup \{idle\_task\}) - \{task\} \\
\qquad \textbf{END} \\
\quad \textbf{END} \\
\textbf{END};
\end{array}
$$

The behavior of the task-related functions of FreeRTOS has then been modeled using these elementary operations. Here, we only show the function specification of the function *xTaskCreate*, that provides the task creation functionality in FreeRTOS. This specification uses the previously presented operation *t_create*. The *return* value of the function *xTaskCreate* indicates if the operation succeeded or failed (for instance due to a lack of available memory) and a handler to the new task is asssigned to to parameter *handler*, passed by reference.

$$
\begin{array}{l}
result, handle \longleftarrow \\
\qquad xTaskCreate(\ code, name, \\
\qquad\qquad\qquad stackSize, params, \\
\qquad\qquad\qquad priority) = \\
\textbf{PRE} \\
\quad code \in TASK\_CODE \wedge \\
\quad name \in NAME \wedge \\
\quad stackSize \in NATURAL \wedge \\
\quad params \subset PARAMETER \wedge \\
\quad priority \in PRIORITY \wedge \\
\quad scheduler = NOT\_STARTED \wedge \\
\textbf{THEN} \\
\quad \textbf{CHOICE} \\
\qquad handle \leftarrow \\
\qquad\quad \textbf{t\_create}(priority) \parallel \\
\qquad result := pdPASS \\
\quad \textbf{OR} \\
\qquad result := errMEMORY \parallel \\
\qquad handle :\in TASK \\
\quad \textbf{END} \\
\textbf{END}
\end{array}
$$

Note that in the notation of the B method, parameters are always passed by value, and operations may return multiple values. In the C implementation of FreeRTOS, operations such as xTaskCreate return more than one value and use pointer typed parameters to store these additional results. Wherever this is the case, the functional model includes an additional return parameter.

5.2 Message queues

The machine *Queue* defines the basic functionality to handle message queues. There are two types of entities: $QUEUE$, the queues, $ITEM$, the messages. Its state is formed by a variable *queues* representing queues, and the variables *items*, *sending* and *receiving* associating each queue with its contents, the tasks waiting to write in the queue, and the tasks waiting to read from the queue, respectively.

VARIABLES **INVARIANT**

$queues$,	$queues \in \mathbb{P}(QUEUE) \wedge$
$items$,	$items \in QUEUE \pfun \mathbb{P}(ITEM) \wedge \mathbf{dom}(items) = queues \wedge$
$receiving$,	$receiving \in QUEUE \pfun \mathbb{P}(TASK) \wedge \mathbf{dom}(receiving) = queues \wedge$
$sending$	$sending \in QUEUE \pfun \mathbb{P}(TASK) \wedge \mathbf{dom}(sending) = queues$

The *Queue* machine also contains operations that define the basic functionality to manipulate the message queues. There is a total of six such operations. For instance, the operation *sendItem* defines the inclusion of a new message *item* at position *pos* of *queue*, and destination *task*:

$sendItem(queue, item, task, pos) =$
PRE
$queue \in queues \wedge$
$item \in ITEM \wedge$
$task \in TASK \wedge$
$pos \in COPY\_POSITION \wedge$
$task \in receiving(queue)$
THEN
$items(queue) :=$
$items(queue) \cup \{item\} \parallel$
$receiving(queue) :=$
$receiving(queue) - \{task\}$
END

Such low-level operations are used to specify the behavior of section of the FreeRTOS API dealing with communication. There are basically two classes of functions: one for read access, and one for write access. They can all be specified by means of two basic operations: *xQueueGenericSend* and *xQueueGenericReceive*, which, in our model, are defined in the machine *FreeRTOSBasic*. For instance, the operation *xQueueGenericSend* specifies a generic write access of a message i in a queue q. This operation is also parameterized by the access position *pos* and the maximum number of ticks *wait* that the sending task may be blocked waiting for the queue. The sending task is always the running task. Three different behaviors are possible. First, if the queue is already full, then the running task is inserted in the set of tasks waiting to write on the queue, and a deadline is associated to this task with operation *t_delayTask*. In this scenario, the result is the constant *pdTRUE*. Second, if the queue is full, but the task is not willing to wait, the operation returns the constant *errQUEUE_FULL*. Finally, in case the destination task is already blocked

waiting to read from this queue, then this task is unblocked, and the operation returns *pdPASS*. Note that the capacity of the queues is not part of the abstract model, which is why the specification of this function is non-deterministic. This aspect will be included in the specification through a refinement.

```
res ← xQueueGenericSend(q, i, wait, pos) =
  PRE
    q ∈ queues ∧ i ∈ ITEM ∧ wait ∈ TICK∧
    pos ∈ COPY_POSITION∧
    active = TRUE ∧ running ≠ idle
  THEN
    CHOICE
      IF wait > 0 THEN
        q_insertTaskWaitingToSend(q, running) ||
        t_delayTask(wait) ||
        res := pdTRUE
      ELSE
        res := errQUEUE_FULL
      END
    OR
      ANY t WHERE
        t ∈ TASK ∧
        t ∈ blocked ∧
        t ∈ receiving(q)
      THEN
        q_sendItem(q, i, t, pos) ||
        t_unblock(t) ||
        res := pdPASS
      END
    END
  END
```

Finally, the machine *FreeRTOS* instantiates these generic operations to specify the behavior of FreeRTOS' functions providing task communication facilities. For instance, the operation *xQueueSend* specifies the behavior of the homonym FreeRTOS function, one of the three message sending variants in the API:

```
res ⟵ xQueueSend(q, i, w) =
PRE
  q ∈ queues ∧ i ∈ ITEM ∧ w ∈ TICK∧
  active = TRUE ∧ running ≠ idle
THEN
  res ← xQueueGenericSend(q, i, w, queueSEND_TO_BACK)
  END
END
```

5.3 Taking priorities into account

The functional requirements state that the running task should have a priority greater or equal than all the ready tasks. In order to take into account such requirement, the invariant needs to be strengthened to define task priorities and to specify the desired property. From the methodological viewpoint, starting from the first version of the *Task* machine (described in Section 5.1), we can either define a new version, or create a refinement. We chose the latter solution and we describe it now.

We defined a refinement module called *Task_r*. In it, we defined a type *PRIORITY* that represents tasks priorities. The state variable *prio* maps each task to its priority and the invariant states that when the scheduler has been initialised, no ready task has a priority greater than the running task. We also

include restrictions on the priority of the idle task, which is the lowest possible priority.

CONSTANTS
$MAX\_P, IDLE\_P$
PROPERTIES
$PRIORITY = 0..(MAX\_P - 1) \wedge$
$MAX\_P > 0 \wedge IDLE\_P = 0$
VARIABLES
$prio$

INVARIANT
$prio \in TASK \pfun PRIORITY \wedge$
$\mathbf{dom}(prio) = tasks \wedge$
$(active = TRUE \Rightarrow$
$prio(idle) = IDLE\_P \wedge$
$\forall t.(t \in ready \Rightarrow prio(t) \leq prio(running)) \wedge$
$\forall t.(t \in ready \Rightarrow IDLE\_P \leq prio(t)))$

Most of the operations of the *Task* machine involve defining a new running task, and these operations need to be refined to maintain the new invariant. In order to simplify the definition of these refined operations, a scheduling function was introduced. It takes as input a set of tasks and a function mapping tasks to their priorities, and it returns those given tasks that have the highest priority:

CONSTANTS
$schedule\_p$
PROPERTIES
$schedule\_p : (\mathbb{F}(TASK) \times (TASK \pfun PRIORITY)) \pfun \mathbb{F}(TASK) \wedge$
$schedule\_p = \lambda(tasks, prio) \bullet$
$(tasks : \mathbb{F}(TASK) \wedge prio : TASK \pfun PRIORITY \wedge tasks \neq \emptyset \wedge tasks \subseteq \mathbf{dom}(prio)$
$\mid \quad tasks \cap prio^{-1}(\max(prio[tasks])))$

To illustrate the refinement of the operations, we present the case of the operations *t_create* and *t_startScheduler*:

$result \longleftarrow t\_create(priority) =$
PRE
$priority \in PRIORITY \wedge$
$active = FALSE$
THEN
ANY $task$ **WHERE**
$task \in TASK \wedge task \notin tasks$
THEN
$tasks := tasks \cup \{task\} \parallel$
$prio := prio \cup \{task \mapsto priority\} \parallel$
$ready := ready \cup \{task\} \parallel$
$result := task$
END
END

$t\_startScheduler =$
BEGIN
$active := TRUE \parallel$
$blocked, suspended := \emptyset, \emptyset \parallel$
ANY i **WHERE**
$i \in TASK \wedge$
$i \notin tasks$
THEN
$tasks := tasks \cup \{i\} \parallel$
$prio := prio \cup \{i \mapsto IDLE\_P\} \parallel$
$idle := i \parallel$
ANY t **WHERE**
$t \in TASK \wedge$
$(ready = \emptyset \Rightarrow t = i) \wedge$
$(ready \neq \emptyset \Rightarrow t \in ready \wedge$
$t \in schedule\_p(ready, prio)$
THEN
$running := t \parallel$
$ready := (ready \cup \{i\}) - \{t\}$
END
END
END

In *t_create*, a substitution was added to update the information on the new task priority, and in *t_startScheduler*, *idle* is registered to have priority 0 and the new running task is selected among those ready tasks with highest priority (or *idle*, in case there are no ready tasks waiting to be executed).

5.4 Comments on the verification of the models

One of the main features of the B method is the tool support for project management, syntactic verification, and semantic analysis of the produced artifacts. In particular, the semantic analysis produces proof obligations the verification of which guarantees: (1) all expressions appearing in the text of the different artifacts are well-defined, (2) the logic consistency of the specification and its refinements. The development environment thus includes support for the construction of the proofs, by providing a number of theorem provers. However, due to the incomplete nature of the specification logic, as well as the computational complexity of finding proofs, human intervention is needed to establish part of the proofs. This is a time-consuming activity that pays off in two ways. First, when confronted with a proof obligation that is not valid, the developer has access to the context where such proof obligation was generated and has clues as where the artifact needs to be corrected. Second, when all proof obligations have been successfully validated, then the user has a very strong confidence in its models.

To give an idea of the effort needed to establish the correctness of the development, Table 1 provides the number of proof obligations generated for each artifact[2]. This table does not include however the effort needed to reach consistent models, as several iterations were needed to produce a correct definition of the invariant and of the operations.

6 Conclusion and future work

This paper presented the first steps of a formal modeling, using the B method, of a significant part of the real-time operating system FreeRTOS. This model provides a functional specification of the operations related to task management and message queues. This effort was initiated in response to the challenge set by Jim Woodcock to the Brazilian community on Formal Methods [3] to contribute with this case study to the *Verified Software Repository* [13], as part of the *International Grand Challenge on Verified Software*. Thus, a first contribution of this work is the execution of a case study for the development of a verified model of a moderately complex software library. We have already extended the presented model to specify semaphores and related functions; next, we will include the definition of functional specifications for mutexes and refine the scheduling policy to take into account fairness requirements.

[2] Professionals using the B method estimate that a seasoned practitioner averages sixteen interactive proofs per day.

Module	Size		Proof obligations			Interactive
	Operations	Lines	W.D.	Corr.	Total	proofs
Config	0	89	0	0	0	0
Types	0	103	1	1	2	1
Scheduler	5	90	0	0	0	0
Task	12	467	1	219	220	28
Queue	7	231	12	33	45	0
FreeRTOSBasic	19	562	37	46	83	2
FreeRTOS	19	562	43	3	49	0
Task_r	12	432	42	100	142	18
Total	55	1974	136	402	538	49

Table 1. The table presents, for each module, the number of operations defined in the module, the total number of lines (including comments), the number of proof obligations (well-definedness lemas, correctness theorems, and total), and the number of interactive proofs required to establish the correctness theorems. Most of our interactive proofs have fewer than 10 steps. In the one case (lowering the priority of the running task below that of at least one ready task), we needed more than a hundred steps. We do not claim that we were able to find the shortest proofs.

A relevant question in our context is the cost-effectiveness of the approach we have taken. For circumstantial reasons, this is a difficult question to answer. Indeed, the model was mainly developed by a student with little previous experience with formal methods, and even less with guiding an interactive prover. In retrospect, assuming that the development would be carried out by a professional with proficiency with the B method and its tool support (including the interactive prover), we estimate that the model could have been developed in a few weeks time. Also we are not sure that the modular approach we have taken is indeed the most suitable, compared to introduce system features such as queues incrementally, through horizontal refinements. It would certainly be useful for formal methods practicioners to have a published body of architectural patterns for large specifications.

Also, we feel that there is some space for improvement in the tool support. In the case of interactive proofs, hypothesis selection is often required, however the selection interface is a bit clumsy. Proof management is rudimentary and still has bugs: at times proofs are lost, at times the prover gets into an infinite loop and the whole interface needs to be restarted. Also the development environment was not designed for multi-user efforts and we have not found a satisfactory way to integrate Atelier B with a version control system. Since the graphical interface of Atelier B has recently gone open source, we hope that such improvements will soon be implemented by the community.

A second important question is: what is this model worth for? Several possible applications could be foreseen. The first would be to use it to verify existing

implementations of FreeRTOS[3], or to derive formally a new implementation of FreeRTOS. To verify an existing application, we could proceed either by reviewing code, taking as a reference the functional specification and try to manually find errors in the source code, or by deriving tests from the specification, using techniques such as [11]. Another approach to verification would be to use the B specification to instrument the source code of FreeRTOS with assertions, using a formalism such as ACSL [14], and formally prove that they are satisfied using low-level code verifiers such as VCC [15] or Frama-C [16]. A third possibility would be to use the model of FreeRTOS in formal development of real-time applications based on this system. It remains to be seen if this is possible to do this strictly within the scope of the B method, or if it would be necessary to couple it with other formalisms to handle e.g. concurrency and real-time properties.

The B method could also be applied to build an implementation of FreeRTOS from the model. However the B method currently has some restrictions that would make this task more difficult than a straightforward application of existing techniques. Indeed, the B method is targeted to safety-critical applications where dynamic memory allocation is prohibited. So current C code generators do not have support for pointers. However, such functionality is required in the case of FreeRTOS. It would be necessary to develop solutions to represent and manage memory representation in B.

Finally, since FreeRTOS is a library to build real-time embedded applications, the functional model presented in this paper could be used, in combination with a model checker for B such as ProB [17], as as an oracle when testing real-time applications based on FreeRTOS as proposed in [18].

Acknowledgements. We thank the anonymous reviewers for many insightful and challenging comments.

References

1. SBC: Grandes Desafios da Pesquisa em Computação no Brasil: 2006–2016. `http://www.sbc.org.br` (2006)
2. Jones, C., O'Hearn, P., Woodcock, J.: Verified software: a grand challenge. Computer **39**(4) (April 2006) 93–95
3. Woodcock, J.: Grand challenge in software verification. In: Brazilian Symposium on Formal Methods (SBMF 2008). (2008)
4. Craig, I.D.: Formal Models of Operating System Kernels. Springer (2007)
5. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996)
6. Schneider, S.: The B-Method: An Introduction. Palgrave (2001)
7. Jones, C.B.: Systematic Software Development Using VDM. Prentice Hall (1990)
8. Spivey, J.: The Z Notation: a Reference Manual. 2nd edn. Prentice-Hall International Series in Computer Science. Prentice Hall (1992)

[3] It is important to have in mind that part of the implementation is written in assembly, thus needing to be rewritten and re-verified for each target platform.

9. Requet, A.: Bart: A tool for automatic refinement. In: Abstract State Machines, B and Z. Volume 5238 of LNCS. (2008) 345–345
10. Abrial, J.R.: Faultless system: Yes we can! Technical Report 629, Department of Computer Science, ETH Zurich (2009)
11. Jaffuel, E., Legeard, B.: LEIRIOS test generator: Automated test generation from B models. In: The 7th International B Conference. (2007) 277–280
12. Clearsy: Atelier B 4.0 (2009) `http://www.atelierb.eu`.
13. Bicarregui, J., Hoare, C., Woodcock, J.: The verified software repository: a step towards the verifying compiler. Formal Aspects of Computing **18**(2) (2006) 143–151
14. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language (2008)
15. Dahlweid, M., Moskal, M., Santen, T., Tobies, S., Schulte, W.: Vcc: Contract-based modular verification of concurrent c. In: ICSE Companion, IEEE (2009) 429–430
16. CEA: Frama-c: Software analyzers (2009) `http://frama-c.cea.fr`.
17. Leuschel, M., Butler, M.: ProB: A model checker for B. In Araki, K., Gnesi, S., Mandrioli, D., eds.: FME 2003: Formal Methods. LNCS 2805, Springer-Verlag (2003) 855–874
18. Andrade, W.L., Alves, E.L.G., Almeida, D.R., , Machado, P.D.L.: Test case generation of embedded real-time systems with interruptions for FreeRTOS. In: Brazilian Symposium on Formal Methods (SBMF 2009). (2009)