DOI 10.1007/s00165-014-0308-9
BCS © 2014
Formal Aspects of Computing (2015) 27: 167–192

Formal Aspects of Computing

Using formal reasoning on a model of tasks for FreeRTOS

Shu Cheng[1], Jim Woodcock,[1] and Deepak D'Souza[2]

[1] Department of Computer Science, University of York, Deramore Lane, York, YO10 5GH, UK
[2] Dept of Computer Science and Automation, Indian Institute of Science, Bangalore, 560 012, India

Abstract. FreeRTOS is an open-source real-time microkernel that has a wide community of users. We present the formal specification of the behaviour of the task part of FreeRTOS that deals with the creation, management, and scheduling of tasks using priority-based preemption. Our model is written in the Z notation, and we verify its consistency using the Z/Eves theorem prover. This includes a precise statement of the preconditions for all API commands. This task model forms the basis for three dimensions of further work: (a) the modelling of the rest of the behaviour of queues, time, mutex, and interrupts in FreeRTOS; (b) refinement of the models to code to produce a verified implementation; and (c) extension of the behaviour of FreeRTOS to multi-core architectures. We propose all three dimensions as benchmark challenge problems for Hoare's Verified Software Initiative.

Keywords: Verified software initiative, FreeRTOS, formal verification, Z/Eves.

1. Introduction

1.1. FreeRTOS

FreeRTOS is a widely used real-time operating system written by a team led by Richard Barry of Wittenstein High-Integrity Systems in the UK [Bar12a]. Introductions to FreeRTOS informally describe the application programmers' interface (API) for the real-time operating system kernel [Bar12b]. Verifying the correctness of FreeRTOS has been proposed as a pilot project for the international Verified Software Initiative (VSI) [Hoa03, JOW06, Woo06, HMLS09], led by Tony Hoare. This verification experiment presents two distinct challenges: (a) Code-level verification to automatically analyse FreeRTOS for structural integrity properties. (b) The creation of a rational reconstruction of the refinement of the FreeRTOS code starting from an abstract specification, discharging all verification conditions automatically. Note, refinement here indicates that the refined version of a specification mathematically implies its abstract specification; thus, every property of the abstract specification is preserved by refinement. The project was chosen as a contribution to the VSI at a workshop held at Microsoft Research Cambridge in 2008 that was gathering difficult research problems from industry. Modelling and verifying operating system kernels is considered to be scientifically interesting, pushing current capabilities of software verification research and technology. Klein is the first to have formally verified an operating system kernel, and describes the main scientific challenges [Kle09, KEH+09].

Correspondence and offprint requests to: S. Cheng, E-mail: sc677@york.ac.uk

FreeRTOS has a large community of users programming embedded microcontrollers: it was downloaded more than 100,000 times during 2013, putting it high in the top 100 SourceForge codes (there are more than 200,000 available). Verification of FreeRTOS, entailing discovering residual errors, would thus have a strong impact on the international embedded systems community. FreeRTOS has relatively sophisticated functionality, but only a very small code base, all of which is open source and well documented (albeit informally).

FreeRTOS is a lightweight, embeddable, multi-tasking, Real-Time Operating System (RTOS). It makes the key assumption that the target system has a single processing unit. It is really a library of types and functions that can be used to build microkernels using a combination of C and assembly language, and has been ported to most embedded systems architectures. It allows a very small kernel to be produced for target microcontrollers, somewhere between 4–9kB. It provides services for embedded programming tasks, communication and synchronisation, memory management, real-time events, and I/O-device control.

Fourteen different compilers are used with FreeRTOS, giving complex configuration options and extensive parametrisation. A version of the software, SafeRTOS, has been certified to Safety Integrity Level 3 by the Technical University of Vienna for the following safety standards: IEC 61508, FDA 510(k), and DO-178B. These certificates are for the process of development, rather than for the correctness of the software against stated requirements.

The objective of formally verifying FreeRTOS would be to find some errors and make some guarantees about the code's behaviour. Since the requirements are distributed throughout the documentation, there is a clear need to produce a formal abstract specification. A broader aim of our work is to study the verification problem for an entire class of software, namely *real-time operating systems for embedded applications*, and we have chosen to focus on an exemplar of this class of systems, namely the FreeRTOS kernel. The techniques and methodology developed here can be expected to be applicable to other software in this class of systems.

Specifically, our aim is to carry out a systematic exercise towards the verification of FreeRTOS that will: (a) Produce a formal specification of its intended behaviour. (b) Identify aspects of its implementation that do not conform to this specification. (c) Produce a detailed model of the core scheduling-related functionality that can serve as a basis for fixing the current implementation to obtain a "verified" version of FreeRTOS, engineered as originally intended by the developers.

The main scientific difficulty with the verification of FreeRTOS is the low level of the code. The usual abstractions that make it easier to program systems software do not exist; it is the purpose of FreeRTOS to provide them. They include: (a) communication and synchronisation; (b) scheduling guarantees; (c) interference freedom; and (d) direct hardware interaction using clocks and interrupts. These are provided through many complex pointer-based operations, which present yet another challenge: verifying pointer programs is a complex and difficult business.

We specify the behaviour of FreeRTOS in the Z notation [WD96]; our specification is organised into five parts—(a) task, (b) queue, (c) time, (d) mutex, and (e) interrupt. In this paper, we introduce the first part of our work—the *Task* model. Sect. 1.2 summarises related work on formalising operating system kernels. We illustrate the contribution of this paper in Sect. 1.3. Following this, a general introduction to FreeRTOS is given in Sect. 2. Section 3 contains our formalisation in detail. Finally, in Sect. 5, we present our conclusions and make some suggestions for future work.

1.2. Related work

There is existing work on verifying operating systems:

1. Craig describes the specification and refinement in the Z notation of a microkernel, Labrosse's μC/OS operating system, which is similar to FreeRTOS [Cra06, Cra07, Lab02]. The refinement of the requirements targets mathematical data types at a level of abstraction well above program data types. The lowest-level of the refinement is also non-algorithmic, and there are no real-time properties. Freitas & Woodcock [FW09] have continued Craig's refinement to target datatypes at the level of FreeRTOS, but without a pointer implementation. Börger & Craig [BC09] extend this work by modelling with pseudo-code descriptions as Abstract State Machines (ASMs), which produce an elegant restructuring of the model that make it easier to understand and easier to refine to executable code.

2. Klein has verified seL4, a high-performance microkernel [Kle09, Kle10c, Kle10a, Kle10b, SWG+11]. An abstract specification in Isabelle/HOL is refined into an executable specification in Haskell, which is then manually refined into a high performance implementation in the C programming language. The theoretical basis for the work is in separation logic. There is an almost complete handling of the features of C. The entire exercise involved 8,700 lines of C, 200,000+ lines of proof script, and 30 man-years of effort to establish the functional correctness of the operating system.
3. Déharbe et al. have produced a specification in the B language of a restricted subset of FreeRTOS [DGM09]. They provide a formalisation of a subset of the API, including task and queue-related functions, verify that all its expressions are well-defined, and demonstrate logic consistency. The model contains seven basic B-machines, *FreeRTOSConfig*, *Type*, *Task*, *Queue*, *Scheduler*, *FreeRTOSBasic*, and *FreeRTOS*, with which the first model without priority is formalised. Then it is refined to the second model, which takes priority into account. However, there are some problems with this model; for instance, it prevents task creation while the scheduler is running, which is allowed by FreeRTOS; it forbids tasks sending and receiving messages to and from a queue when there is no task waiting to receive or send. Comparing this with our work, we introduce a model covering more functions of FreeRTOS, although in this paper we present only the Task part of the specification. Due to a finer structure of definitions and abstractions, our specification has an increased proof automation. Furthermore, we correct the problems discovered in Déharbe's work.
4. Pronk has studied the verification problem for FreeRTOS [Pro10]. He discusses and compares the advantages and disadvantages of theorem proving and refinement in this arena, compared with model checking. He concentrates on the latter, using Promela and the SPIN model checker. He abstracts from the pointer implementation.
5. Lin, Freitas, & Woodcock produced a specification of FreeRTOS in Z covering the top-level functionality [Lin10]. This was derived from Déharbe's B specification [DGM09] (see item 3), but then extended to capture all the main FreeRTOS functionality. An attempt was made at the verification of consistency using the Z/Eves theorem prover [MS97], although this could also be proved in ProofPowerZ [Art12], Isabelle/HOL [Pau12], or PVS [OSR12]. Originally, there were 30 unproved theorems out of 241. We have carried out further work to reduce their number of unproved theorems to around 10. During this process, we found the key reason for struggling with the proofs is that the model is very concrete, which leads to proof complexity. For example, to represent the different states in FreeRTOS, the model uses seven different type variables, such as: functions, sequences, finite sets, *etc*. Furthermore, even when Z/Eves can prove a theorem, it takes long time. Compared with this, our model is much more abstract and more tractable for proof.
6. Mühlberg & Freitas report on the application of the SOCA and VeriFast tools to FreeRTOS [MF11]. They focus on the verification of structural properties (e.g., pointer safety and arithmetic overflow) and liveness properties, but ultimately aim at demonstrating functional correctness. This includes the reconstruction of a formal specification of FreeRTOS in Z (actually the one mentioned in item 5), bounded model-checking of the FreeRTOS code using the SOCAVerifier [ML10], as well as annotating the source code with assertions in separation logic to apply the VeriFast software verifier [BJ10].
7. Ferreira has used separation logic to verify code-level pointer structures in FreeRTOS [FHQ12].
8. Abrial has an unpublished specification of much of the functionality of FreeRTOS using the B method (it excludes interrupts). The Z specification in this paper is based on his work, although the verification is necessarily very different.
9. Mistry, Naylor, & Woodcock have developed a multi-core version of FreeRTOS on a Field-Programmable Gate Array (FPGA), which is able to schedule tasks on multiple processors and support mutex in concurrent environment [Mis11, MNW13]. They present an adapted version of FreeRTOS that is able to schedule tasks on multiple processors, as well as provide full mutual exclusion support for use in concurrent applications that is independent of the chosen platform, thus preserving one of FreeRTOS's most attractive features: portability.
10. In a collaboration with the authors of the current paper, Kushwah, Divakaran, & D'Souza aim to give a proof of functional correctness by proving that the C implementation refines the abstract Z specification. The commonality with this work is that they also focus on the task-related functionality of FreeRTOS, although

their specification is deterministic, more detailed, and closer to the implementation than ours. They do not carry out a consistency check or prove properties about their Z model.

1.3. Contribution of this paper

As can be seen from Sect.1.2, there are a number of researchers working on the verification of FreeRTOS as part of the Verified Software Initiative, and this kind of community effort is an important aspect of the VSI. This paper contains a complete specification in Z of FreeRTOS's task model, together with proofs of consistency (well-definedness, initialisation, preconditions, and a few properties). The abstract characterisation of the tasking model is a first step towards a verified implementation of FreeRTOS on multicore. We were the first to promote FreeRTOS as a pilot project in the VSI, and the work presented continues this by establishing a benchmark for others to follow. We believe that this is an important contribution to both the verification community and also the embedded systems community.

2. Introduction to FreeRTOS

2.1. Overview of FreeRTOS

The key elements of FreeRTOS are:

Tasks: user processes.
Queues: communication mechanisms between tasks and interrupts.
Semaphores and Mutexes: used for resource management, event counting, mutual exclusion locks, *etc*.

They are implemented as a set of functions written in C. It provides the following functionality to the application programmer.

1. Implement fixed-priority, preemptive scheduling.
2. Trap software interrupts: (a) Find the highest priority ready task to run. (b) Save the context of the yielding task. (c) Restore the context of the new task.
3. Trap timer interrupts: (a) Update the tickcount. (b) Check delayed tasks, and move to ready if required, (c) Do context switching if required.
4. Provide API functions for: (a) Creation and management of multiple tasks. (b) Inter-task communication through queues, semaphores, and mutexes. (c) Heap memory management through `malloc` and `free`.

In this paper, we focus on the first part of FreeRTOS, *Tasks*.

2.2. Tasks of FreeRTOS

Tasks in FreeRTOS can be regarded as occupying one of two top-level states, `running` or `notRunning`. The running task is recorded by the task control block handler `pxCurrentTCB`, and simply indicates that the task is currently executing on the processor. The `notRunning` state can be decomposed into three substates: `ready`, `suspended`, and `blocked`; tasks in the `ready` state are available for scheduling to the `running` state. Tasks can also be blocked by an event for a certain period; these are the `blocked` tasks. `suspended` tasks wait until they are resumed by another task. Tasks transit between these states as described in Fig. 1. For instance, a task cannot transit from `suspended` to `running`, because only `ready` tasks can be scheduled as `running` [Bar12b].

In FreeRTOS, the scheduler takes responsibility for counting clock ticks, used to express time, and schedules tasks. The scheduling policy adopted is priority-based scheduling, which means that the task with the highest priority and in the ready state can be executed. As a result, it is impossible to use FreeRTOS in hard real-time environments. When a ready task has a higher priority than the running task, it will displace the running task from the CPU. The scheduler has two ways for switching tasks: preemptive and cooperative scheduling. In preemptive mode, the task with the highest priority will block the running task immediately and take the CPU. In cooperative mode, the running task can finish its CPU time before the task with the highest priority takes over. API functions are provided for task creation, deletion, and control. It is worth noting that the deletion API does not actually delete a task from the system: it only marks the task and removes its reference from related queues.

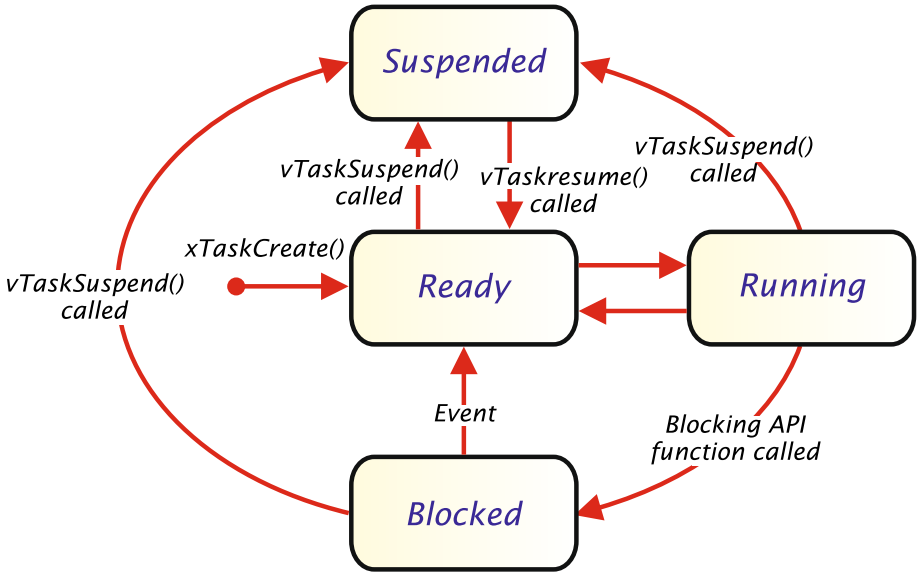

Fig. 1. State chart for Tasks

The *idle* task, with permanent priority 0, the lowest priority, is used to actually do the deleting job and release the memory allocated by the kernel; however, it does not collect the memory allocated by user, so tasks have to release used memory by themselves before deleting.

In our specification, the function *state* in schema *StateData* is used to specify the states of tasks. Further, the reverse function of *state* can be used to calculate tasks in a specific state; for instance, tasks in `ready` state are represented by $state^{\sim}(\!|\ \{ready\}\ |\!)$. This also works for running tasks: the result of $state^{\sim}(\!|\ \{running\}\ |\!)$ is a set with only one element—*running_task*, which represents the handler `pxCurrentTCB`. The function *priority* in schema *PrioData* represents tasks' priorities. They are defined in Sect. 3.1.

2.3. A simple example of FreeRTOS

We use a simple example application to illustrate the functionality provided by FreeRTOS. Fig. 2 shows the C code of an application that uses the FreeRTOS. Initially, the application creates two tasks: `Task1` and `Task2`, with priority 1 and 2 respectively (a higher number indicates higher priority), and then starts the FreeRTOS scheduler. The scheduler then runs `Task2`, which immediately increases the priority of `Task1` to 3. `Task2` is now preempted by `Task1`, which gets to execute and creates a new task—`Task3` with priority 4, which is the highest at the moment. Therefore, it preempts `Task1` and gets to execute. Once `Task3` is executing, it deletes itself, which triggers the scheduler to reschedule the system. As `Task1` has the highest priority at this moment, it gets to execute again and forever.

We now describe in more detail what happens in the FreeRTOS implementation code. The application code for `main`, `tx1`, `tx2` and `tx3` is compiled along with the FreeRTOS code (for the scheduler, and the API calls including `xTaskCreate`), and loaded into memory. The scheduler code is loaded into the Interrupt Service Routine (ISR) code area so that it services software interrupts.

By analysing the source code of FreeRTOS, we see that execution begins with the first instruction in `main`, which is the call to the `xTaskCreate` API function. This code is provided by FreeRTOS, allocates 1 kilobyte of memory on the heap for the task stack, as well as space to store its Task Control Block (TCB) [Bar12a, Bar12b]. From the source code, we can find that the TCB contains all vital information about the task: where its code (`tx1` in this case) is located, where its stack begins, where its current top-of-stack pointer is, what its priority is, and so on. The API call initialises the TCB entries for `Task1`. It then creates and initialises the various lists that the OS maintains, such as `pxReadyTasksLists` for recording tasks in `ready` state, `xSuspendedTaskList` for representing tasks in `suspended` state and so on. It finally adds `Task1` to the ready list and returns. Next, `main` calls `xTaskCreate` for `Task2` and the API call sets up the stack and TCB for `Task2` and adds it to the ready list, in a similar way. The next instruction in `main` is a call to the `vTaskStartSchedular` API, which is also provided by FreeRTOS. This call creates the idle task with priority 0, and adds it to the ready list.

```
xTaskHandle txh1;

void tx1(void * xPara){
    xTaskCreate(tx3, (signed char *) "Task 3", 1000, NULL, 4, NULL);
    for( ;; );
}

void tx2(void * xPara){
    for( ;; ){
        vTaskPrioritySet(txh1, 3);
    }
}

void tx3(void * xPara){
    for( ;; ){
        vTaskDelete(NULL);
    }
}

int main(void){
    xTaskCreate(tx1, (signed char *) "Task 1", 1000, NULL, 1, & txh1);
    xTaskCreate(tx2, (signed char *) "Task 2", 1000, NULL, 2, NULL);

    vTaskStartScheduler();
    return 0;
}
```

Fig. 2. An example application that uses RTOS.

It also sets the timer tick interrupt to occur at the required frequency. Finally, it does a context-switch to the highest priority ready task (i.e., it restores its execution state, namely the contents of its registers, from the task's stack where they were stored). The processor will next execute the instruction in the task that is resumed. In our example, this means that `Task2` will now begin execution.

When `Task2` begins execution, it makes an API call to `vTaskPrioritySet`. The code for this API call compares the new priority and the current priority to decide whether scheduling is needed. If the API increases the priority of a task or decreases the priority of the current running task, a reschedule will be requested. It then assigns the new priority to the target task, and moves the task to the proper position in the ready list, if it is a ready task. In our case, the priority of `Task1` is changed to 3, it is moved to its proper position in `pxReadyTasksLists`. The API code then does a `yield` (a software interrupt) that is trapped by the scheduler. The scheduler picks the longest waiting, highest priority ready task, which in this case is `Task1`, and makes it the running task. Before this, the scheduler saves the registers of `Task2` on its stack, and restores the register context of `Task1` from its stack.

`Task1` now creates the new task `Task3`. The process is similar to the `xTaskCreate` call to create `Task1` and `Task2`. The difference is that this time `xTaskCreate` triggers scheduling to make `Task3` running.

When `Task3` begins execution it makes a call to the `vTaskDelete` API call. The code for this API is simple. It removes the target task from state list and related events list, in this case, `Task3` is removed from `pxReadyTasksLists`. As it is the current running task, the API code triggers scheduling again to makes the highest priority ready task running, which is `Task1`. `Task1` then executes its trivial `for`-loop, *ad infinitum*.

The animation and formal verification of our specification for this process will be illustrated in Sect. 4.

3. Formalising FreeRTOS tasks in Z

Normally, the development process in formal methods starts from requirements, modelling the behaviour of the system, refining several steps to executable code. Because FreeRTOS does not have an explicitly articulated requirement and it has been implemented in C code, we consider the user manual and the practical guide of

FreeRTOS [Bar12a, Bar12b] as the basis of the requirements. However, these sources are not detailed enough for us to build the model, they just provide the basic functional description of the APIs. In this case, we also take the source code of the FreeRTOS into consider for modelling. Therefore, this model is mainly based on the API documentation to verify the functional correctness of FreeRTOS. Some of the details of the specification are derived from the source code. For instance, the documentation of the `xTaskCreate` API function mentions that if the API returns *pdTRUE*, the task has been created. (Note, to simplify the model, we just consider the success case of APIs.) Nevertheless, it does not indicate how it was created. Thus, we analyse the source code to find out how it works and formalise the behaviour of `xTaskCreate` based on that.

In this paper, we formalise the behaviour of FreeRTOS using the Z notation; those unfamiliar with Z may consult the *Z Reference Manual* [Spi92] or a tutorial introduction, such as [WD96]. In addition, as we introduce our model we also summarise the related Z notation we use. We use the Z/Eves theorem prover to reason about the Z specification; see [MS97, Saa99b, Saa99a]. We give a summary of the proof commands in Appendix A, so that the reader can follow the general argument behind the formal proofs or even recreate the proof in Z/Eves.[1]

3.1. Basic statements

We introduce only the first part of our model: the formal specification of *Task*. First of all, it is essential to state some basic context that will be used in the specification.

The given sets *CONTEXT* and *TASK* are provided as given sets to represent the environment of the processor and the tasks, respectively; in Z, given sets are basic, maximal types.

$$[CONTEXT, TASK]$$

Two constants, *bare_context* and *idle*, are introduced by an axiomatic definition, which contains a declaration and a constraint. Here, the constraint is trivially true and is omitted. The constant *bare_context* is an element of the set *CONTEXT*; it represents the initial state of the processor. The constant *idle* is of type *TASK* it represents the system task that runs when no other task is scheduled.

$$\begin{array}{|l} bare\_context : CONTEXT \\ idle : TASK \end{array}$$

STATE is defined using a free type in its simplest form as enumerating exactly five distinct constants.

$$STATE ::= nonexistent \mid ready \mid blocked \mid suspended \mid running$$

The set of legal state transition is described by an abbreviation: *transition* names the appropriate set that models the diagram in Fig. 1.

$$\begin{aligned} transition == & (\{blocked\} \times \{nonexistent, ready, running, suspended\}) \\ & \cup \ (\{nonexistent\} \times \{ready, running\}) \\ & \cup \ (\{ready\} \times \{nonexistent, running, suspended\}) \\ & \cup \ (\{running\} \times \{blocked, nonexistent, ready, suspended\}) \\ & \cup \ (\{suspended\} \times \{nonexistent, ready, running\}) \end{aligned}$$

Transitions (*blocked*, *running*) and (*suspended*, *running*) are included because when a task is woken up from the `blocked` state or resumed from the `suspended` state, its state actually transits to `ready`; however, if it has a higher priority than the running task, it will be scheduled to `running`. At this level of abstraction, we consider these two steps as a single step, which makes state transition (*blocked*, *running*) and (*suspended*, *running*) possible. This definition turns out not to be very useful in automating proofs about transitions, because Z/Eves would expand *transition* into the set in all possible proof contexts. This increases the load on the prover massively. So we `disable` the definition and add two theorems that are more helpful. The first is a typing lemma that states that *transition* is a set of pairs of *STATE*; its proof is a very simple consequence of the definition of *transition*.

[1] Z/Eves project file and other related files can find on the web page: https://code.google.com/p/z-spec-freertos/source/browse/.

Theorem 1 (gTransitionType)

$$transition \in \mathbb{P}(STATE \times STATE)$$

proof [*gTransitionType*]
with enabled (*transition*) *prove by reduce*;
■

Next, we add the following lemma to tell Z/Eves about each individual pair in *transition*, which is helpful to Z/Eves for automatically proving; again, the proof is very simple.

Theorem 2 (rule lInTransition)

$$\begin{aligned}&\forall\, l, r : STATE \\ &\quad \mid (l, r) \in \{(nonexistent \mapsto ready), (running \mapsto ready), \\ &\qquad (blocked \mapsto ready), (suspended \mapsto ready), \\ &\qquad (ready \mapsto running), (blocked \mapsto running), \\ &\qquad (suspended \mapsto running), (nonexistent \mapsto running), \\ &\qquad (running \mapsto suspended), (ready \mapsto suspended), \\ &\qquad (blocked \mapsto suspended), (running \mapsto blocked), \\ &\qquad (running \mapsto nonexistent), (ready \mapsto nonexistent), \\ &\qquad (blocked \mapsto nonexistent), (suspended \mapsto nonexistent)\} \\ &\quad \bullet\ (l, r) \in transition\end{aligned}$$

proof [*lInTransition*]
with normalization with enabled (*transition*) *prove by reduce*;
■

Based on these definitions, the state schema of the model can be specified, describing basic system properties. Due to the scope of this paper, we focus only on task-related information in FreeRTOS. To simplify the proof and the specification, we verify system only when the scheduler is running; therefore, we assume the scheduler is always running.

To describe the tasks in FreeRTOS, the following four kinds of data are needed, which are defined by schema definition. In the Z notation, the schema is used to structure and compose descriptions. Once a schema is assigned a name, it is possible to use that name to reuse the schema in other expressions or schemas.

1. **Task data.** The variables recorded in this category are directly related to tasks. First, to simplify the description of the model and the following proofs, we need to distinguish tasks that are known to the system from others; therefore, a set *tasks* is defined as a finite subset of *TASK*. Second, according to the source code of FreeRTOS, *task.c* file, a pointer (`pxCurrentTCB`) is used to record the current running task, which is useful in several cases, such as scheduling. In the specification, a variable *running_task* in the type *TASK* is used to represent this. Two constraints are specified: the *idle* task and the *running_task* have to be known to the system at all times.

TaskData

$$\begin{aligned}&tasks : \mathbb{P}\, TASK \\ &running\_task : TASK \\ &\hline &running\_task \in tasks \\ &idle \in tasks\end{aligned}$$

2. **State data.** As described in Sect. 2.2, FreeRTOS uses different lists to manage the tasks known to the system. Abstractly, two tasks in different lists have different states. Therefore, the variable *state* is used to indicate the state of the tasks. Specifically, the *idle* task, which is a system task and responds to do some maintenance job for the system (such as garbage collection), can only be *ready* or *running*; it cannot be *blocked*, *suspended*, or deleted (*nonexistent*).

StateData

$$\begin{aligned}&state : TASK \to STATE \\ &\hline &state(idle) \in \{ready, running\}\end{aligned}$$

3. **Context data.** The two variables *phys_context* and *log_context*, respectively, represent the physical context of the system and the logical context for all the tasks that are not running.

$$
\begin{array}{|l}
\textit{ContextData} \\
\hline
\textit{phys\_context} : \textit{CONTEXT} \\
\textit{log\_context} : \textit{TASK} \rightarrow \textit{CONTEXT} \\
\hline
\end{array}
$$

4. **Priority data.** FreeRTOS is a priority-based operating system: all the tasks in the system have their own priority, and a total function, *priority*, is introduced to record this. The priority of *idle* task must be the lowest priority all the time, which is 0.

$$
\begin{array}{|l}
\textit{PrioData} \\
\hline
\textit{priority} : \textit{TASK} \rightarrow \mathbb{N} \\
\hline
\textit{priority}(\textit{idle}) = 0 \\
\hline
\end{array}
$$

Invariant Based on these definitions, we can now describe the state data for tasks that is maintained by this part of FreeRTOS.

$$
\begin{array}{|l}
\textit{Task} \\
\hline
\textit{TaskData} \\
\textit{StateData} \\
\textit{ContextData} \\
\textit{PrioData} \\
\hline
\textit{tasks} = \textit{TASK} \setminus (\textit{state}^{\sim}(\!| \{\textit{nonexistent}\} |\!)) \\
\textit{state}^{\sim}(\!| \{\textit{running}\} |\!) = \{\textit{running\_task}\} \\
\forall\, \textit{pt} : \textit{state}^{\sim}(\!| \{\textit{ready}\} |\!) \bullet \textit{priority}(\textit{running\_task}) \geq \textit{priority}(\textit{pt}) \\
\hline
\end{array}
$$

Apart from the four schemas describing the task, state, context, and priority data, there are three more constraints added in this schema. They show that

- All the tasks whose state is not *nonexistent* are known to the system. Here, as mentioned above, the *state* is a function, a special case of a relation. The operator, $^{\sim}$, takes the inverse relation, so that $\textit{state}^{\sim}$ is a relation in $\textit{STATE} \leftrightarrow \textit{TASK}$. The operand, $(\!|$ and $|\!)$ calculates relational image. The result for this predicate is a set that contains all the *TASK*s whose states are *nonexistent*.
- There is only one task that can occupy the *running* state at any given time.
- The priority of *running_task* is the greatest one amongst all the ready tasks.

Initialization Based on the state definition and the assumptions mentioned above, we describe the initialisation of the *Task* state in a similar piecewise fashion: we separately initialise the four sub-states, and then combine them.

1. **Task data.** Initially, there are no user-defined tasks in the system; there is only one task in the system: *idle*. It is also the initial *running_task*.

$$
\begin{array}{|l}
\textit{Init\_TaskData} \\
\hline
\textit{TaskData}' \\
\hline
\textit{tasks}' = \{\textit{idle}\} \\
\textit{running\_task}' = \textit{idle} \\
\hline
\end{array}
$$

2. **State data.** Furthermore, every other task is in the *nonexistent* state, except *idle* whose state is *running*.

$$
\begin{array}{|l}
\textit{Init\_StateData} \\
\hline
\textit{StateData}' \\
\hline
\textit{state}' = (\lambda\, x : \textit{TASK} \bullet \textit{nonexistent}) \oplus \{(\textit{idle} \mapsto \textit{running})\} \\
\hline
\end{array}
$$

3. **Context data.** Also, initially, the logical and physical contexts of all tasks are the *bare_context*.

$$
\begin{array}{|l}
\textit{Init\_ContextData} \\
\hline
\textit{ContextData}' \\
\hline
\textit{phys\_context}' = \textit{bare\_context} \\
\textit{log\_context}' = (\lambda\, x : \textit{TASK} \bullet \textit{bare\_context}) \\
\hline
\end{array}
$$

4. **Priority data.** Finally, all tasks have the lowest priority, 0.

$$
\begin{array}{|l}
\textit{Init\_PrioData} \\
\hline
\textit{PrioData}' \\
\hline
\textit{priority}' = (\lambda\, x : \textit{TASK} \bullet 0) \\
\hline
\end{array}
$$

The initial state for *Task* can be defined using these four definitions.

$$
\begin{array}{|l}
\textit{Init\_Task} \\
\hline
\textit{Task}' \\
\hline
\textit{Init\_TaskData} \\
\textit{Init\_StateData} \\
\textit{Init\_ContextData} \\
\textit{Init\_PrioData} \\
\hline
\end{array}
$$

In order to prove that all these initial states are reachable, the following five theorems are introduced. They assert that there is at least one possible postcondition for initialising each sub-state schema and the overall schema. Due to the simplicity of these theorems, Z/Eves is able to prove them fully automatically.

Theorem 3 (TaskDataInit)

$$\exists\, \textit{TaskData}' \bullet \textit{Init\_TaskData}$$

proof [*TaskDataInit*]
prove by reduce;
■

Theorem 4 (StateDataInit)

$$\exists\, \textit{StateData}' \bullet \textit{Init\_StateData}$$

Theorem 5 (ContextDataInit)

$$\exists\, \textit{ContextData}' \bullet \textit{Init\_ContextData}$$

Theorem 6 (PrioDataInit)

$$\exists\, \textit{PrioData}' \bullet \textit{Init\_PrioData}$$

It is easy to prove these theorems with the proof command "*prove by reduce*", except for *TaskInit*, because it has more constraints on its state variables.

Theorem 7 (TaskInit)

$$\exists\, \textit{Task}' \bullet \textit{Init\_Task}$$

proof [*TaskInit*]
prove by reduce;
apply extensionality;
with enabled (*applyOverride*) *prove*;
■

After the automatic proving ordered by *prove by reduce*, Z/Eves is confused about the equivalence between sets defined in schema *Task*. The application of override also confuses the prover. Therefore, we need to guide the prover to apply theorems, *extensionality* and *applyOverride*, to discharge them. These theorems are provided by the Z/Eves toolkit.

We can check whether the state change respects the transition relation as a dynamic invariant that must be satisfied by all the operations on the *Task* state by redefining $\Delta Task$:

$$\Delta Task$$
$$Task$$
$$Task'$$
$$\forall st : TASK \mid state'(st) \neq state(st) \bullet state(st) \mapsto state'(st) \in transition$$

It is worth mentioning that in this schema we use $Task'$ to refer the post state of the *Task*. Initially, the expression "$\Delta Schema$" (*Schema* refers to a state schema) has been defined to contain both the pre and post state of *Schema*. We redefine it here to add further constraints for *Task*.

Based on these fundamental definitions, operations related to tasks can be specified.

3.2. Additional schema for reschedule

In a multi-tasking real-time operating system, rescheduling tasks is essential and occurs frequently. Generally, depending on the purpose of the system, the operating system would follow some suitable algorithm to determine the task to be scheduled. Afterwards, other system states can be updated accordingly. Therefore, at this level of abstract specification, it is possible to define the rescheduling process nondeterministically. However, the model described in this paper focuses on FreeRTOS. We will follow the algorithm used in FreeRTOS to specify rescheduling, which is based on the priority of tasks. Specifically, once a `ready` task obtains higher priority than the running task, it will be scheduled as the new running task. Subsequently, the system will switch the context of the current running task out and swap in the context of a new running task. It is also necessary to manage related lists and system states properly; for instance, by setting the selected task as the running task and inserting the current running task in a suitable list. In this specification, we introduce the schema *Reschedule* to perform the swapping part of the rescheduling process, which can then be reused by other schemas. The priority-based scheduling algorithm is embedded in the operation schemas for different APIs that need rescheduling, being specialised by the destination to which the current running task is moved. For example, when suspending the running task, the destination of the running task is the suspended list; but when we create a task with a higher priority than the running task, the destination of the current running task is actually the ready list. These lists are represented by the function *state*, so updating the *state* with the variable *st*? is the way to manage these lists. In the Z notation, the variables marked with "?" and "!" indicate that they are I/O variables, respectively, for a schema. When other schemas reuse the *Reschedule* schema, *st*? will be introduced within these schemas with the value of the destination of the current running task. Because both schemas contain the variable with the same name, these two variables will be bound together. Consequently, the schema *Reschedule* can get the destination of the running task by accessing the value of *st*?. The operator, $\oplus$, is normally used to update functions in the Z notation. If the first element of a pair exists in the domain of the function, it would update the second element of the pair in the function to the new value; otherwise it appends the pairs to the function. Therefore, it is used here to update the *state* of *running_task* and *target*?. Similarly, for each case the new running task, the final state of *tasks*, and the priority of tasks may also be different. We leave these decisions to the calling schemas. Therefore, variables—*target*?, *tasks*? and *pri*?—are introduced to represent these properties respectively.

Reschedule

$\Delta Task$
$target? : TASK$
$tasks? : \mathbb{P}\, TASK$
$st? : STATE$
$pri? : TASK \rightarrow \mathbb{N}$

$tasks' = tasks?$
$running\_task' = target?$
$state' = state \oplus \{(target? \mapsto running), (running\_task \mapsto st?)\}$
$phys\_context' = log\_context(target?)$
$log\_context' = log\_context \oplus \{(running\_task \mapsto phys\_context)\}$
$priority' = pri?$

The calling schema just needs to specify the values for these variables properly, then *Reschedule* schema can handle the rest of work.

3.3. Creating and deleting tasks

After initialising the system, there is only one task (*idle*); in order to add more tasks into the system, the *Create* operation can be used. Once a task finishes its job, it should be *Delete*d to allow other tasks to use the resources held by it. They are also the first group of the API interface provided by FreeRTOS, `xTaskCreate` and `vTaskDelete`. Generally, there are two cases for each of these two operations: one is simply to add or remove a task from the system; the other one leads to a re-scheduling of tasks.

First Case of Creating Tasks If the assigned priority is not greater than the priority of the current running task, it simply adds the new task that does not already exist. The input *target*? represents the task that will be created. The input *newpri*? contains the priority assigned to the new task. Therefore, the precondition is specified as: first, *target*? is not known by the system; second, the assigned priority, *newpri*? is no more than the priority of *running_task*. After the operation, the *target*? is known by the system, the task *target*? is added to *tasks* and updates the *state* function to record that the state of *target*? is *ready*. The input *newpri*? is assigned to the task *target*? by updating the function *priority*. Because this operation will not cause rescheduling, besides these changes, other properties of *Task* remain unchanged. The "Ξ" operation has been used in here: it is defined in the Z notation to show that the pre and the post states are unchanged. The schema *CreateTaskN_T* can be introduced, which indicates that this schema is for *Create Task operation* in the *normal case* for *Task model*. Generally, we use postfix *N* to stand for *Normal case* of the operation, which does not lead to rescheduling; and *S* stands for *Scheduling case*. The postfix after the dash indicates which model it is specified for, like *T* in this case shows the schema is part of task model.

CreateTaskN_T

$\Delta Task$
$target? : TASK$
$newpri? : \mathbb{N}$

$state(target?) = nonexistent$
$newpri? \leq priority(running\_task)$
$tasks' = tasks \cup \{target?\}$
$running\_task' = running\_task$
$state' = state \oplus \{(target? \mapsto ready)\}$
$\Xi ContextData$
$priority' = priority \oplus \{(target? \mapsto newpri?)\}$

Having defined this operation as a relation on *Task* states, we need to work out what its precondition is. We posit that the before-state, the input, and the first two predicates are exactly the precondition, and collect these into the following schema, where the suffix *FSBSig* in the schema name stands for *Feasibility Signature*.

CreateTaskN_TFSBSig

$$
\begin{array}{l}
Task \\
target? : TASK \\
newpri? : \mathbb{N} \\
\hline
state(target?) = nonexistent \\
newpri? \leq priority(running\_task)
\end{array}
$$

These declarations and predicates are clearly necessary for the actual precondition as we stated above; we show that they are also sufficient in the next theorem, which can be automatically generated. Specifically, for any "state" that satisfies the definition of *CreateTaskN_TFSBSig*, the precondition of *CreateTaskN_T* is satisfied. The operator "pre" is defined in the Z notation to calculate the precondition schema of a schema [WD96, Chap. 14]. For instance, the predicate "pre *CreateTaskN_T*" in the following theorem obtains the precondition schema by calculating $\exists\, Task' \bullet CreateTaskN\_T \setminus (outputs)$, where *output* refers to the list of output variables related to the operation, which will be hidden, and is empty in this case. The schema hiding operator, \, hides the variables list in the *outputs* from the declaration of the operation by introducing them in predicate part of schema with existential quantifier.

Theorem 8 (CreateTaskN_T_vc_ref)

$$\forall\, CreateTaskN\_TFSBSig \mid true \bullet \text{pre}\ CreateTaskN\_T$$

It is interesting to understand the proof of this theorem. First of all, as we mentioned above, the Z/Eves prover is used to verify our specification. All the proof scripts shown in this paper are used for helping Z/Eves to finish the proving work. Generally, there are two ways to finish a proof [2]: (a) exploratory proof—directly prove the theorem without any previous plan and address any proof goals returned by prover; (b) planned proof—carry out a detailed plan for the proof, which is enough to finish the proof by hand, then transfer the plan to a proof script for the prover. To maximise the benefit of the proof automation, we adopt the exploratory proof approach in many cases. The general idea for this approach is:

1. Expand terms such as schema references, and let Z/Eves prove the proof goal automatically.
2. When Z/Eves is stuck, stop at some proof goals, guide Z/Eves by using or applying related theorems or lemmas to rewrite the proof goals, provide more conditions, *etc*.
3. Let Z/Eves progress based on the new goal.
4. Repeat step 2 and step 3 until the proof is finished.

For efficiency issues, it is necessary to expend as few terms as possible in step 1. This can significantly reduce the proof time, especially when the system is complex. This is also the reason for defining our system in parts.

Specifically, we first use the following proof command to expand all necessary terms and then let the prover automatically apply rules and theorems, which are included by Z/Eves, to prove the goal.

with disabled (*ContextData*) *prove by reduce*;

Meanwhile, because in this schema the *ContextData* is unchanged, we keep it unexpanded. The prefix *with disabled* (*ContextData*) can achieve this by making the prover ignore *ContextData*, when expending the terms. Note that, as some theorems are rarely used when proving and some other theorems are time consuming, Z/Eves disables them by default. This is helpful for improving the efficiency of the proof process; however, it is also one of the reasons why Z/Eves may be stuck in some cases. As a result, the original proof goal is transferred to the following four goals.[3]

[2] This idea is suggested by Leo Freitas.

[3] Because the proof goals are too long to present in this paper, we just list the most important part here. Please download the Z/Eves project file from the web page and open it with Z/Eves to find the details.

1. The *running_task* remains the same before and after the operation:

$$(state \oplus \{(target?, ready)\})^{\sim}(\!| \{running\} |\!) = state^{\sim}(\!| \{running\} |\!)$$

   As we defined in *CreateTaskN_T*, the expression $state \oplus \{(target?, ready)\}$ is equal the post *state*. Therefore the image of *running* under the inverse function $(state \oplus \{(target?, ready)\})^{\sim}$ represents the *running_task* after the operation.

2. The *target*? task is added into the system by the operation:

$$TASK \setminus ((state \oplus \{(target?, ready)\})^{\sim}(\!| \{nonexists\} |\!)) = \{target?\} \cup (TASK \setminus (state^{\sim}(\!| \{nonexistent\} |\!)))$$

   Similarly, the left side of equation indicates all the tasks known by the system after the operation. It should be the same as the known tasks of the pre state of the system plus the created task, which is *target*?.

3. The priority of the *target*? task is less than or equal to the running task:

$$((state \oplus \{(target?, ready)\})(pt) = ready \land (pt = target? \lor pt \in TASK) \Rightarrow priority(running\_task) \geq (priority \oplus \{(target?, newpri?)\})(pt))$$

   Comparable to *state*, the post state of the *priority* function can also be written as: $priority \oplus \{(target?, newpri?)\}$. In this case, this expression is easy to understand.

4. Every state transition made by any task respects the *transition* relation.

$$(st \in TASK \land \neg (state \oplus \{(target?, ready)\})(st) = state(st) \Rightarrow (state(st), (state \oplus \{(target?, ready)\})(st)) \in transition)$$

For the first two goals, the prover is confused by the equivalence relation between two sets. Therefore, the second and the third commands in the script

apply extensionality to predicate
$(state \oplus \{(target?, ready)\})^{\sim}(\!| \{running\} |\!) = state^{\sim}(\!| \{running\} |\!)$;
apply extensionality to predicate
$TASK \setminus ((state \oplus \{(target?, ready)\})^{\sim}(\!| \{nonexistent\} |\!)) = \{target?\} \cup (TASK \setminus (state^{\sim}(\!| \{nonexistent\} |\!)))$;

are used to discharge them.

After this, the proof goal 3 is given by the constraint in *Task* schema. Tasks other than *target*? maintain the requirement that the priority of running task is at least as great as that of all the ready tasks:

$$\forall pt : state^{\sim}\{ready\} \bullet priority(running\_task) \geq priority(pt)$$

A copy of this constraint is in the assumption part of the goal as well, and to distinguish *pt* in these two, Z/Eves renames one from *pt* to *pt__0*. Therefore, to prove that tasks other than *target*? obey the constraint, we just need to indicate that *pt__0* and *pt* are the same. For *target*?, the priority is defined as *newpri*?, which is specified to be no more than the priority of *running_task* as the precondition of this schema. The rule *applyOverride* is applied to analyse the expressions that contain the operator $\oplus$. Finally, the command *with normalization prove*; is used to finish the proof.[4] Thus, the theorem *CreateTaskN_T_vc_ref* can be proved by following script in Z/Eves.

[4] The details about the proof command, *with normalization prove*; , can be found in the appendix A.

proof [*CreateTaskN_T_vc_ref*]
with disabled (*ContextData*) *prove by reduce*;
apply extensionality to predicate
$(state \oplus \{(target?, ready)\})^{\sim}(\!|\{running\}|\!) = state^{\sim}(\!|\{running\}|\!)$;
apply extensionality to predicate
$TASK \setminus ((state \oplus \{(target?, ready)\})^{\sim}(\!|\{nonexistent\}|\!)) =$
$\{target?\} \cup (TASK \setminus (state^{\sim}(\!|\{nonexistent\}|\!)))$;
instantiate $pt\_\_0 == pt$;
with enabled (*applyOverride*) *prove*;
apply applyOverride;
with normalization reduce;
■

Second Case of Creating Tasks If the priority assigned to the new task is greater than the priority of the running task, then rescheduling is required; this is achieved by calling the *Reschedule* schema. The current running task will be moved into the ready state; the new priority and initial context is allocated for the new task, which is then scheduled to be running. To reuse *Reschedule*, the variables *st*?, *pri*? and *tasks*? are declared and assigned appropriately. Note, the default logical context for new tasks is *bare_context*, we do not need to set it separately. Therefore, the schema for the second case of the create task operation can be defined as follows:

CreateTaskS_T

$\Delta Task$
$target? : TASK$
$newpri? : \mathbb{N}$

$state(target?) = nonexistent$
$newpri? > priority(running\_task)$
$\exists st? : STATE;\ tasks? : \mathbb{P}\, TASK;\ pri? : TASK \to \mathbb{N}$
$\quad | st? = ready$
$\quad \wedge tasks? = tasks \cup \{target?\}$
$\quad \wedge pri? = priority \oplus \{(target? \mapsto newpri?)\} \bullet Reschedule$

Similarly to the previous case, the signature schema and the precondition theorem can be defined.

CreateTaskS_TFSBSig

$Task$
$target? : TASK$
$newpri? : \mathbb{N}$

$state(target?) = nonexistent$
$newpri? > priority(running\_task)$

Theorem 9 (CreateTaskS_T_vc_ref)

$\forall$ *CreateTaskS_TFSBSig* | *true* • pre *CreateTaskS_T*

This indicates that the new task is unknown by the system before the operation and the priority of new task is higher than the priority of the running task; this is sufficient and necessary for the precondition of schema *CreateTaskS_T*.

Deleting Tasks The first case for deleting a task is that it is not the running task: the state of this task—provided it is not the *idle* task—can be *ready*, *blocked*, or *suspended*. Because, normally the handle of *idle* task, `xIdleTaskHandle`, is private to the system and impossible for user to obtain, after the operation, the deleted task will become unknown to the system by deleting it from *tasks*, setting its state to *nonexistent*, and setting its logical context to the *bare_context*. It is worth mentioning that in the source code of `vTaskDelete` in FreeRTOS, the context of the deleted task is not actually deleted, but instead moved to the `xTasksWaitingTermination`

list. It is the `idle` task that actually performs garbage collection to recover the resources allocated by the system. At this level of abstraction, we consider all this as part of the deletion operation, resetting the *log_context* of the deleted task to the *bare_context*. Note, due to space limitations, we list parts of our model where there is something special; the rest of the specifications, precondition theorems, and proof scripts can be found on the web page.

Secondly, if the task to be deleted is the running task—but not the *idle* task—then we remove it from the system. This leaves a vacuum to be filled: we need to schedule another process to use the CPU. Of course, we will choose the task in a ready state with the highest priority. However, we cannot use *Reschedule* to achieve this because the logical context of the running task will be reset, which is requested by this operation but not supported by *Reschedule*. The output variable *topReady*! is introduced. The universally quantified expression specifies that the *topReady*! holds the highest priority. It is worth mentioning here that if there are several solutions, then *topReady*! is chosen nondeterministically. Similarly, the *tasks*, *state*, *phys_context* and *log_context* are updated.

DeleteTaskS_T

$$
\begin{array}{l}
\Delta Task \\
target? : TASK \\
topReady! : TASK \\
\hline
target? \in tasks \setminus \{idle\} \\
state(target?) \in \{running\} \\
state(topReady!) = ready \\
\forall\, t : state^{\sim}(\!|\, \{ready\} \,|\!) \bullet priority(topReady!) \geq priority(t) \\
tasks' = tasks \setminus \{target?\} \\
running\_task' = topReady! \\
state' = state \oplus \{(topReady! \mapsto running), (target? \mapsto nonexistent)\} \\
phys\_context' = log\_context(topReady!) \\
log\_context' = log\_context \oplus \{(target? \mapsto bare\_context)\} \\
\Xi PrioData
\end{array}
$$

The signature schema of this can be obtained as follows.

DeleteTaskS_TFSBSig

$$
\begin{array}{l}
Task \\
target? : TASK \\
\hline
target? \in tasks \setminus \{idle\} \\
state(target?) \in \{running\}
\end{array}
$$

Theorem 10 (DeleteTaskS_T_vc_ref)

$$\forall\, DeleteTaskS\_TFSBSig \mid true \bullet \text{pre}\, DeleteTaskS\_T$$

As mentioned above, the "pre" operator calculates the precondition schema for *DeleteTaskS_T*, which is the result of $\exists\, Task' \bullet DeleteTaskS\_T \setminus (topReady!)$. When the prover automatically discharges this predicate, it attempts to eliminate existentially quantified variables. Because the post state of the system, *Task*, has been defined in the operation, the one-point rule[5] is applied to handle them. However, for the variable *running_task'* and *topReady*! it can only eliminate one of them, because the output variable *topReady*! is assigned the value of *running_task'* in this operation. Therefore, the proof goal will become:

[5] One-point rule: $\exists x : X \mid p \bullet q \wedge x = t \Leftrightarrow p[t/x] \wedge q[t/x] \wedge t \in X$, provided that x is not free in t.

$$
\begin{aligned}
&\exists\, running\_task' : TASK \bullet \\
&\quad Task[log\_context := log\_context \oplus \{(target?, bare\_context)\}, \\
&\qquad phys\_context := log\_context(running\_task'),\ running\_task := running\_task', \\
&\qquad state := state \oplus (\{(target?, nonexistent)\} \cup \{(running\_task', running)\}), \\
&\qquad tasks := tasks \setminus \{target?\}] \\
&\quad \land\ (\forall\, st \in TASK \mid \\
&\qquad \neg\ (state \oplus (\{(target?, nonexistent)\} \cup \\
&\qquad\quad \{(running\_task', running)\}))(st) = state(st) \bullet \\
&\qquad (state(st), (state \oplus (\{(target?, nonexistent)\} \cup \\
&\qquad\quad \{(running\_task', running)\}))(st)) \in transition) \\
&\quad \land\ state(running\_task') = ready \\
&\quad \land\ (\forall\, t\_\_0 : state^{\sim}(\!|\, \{ready\} \,|\!) \bullet priority(running\_task') \geq priority(t\_\_0)
\end{aligned}
$$

Meanwhile, as defined in the specification, a highest priority ready task is nondeterministically assigned to the variable *running_task′*. In this case, with the existential-elimination rule, if we can find an instance of these tasks that satisfies this predicate, the proof goal can be verified. Therefore, we introduce the following function to discover a member of a set of tasks that has the highest priority among other tasks in that set.

$$
\begin{aligned}
&f : \mathbb{P}\, TASK \pfun TASK \\
&\hline
&\emptyset \notin \mathrm{dom}\, f \\
&\langle\!\langle\, \mathrm{topReadyTask} \,\rangle\!\rangle \\
&\forall\, Task;\ a : \mathbb{P}\, TASK \\
&\qquad \bullet\ f(a) \in a \land (\forall\, t : a \bullet priority(f(a)) \geq priority(t))
\end{aligned}
$$

Afterwards, it is possible to use this function to find out the highest priority task in ready state and use it to instantiate the *running_task′*. If we let $p, \exists\, x : X \bullet q$ represent the conditions and the goal of proof above, the predicate to be proved can be considered as

$$p \Rightarrow \exists\, x : X \bullet q. \tag{1}$$

Further, let t represents $f(state^{\sim}(\!|\, \{ready\} \,|\!))$. When we instantiate the *running_task′* with the delegate, with one-point rule, we have $\exists\, x : X \bullet q \land x = t \Leftrightarrow t \in X \land q[t/x]$, which gives $\exists\, x : X \bullet q \Leftrightarrow \exists\, x : X \bullet q \lor (t \in X \land q[t/x])$. Therefore, the equation (1) transfers into:

$$p \Rightarrow (\exists\, x : X \bullet q) \lor (t \in X \land q[t/x]). \tag{2}$$

Reorganising the equation, the relation

$$p \land \neg\ (t \in X \land q[t/x]) \Rightarrow \exists\, x : X \bullet q. \tag{3}$$

can be acquired.

Therefore, applying the proof command "*instantiate running_task′ == f(state~(| {ready} |))*; ", a negative copy of this proof goal will be added to the condition part, of which *running_task′* will be replaced by $f(state^{\sim}(\!|\, \{ready\} \,|\!))$. Analysing this negative copy of the goal, we will find

$$
\begin{aligned}
&Task[log\_context := log\_context \oplus \{(target?, bare\_context)\}, \\
&\quad phys\_context := log\_context(f(state^{\sim}(\!|\, \{ready\} \,|\!))), \\
&\qquad\qquad \ldots \quad \Rightarrow \\
&\qquad\qquad\quad t \in TASK \land \\
&\qquad\qquad\quad state(t) = ready \land \\
&\qquad\qquad\quad \neg\ priority(f(state^{\sim}(\!|\, \{ready\} \,|\!))) \geq priority(t)
\end{aligned}
\tag{4}
$$

which conflicts with the definition of function f; therefore, it is `not` *true*. However, according to the implication, if we can prove that the condition is *false*, the result of proof is *true*. Due to the complication for proving this condition is *false*, an assistant theorem, *lDeleteTaskS_T_Lemma*, is introduced. When we use it to prove *DeleteTaskS_T_vc_ref*, the variable *topReady*! can be substituted by $f(state^{\sim}(\!|\, \{ready\} \,|\!))$.

Theorem 11 (lDeleteTaskS_T_Lemma)

$$
\begin{aligned}
&\forall\, Task;\ topReady!, target? : TASK \\
&\quad \mid target? \in tasks \setminus \{idle\} \\
&\quad \land\ state(target?) \in \{running\} \\
&\quad \land\ state(topReady!) = ready \\
&\quad \land\ (\forall\, rtsk : state^{\sim}(\!|\ \{ready\}\ |\!) \bullet priority(topReady!) \geq priority(rtsk)) \\
&\quad \bullet\ \neg\ (Task[log\_context := log\_context \oplus \{(target?, bare\_context)\}, \\
&\qquad\qquad phys\_context := log\_context(topReady!), \\
&\qquad\qquad running\_task := topReady!, \\
&\qquad\qquad state := state \oplus \\
&\qquad\qquad\quad (\{(target?, nonexistent)\} \cup \{(topReady!, running)\}), \\
&\qquad\qquad tasks := tasks \setminus \{target?\}] \\
&\qquad \land\ (st \in TASK \\
&\qquad\qquad \land\ \neg\ (state \oplus (\{(target?, nonexistent)\} \cup \\
&\qquad\qquad\quad \{(topReady!, running)\}))(st) = state(st) \\
&\qquad\qquad \Rightarrow (state(st), (state \oplus (\{(target?, nonexistent)\} \cup \\
&\qquad\qquad\quad \{(topReady!, running)\}))(st)) \in transition) \\
&\qquad \Rightarrow\ t \in TASK \\
&\qquad\qquad \land\ state(t) = ready \\
&\qquad\qquad \land\ \neg\ priority(topReady!) \geq priority(t))
\end{aligned}
$$

Generally, the purpose of Theorem 11 is to prove and notify Z/Eves that for all the state that satisfies the definition of *Task*, based on the precondition of schema *DeleteTaskS_T*, the proof goal (4) is *false*. With this information and the following script, Z/Eves can easily prove the result of Theorem 10 is *true*. *DeleteTaskS_T_vc_ref* can be continued.

proof [*DeleteTaskS_T_vc_ref*]
use topReadyTask[$a := state^{\sim}(\!|\ \{ready\}\ |\!)$];
with disabled (*Task*) *prove by reduce*;
instantiate running_task' == $f(state^{\sim}(\!|\ \{ready\}\ |\!))$;
prove;
use lDeleteTaskS_T_Lemma[*topReady!* := $f(state^{\sim}(\!|\ \{ready\}\ |\!))$];
prove;
instantiate t__0 == *rtsk*;
prove;
■

3.4. Executing tasks

In FreeRTOS, there is no API for this: once the task scheduled, it will be executed automatically. However, it is helpful for specifications to show the task be executed, especially when executing the specification with an animator. In detail, when the processor executes a task, it updates registers, flags, memory locations, and so on. We model this by updating the physical context of the processor. Here, we are not interested in the new value after the operation. What we would like to know is that it is changed and the new value has some special property. Therefore, we use a nondeterministic definition again for updating *phys_context*. Because this schema describes executing the task, if the new value of *phys_context* is different from the original one, it would be satisfied.

3.5. Suspending/resuming tasks

Just like creating and deleting, suspending and resuming tasks also have two cases. When the system suspends a ready or blocked task, it does not lead to rescheduling; however, if the task to be suspended is the running task, then the system needs to find another task to take the processor. If a resumed task has a higher priority than the running task, it becomes the new running task; otherwise, it goes to the ready state. As mentioned above, normally the handle of the *idle* task is not obtainable. Even though the user may extend the behaviour of the *idle*

task by modifying the `vApplicationIdleHook` function, the *idle* task must never be suspended [Bar12b], and consequently can never be resumed. It is possible to suspend a suspended task: the system just keeps everything the same as before. So the first case concerns suspending a task that is ready or blocked; the only change necessary is to update the task's state. The following script shows the precondition theorem and proof script of the schema of this case.

Theorem 12 (SuspendTaskN_T_vc_ref)

$$\forall\, SuspendTaskN\_TFSBSig \mid true \bullet \text{pre}\ SuspendTaskN\_T$$

proof [*SuspendTaskN_T_vc_ref*]
prove by reduce;
apply extensionality to predicate $TASK \setminus (state \sim (\!| \{nonexistent\} |\!)) =$
$TASK \setminus ((state \oplus \{(target?, suspended)\}) \sim (\!| \{nonexistent\} |\!))$;
apply extensionality to predicate $(state \oplus$
$\{(target?, suspended)\}) \sim (\!| \{running\} |\!) = state \sim (\!| \{running\} |\!)$;
instantiate $pt\_0 == pt$;
prove;
apply applyOverride;
with normalization prove;
■

Due to the complication of the proof goal, the final proof command "*with normalization prove*; "requires a significant amount of time to complete. However, if we use the "*cases*, *next*" commands to separate the proof goals into different cases and then apply "*with normalization prove*; ", it becomes much more efficient.

The second case of the suspend operation is that the suspended task is the running task. Clearly, this leads to rescheduling. This operation ensures that the running task is not the idle one. It selects a target that is running and is one with the greatest priority of all ready tasks (there may be many such tasks). The *Reschedule* schema is used to achieve the necessary rescheduling. Similar to *DeleteTaskS_T*, a nondeterministically chosen value is assigned to *running_task'*. The prover is confused about its value. An additional theorem, *lSuspendTaskS_T_Lemma*, is introduced to help the prover with the precondition. Finally, it is also possible to suspend a suspended task. According to the reference manual of FreeRTOS [Bar12a], nothing changes when a suspended task is suspended. A single call to `vTaskResume` can resume the task that has been suspended several times. For this case, in schema *SuspendTaskO_T*, predicate $\Xi Task$ is used to represent the pre and post value of all variables within *Task* schema are consistent.

Similarly, the first case of resuming a task does not cause rescheduling. The priority of the resumed task must be no higher than the running task. The task is simply moved to *ready* state and keeps everything else unchanged.

In the second case, the resumed task has a higher priority than the running task, and rescheduling is required. Again, the schema *Reschedule* is used to approach this.

3.6. Changing priority of tasks

Because the priority of the *idle* task is permanently 0, if the target task is *idle*, the *newpri*? should equal 0. Specifically, to change the priority of tasks, there are three different cases that need to be considered. In the first case, there is no scheduling required, and this follows if one of the following conditions hold.

1. The target is the `running` task and the new priority is at least as high as every other ready task.
2. The target is `ready` and the new priority does not have a greater priority than the running task.
3. The target is the idle task and the new priority is 0.
4. The target is blocked.
5. The target is suspended.

Note that we cannot change the priority of *nonexistent* tasks. Further, as the set *TASK* is composed of *running*, *ready*, *blocked*, *suspend*, and *nonexistent*, tasks in these states are disjoint. Therefore, the predicate $state(target?) \neq nonexistent$ could imply that the *target*? is one of the other four states. That means for the conditions related to the *blocked* and *suspended* states, we do not need other predicates. Finally, the effect of the operation is to change only the priority of the target, nothing else. Then, we update the function *priority* by overriding the priority of the *target*? task with *newpri*?.

Table 1. API mappings & preconditions for operations

API	Operation	Precondition
xTaskCreate	$CreateTask\_T \mathrel{\widehat=} CreateTaskN\_T \lor CreateTaskS\_T$	$state(target?) = nonexistent$
	$CreateTaskN\_T$	$state(target?) = nonexistent$ $newpri? \leq priority(running\_task)$
	$CreateTaskS\_T$	$state(target?) = nonexistent$ $newpri? > priority(running\_task)$
vTaskDelete	$DeleteTask\_T \mathrel{\widehat=} DeleteTaskN\_T \lor DeleteTaskS\_T$	$target? \in tasks \setminus \{idle\}$ $state(target?) = running \Rightarrow$ $(\exists topReady! : state^{\sim}(\!\mid \{ready\} \mid\!)$ $\bullet (\forall t : state^{\sim}(\!\mid \{ready\} \mid\!)$ $\bullet priority(topReady!) \geq priority(t)))$
	$DeleteTaskN\_T$	$target? \in tasks \setminus \{idle\}$ $state(target?) \in \{ready, blocked, suspended\}$
	$DeleteTaskS\_T$	$target? \in tasks \setminus \{idle\}$ $state(target?) \in \{running\}$ $\exists topReady! : state^{\sim}(\!\mid \{ready\} \mid\!)$ $\bullet (\forall t : state^{\sim}(\!\mid \{ready\} \mid\!)$ $\bullet priority(topReady!) \geq priority(t))$
-	$ExecuteRunningTask\_T$	$\exists phys\_context' : CONTEXT$ $\bullet phys\_context' \neq phys\_context$
vTaskSuspend	$SuspendTask\_T \mathrel{\widehat=} SuspendTaskN\_T \lor SuspendTaskS\_T \lor SuspendTaskO\_T$	$target? \in tasks \setminus \{idle\}$ $state(target?) = running \Rightarrow$ $(\exists topReady! : state^{\sim}(\!\mid \{ready\} \mid\!)$ $\bullet (\forall t : state^{\sim}(\!\mid \{ready\} \mid\!)$ $\bullet priority(topReady!) \geq priority(t)))$
	$SuspendTaskN\_T$	$target? \in tasks \setminus \{idle\}$ $state(target?) \in \{ready, blocked\}$
	$SuspendTaskS\_T$	$target? \in tasks \setminus \{idle\}$ $state(target?) \in \{running\}$ $\exists topReady! : state^{\sim}(\!\mid \{ready\} \mid\!)$ $\bullet (\forall t : state^{\sim}(\!\mid \{ready\} \mid\!)$ $\bullet priority(topReady!) \geq priority(t))$
	$SuspendTaskO\_T$	$state(target?) = suspended$
vTaskResume	$ResumeTask\_T \mathrel{\widehat=} ResumeTaskN\_T \lor ResumeTaskS\_T$	$state(target?) = suspended$
	$ResumeTaskN\_T$	$state(target?) = suspended$ $priority(target?) \leq priority(running\_task)$
	$ResumeTaskS\_T$	$state(target?) = suspended$ $priority(target?) > priority(running\_task)$

In the second case, the target is a ready task whose new priority is higher than that of the running task. The target displaces the running task as the tasks are rescheduled. Similarly, the *Reschedule* schema is used to achieve this.

Third, similar to the second case, rescheduling is required. However, the target task, whose priority we wish to change, is the running task. Meanwhile, the new priority is not the greatest one among the ready tasks. The schema for this would firstly pick up the task with the highest priority among the ready tasks. It updates the value of the priority of the running task. Finally, it reschedules the system with the *Reschedule* schema. The variable "*topReady*!" here, similar to the schedule case of delete task and suspend task, is used to represent which ready task holds the highest priority among other ready tasks and would be scheduled as the new running task after the operation. Also, the schema *lChangeTaskPriorityD_T_Lemma* is introduced to handle the nondeterministically chosen value of "*running_task*′".

Table 2. API mappings & preconditions for operations(continue)

vTaskPrioritySet	*ChangeTaskPriority_T* $\widehat{=}$ *ChangeTaskPriorityN_T* $\lor$ *ChangeTaskPriorityS_T* $\lor$ *ChangeTaskPriorityD_T*	$state(target?) \neq nonexistent$ $target? = idle \Rightarrow newpri? = 0$ $\exists topReady! : state^{\sim}(\!\mid \{ready\} \mid\!\!)$ $\bullet (state(target?) \in \{running\}$ $\land \neg (\forall rtsk : TASK \mid state(rtsk) = ready$ $\bullet newpri? \geq priority(rtsk)))$ $\Rightarrow newpri? < priority(topReady!)$ $\land (\forall t : state^{\sim}(\!\mid \{ready\} \mid\!\!)$ $\bullet priority(topReady!) \geq priority(t))$
	ChangeTaskPriorityN_T	$state(target?) = ready \Rightarrow$ $newpri \leq priority(running\_task)$ $state(target?) = running$ $\Rightarrow (\forall t : state^{\sim}(\!\mid \{ready\} \mid\!\!) \bullet newpri? \geq priority(t))$ $state(target?) \neq nonexistent$ $target? = idle \Rightarrow newpri? = 0$
	ChangeTaskPriorityS_T	$state(target?) = ready$ $newpri? > priority(running\_task)$ $target? = idle \Rightarrow newpri? = 0$
	ChangeTaskPriorityD_T	$state(target?) \in \{running\}$ $target? = idle \Rightarrow newpri? = 0$ $\exists topReady! : state^{\sim}(\!\mid \{ready\} \mid\!\!)$ $\bullet newpri < priority(topReady!)$ $\land (\forall t : state^{\sim}(\!\mid \{ready\} \mid\!\!)$ $\bullet priority(topReady!) \geq priority(t))$

3.7. Summary of interface

We collect together the preconditions for the interface, and the API mapping in Tables 1 and 2. When we define the schemas for APIs, to simplify the specification, we use different schemas to define the different cases of a API. Therefore, we use disjunction to connect them together to get the schema that represents the API in FreeRTOS. Meanwhile, the precondition for these new schemas is also obtained from the preconditions of the old schemas, which are disjoined as well. For instance, the FreeRTOS API for creating a task, `xTaskCreate`, is represented by the schema *CreateTask_T*, which has two cases, *CreateTaskN_T* and *CreateTaskS_T*, as we defined above. Therefore, it is defined by these two sub-schemas linked by "$\lor$". For the first case, the precondition is that *target*? is unknown by the system and the priority of new task is lower than or equal to the priority of running task. Meanwhile, the precondition for the second case is that *target*? is unknown by the system as well, plus the priority of new task is greater than the one of the running task. Therefore, the precondition for the new schema is only that *target*? is unknown by the system before the operation.

Based on the content of this table, it is possible to produce the code-level annotations for VCC. These preconditions can be used in VCC as the content of requires clauses, _(*requires* ...). Further, the postconditions of the schemas can also be transferred into ensures clauses, _(*ensures* ...), of the notation of VCC. In our research, we also verified the task-related functions with VCC. Due to the limitation of the scope and the length of this paper, we have not included these details.

3.8. Some properties

Finally, there are some properties that need to be verified for our specifications. Some of them help us to ensure our specifications have correct behaviours, the properties of the system are consistent with the API document and source code, *etc*. Meanwhile, others are used to help Z/Eves prove our model correct. These theorems may seem trivial to the human eye; however, they are particularly helpful for the prover. Therefore, in this section, we present only a few of these theorems as examples. The details can be found on the web page.

1. As described above, in some schemas we need to find the task with the highest priority among all ready tasks. In these cases, it is important to ensure that the running task is not a member of the ready tasks. Otherwise, the reschedule algorithm would be chaotic. Moreover, it also important for the prover to prove the related properties of the task. For instance, it helps proving the theorem *TaskProperty*6.

Theorem 13 (TaskProperty3)

$$\forall\, Task;\ t : state^{\sim}(\!|\ \{ready\}\ |\!) \bullet t \in tasks \setminus \{running\_task\}$$

2. The variable *tasks* is used to record tasks known in the system. In other words, if the task is not recorded by *tasks*, it should unknown by the system. This theorem is helpful for the prover to sort out the state of this task is *nonexistent*.

Theorem 14 (TaskProperty4)

$$\forall\, Task;\ t : TASK \setminus tasks \bullet state(t) = nonexistent$$

3. As defined by the FreeRTOS, the *idle* task has the lowest priority all the time. If the priority of a task is greater than 0, this task cannot be *idle* task. The command *prove by reduce* can be used for proving this.

Theorem 15 (TaskPriority5)

$$\forall\, Task;\ t : tasks \mid priority(t) > 0 \bullet t \neq idle$$

4. It is also interesting to check that the behaviour of the operation schemas are properly described. To illustrate this, we select the schema *SuspendTaskS_T* to check. This theorem will check for any proper case of *Task*, after the *SuspendTaskS_T* operation, the old running task should be suspended and the new *running_task* has the highest priority among the ready tasks. To prove this theorem, the theorem *TaskProperty*3 will be used. Following that, we apply the one-point rule to the condition. Then the goal can be proved.

Theorem 16 (TaskProperty6)

$$\begin{aligned}&\forall\, Task \mid SuspendTaskS\_T\\ &\quad \bullet\ state'(running\_task) = suspended\\ &\quad \wedge\ (\forall\, t : state^{\sim}(\!|\ \{ready\}\ |\!) \bullet priority(running\_task') \geq priority(t))\end{aligned}$$

proof [*TaskProperty*6]
with disabled (Δ *Task*, *Task*) *prove by reduce*;
*use TaskProperty*3[t := *target*!];
instantiate $t\_\_0 == t$;
prove;

■

4. Case study

As promised in Sect. 2.2, we show how our specification illustrates the execution of FreeRTOS code in Fig. 2.

Firstly, ProZ is used to animate the model. Before animation, we set the size of our given sets to 4; the maximum integer needs to be set to 4 as well. This is because we have four tasks in the application, `idle`, `Task1`, `Task2`, and `Task3`, and the maximum number used is the priority of `Task3`, namely 4. Afterwards, the `.tex` file is loaded in ProZ. Although we have three tasks in the application, it executes in a single-core processor. Therefore, it is possible to sort out the sequence of API calling, which is

xTaskCreate Create `Task1` with priority of 1;
xTaskCreate Create `Task2` with priority of 2;
vTaskPrioritySet Change the priority of `Task1` to 3;
xTaskCreate Create `Task3` with priority of 4;
vTaskDelete Delete `Task3`;

After initialising the machine in ProZ, we call the API in this order (See Fig. 3). As we analysed in Sect. 2.2, the expected final state of execution should be:

1. There are three tasks left in the system, `idle`, `Task1`, and `Task2`;
2. `Task1` is the running task, as it has priority of 3.
3. `Task2` is in ready state with priority 2.
4. `Task3` is unknown to the system, therefore its state is nonexistent.

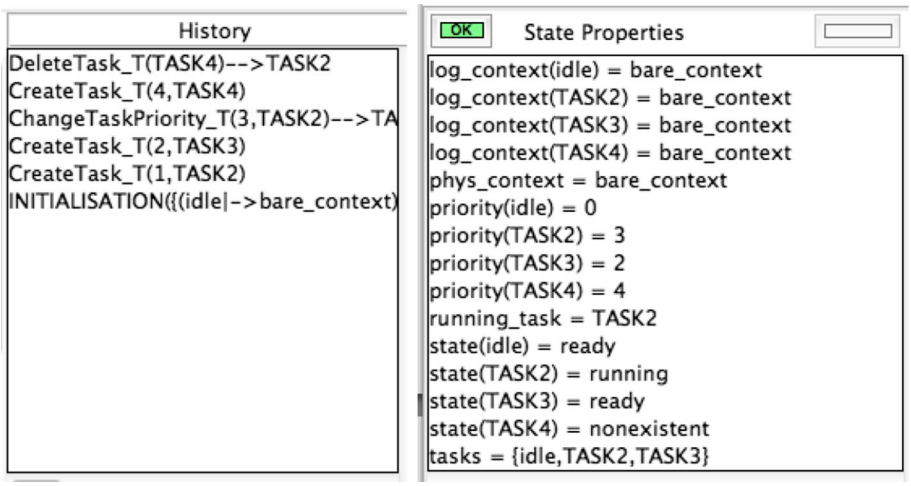

Fig. 3. Screen shot for API execution history and result from ProZ

As we can see from the screen shot of the state properties, the result generated from our model matches our expectation.

Besides this, we also let Z/Eves verify our result. We use a theorem, similar to Theorem 16, to show that the behaviour of the API matches our expectation. The API call `xTaskCreate` is repeated three time in the application; we only show the theorem for one of these calls. Therefore, we have the following theorems to show our model works for the application.

1. Create `Task1`. When we execute this, there is only the `idle` task in the system, which is the running task; therefore, we have to indicate this situation to the prover. The input variables need to be introduced and we need to specify the value of *newpri*? as 1. After this operation, we expect that the new task is created, which means it is in the set *tasks*′. It should be the running task with priority 1, which is higher than the priority of the `idle` task.

   Theorem 17 (CaseStudyStep1)

   $$\forall\, Task;\ target? : TASK;\ newpri? : \mathbb{N}$$
   $$\mid tasks = \{idle\} \land running\_task = idle \land newpri? = 1 \land CreateTask\_T$$
   $$\bullet\ target? \in tasks' \land state'(target?) = running \land priority'(target?) = 1$$

   To prove this, we know that the system needs to be scheduled. Therefore, we try to eliminate the non-schedule part of the specification of *CreateTask_T*. The key condition to distinguish these two cases is whether the priority of the new task is greater than the running task. Thus, we expand the necessary schemas of the proof goal and then let the prover discharge the proof goal automatically by the *prove by reduce*; command.

   proof [*CaseStudyStep*1]
   with disabled (*CreateTaskS_T*, *StateData*, *TaskData*, *ContextData*) *reduce*;
   prove by reduce;

   ■

2. Change the priority of `Task1` to 3. Similarly, we need to inform the prover about the pre-state of the system. The key element of the expected result of this API call is that `Task1` is scheduled as running task with priority of 3. By eliminating the unrelated case of *ChangeTaskPriority_T*, like the previous case for *CreateTask_T*, it is easy to prove this theorem.

Theorem 18 (CaseStudyStep3)

$$\begin{aligned}&\forall\, Task;\ Task1, Task2, target? : TASK;\ newpri? : \mathbb{N}\\ &\quad \mid tasks = \{idle, Task1, Task2\}\\ &\quad \land\ priority(Task1) = 1 \land priority(Task2) = 2\\ &\quad \land\ state(Task1) = ready \land running\_task = Task2\\ &\quad \land\ target? = Task1 \land newpri? = 3\\ &\quad \land\ ChangeTaskPriority\_T \bullet priority'(Task1) = 3 \land running\_task' = Task1\end{aligned}$$

3. Finally, we verify the properties related to the last step of the API call, delete `Task3`. Following the strategy introduced before, the theorem and proof can be obtained. The only difficulty in proving this theorem is the nondeterministic definition for *topReady*!. In order to solve this, it is necessary to inform Z/Eves that (a) The possible value of *topReady*! is one of the elements of *tasks*. (b) The priority of *topReady*! is the greatest amongst all *ready* tasks, i.e., the priority of *topReady*! has to be greater than or equal to the priority of `Task1`.

Theorem 19 (CaseStudyStep5)

$$\begin{aligned}&\forall\, Task;\ Task1, Task2, Task3, target? : TASK;\ newpri? : \mathbb{N}\\ &\quad \mid tasks = \{idle, Task1, Task2, Task3\}\\ &\quad \land\ priority(Task1) = 3 \land priority(Task2) = 2 \land priority(Task3) = 4\\ &\quad \land\ state(Task1) = ready \land state(Task2) = ready \land state(Task3) = running\\ &\quad \land\ target? = Task3\\ &\quad \land\ DeleteTask\_T \bullet state'(Task3) = nonexistent \land running\_task' = Task1\end{aligned}$$

5. Conclusions

We have produced the first completely abstract specification of the task model of FreeRTOS. The model can be animated by the ProZ tool to show how FreeRTOS works. We have shown that the model is internally consistent by discharging all the verification conditions for well-definedness of the specification and by calculating the exact preconditions for the successful operation of each part of the FreeRTOS API. The model contains about 77 Z paragraphs. It also includes 88 theorems. All verification conditions and theorems have been proved by the Z/Eves theorem prover, either automatically (59 out of 88 theorems) or interactively (29 out of 88 theorems). The web-presentation contains all these proofs, so that the entire verification can be replayed to check its authenticity: crucially, our experiment is repeatable.

For future work, the model will be extended to include the behaviour of *Queue*s, *Time*, *Mutex*es, and *Interrupt*s. We are going to refine the entire development of our formal specification into running code, as close as possible to the existing code of FreeRTOS. This is not a trivial matter, even for a small system such as FreeRTOS. Due to the requirements for portability, there are different implementations for some interrupt-related operations, such as *taskENTER_CRITICAL*() and *taskEXIT_CRITICAL*(). Our work will involve targeting code using Microsoft's Verifying C Compiler, with separate verification of assembly language inserts. Finally, we plan to extend the work to multicore following the strategy laid out in [Mis11, MNW13].

We are keen to encourage others to use our specifications and proofs as benchmarks for comparing other notations and tools. This is in a similar spirit to our work on the Mondex smart-card [WSC+08], where we invited others to use a range of different formalisms and tools on the same problem to gather evidence about the current state of the art in mechanical verification (see [JW08]). As well as our formalisation and mechanisation in Z/Eves [FW08], the problem was tackled in Alloy [Ram08], ASM [HSGR08], Event-B [BY08], Maud [KOF07], OCL [KG08], PerfectDeveloper (unpublished), the π-calculus [JP07], RAISE [GH08], and VDM [ABF+11].

Acknowledgements

Part of this work has been funded under the United Kingdom-India Education and Research Initiative (UKIERI) project no. SA08-047 *Verified Software Initiative: Embedded Systems*, 2009–2012. We are grateful to our colleagues at the Indian Institute of Science in Bangalore for their helpful comments on presentations of this work: thanks to Sumesh Divakaran, Anuridh Kushwah, Virendra Singh, and Nigamanth Shridar (on leave from Ohio State University). Thanks are also due to our colleagues from York who have contributed to discussions

on FreeRTOS: Jeremy Jacob, Steve King, Chris Poskitt, James Williams, and Detlef Plump. The work has been presented at a Summer School at ICTAC 2011 and at the 2011 Microsoft Research Summit in Paris. Finally, we would like to thank Jean-Raymond Abrial, whose elegant specification of FreeRTOS in B provided the basis for our formalisation in Z.

A. Summary of Z/Eves proof commands

We summarise the proof commands used in proving of the model. For full instruction of the proof commands, please see Chap. 5 of [MS97].

prove The prover automatically applies sequences of proof commands. For example, *simplify*, *rewrite*, *rearrange*. Besides this, the mathematical rules included in Z/Eves' mathematical toolkit [Saa99a] are applied, if possible.

prove by reduce The prover repeatedly reduces the current proof goal. In addition to what **prove** does, the prover expands all names.

with enabled (theorem) This is a prefix that is applied to the *prove*, *prove by reduce*, or an already prefixed command. Many inefficient rules are disabled by default, and this prefix enables them for the current command. For example, *with enabled* (*applyOverride*) *prove* allows the prover to use the disabled theorem *applyOverride* within the scope of the *prove* command.

with disabled (theorem) This is similar to the previous command, except that it disables the theorem rather than enabling it.

with normalization This is also a prefix for prove commands. It allows the prover to use "if-then-else" normal-form to represent all logical connectives [Saa99a].

instantiate This command allows the prover to instantiate quantified variables (universal in the assumptions, existential in the goal).

apply theorem As mentioned above, there are plenty of disabled rules in Z/Eves' mathematical toolkit. This command applies the specified theorem to rewrite the goal.

use theorem This command allows a specified theorem to be used to deduce additional assumptions.

extensionality A theorem included in Z/Eves Mathematical Toolkits [Saa99a], which defined as:

$$X = Y \Leftrightarrow (\forall x : X \bullet x \in Y) \land (\forall y : Y \bullet y \in X)$$

References

[ABF+11] Andrews Z, Bryans J, Fitzgerald J, Hughes J, Payne R, Pierce K, Riddle S (2011) Modelling and refinement of the Mondex electronic purse in VDM. Technical Report Series 1308, Newcastle University School of Computing Science

[Art12] Arthan R (2012) ProofPower. http://www.lemma-one.com/ProofPower/

[Bar12a] Barry R (2012) FreeRTOS Reference Manual—API functions and configuration options. PDF book available from http://shop.freertos.org

[Bar12b] Barry R (2012) Using the FreeRTOS real time kernel—a practical guide. PDF book available from http://www.freertos.org

[BC09] Egon B, Craig I (2009) Modeling an operating system kernel. In: Diekert V, Weicker K, Weicker N (eds) Informatik als Dialog zwischen Theorie und Anwendung. Vieweg+Teubner, pp 199–216

[BJ10] Jacobs FPB, Smans J (2010) A quick tour of the verifast program verifier. In: APLAS 2010. Lecture notes in computer science, vol 6461. Springer, Berlin

[BY08] Butler M, Yadav D (2008) An incremental development of the Mondex system in Event-B. Formal Aspects Comput J 20(1):61–77

[Cra06] Craig ID (2006) Formal models of operating system kernels. Springer, Berlin

[Cra07] Craig ID (2007) Formal refinement for operating system kernels. Springer, Berlin

[DGM09] Déharbe D, Galvão S, Moreira AM (2009) Formalizing FreeRTOS: first steps. In: Oliveira MVM, Woodcock J (eds) Formal methods: foundations and applications, 12th Brazilian symposium on formal methods, SBMF 2009, Gramado, Brazil, August 19–21, 2009, Revised selected papers. Lecture notes in computer science, vol 5902. Springer, Berlin, pp 101–117

[FHQ12] Ferreira J, He G, Qin S (2012) Automated verification of the FreeRTOS scheduler in HIP/SLEEK. In: 6th international symposium on theoretical aspects of software engineering (TASE'12), 4–6 July

[FW08] Freitas L, Woodcock J (2008) Mechanising Mondex with Z/EVES. Formal Aspects Comput J 20(1)

[FW09] Freitas L, Woodcock J (2009) A chain datatype in Z. Int J Softw Inf 3(2–3):357–374

[GH08] George C, Haxthausen AE (2008) Specification, proof, and model checking of the Mondex electronic purse using RAISE. Formal Aspects Comput 20(1):101–116

[HMLS09] Hoare CAR, Misra J, Leavens GT, Shankar N (2009) The Verified Software Initiative: a manifesto. ACM Comput Surv 41(4)

[Hoa03] Hoare CAR (2003) The verifying compiler: a grand challenge for computing research. J ACM 50(1):63–69

[HSGR08] Haneberg D, Schellhorn G, Grandy H, Reif W (2008) Verification of Mondex electronic purses with KIV: from transactions to a security protocol. Formal Aspects Comput J 20(1):41–59
[JOW06] Jones C, O'Hearn P, Woodcock J (2006) Verified software: a grand challenge. IEEE Comput 39(4):93–95
[JP07] Jones CB, Pierce KG (2007) What can the π-calculus tell us about the Mondex purse system? In: 12th international conference on engineering of complex computer systems (ICECCS 2007), 10–14 July 2007, Auckland, New Zealand. IEEE Computer Society, pp 300–306
[JW08] Jones C, Woodcock J (2008) Special issue on Mondex. Formal Aspects Comput 20(1)
[KEH+09] Klein G, Elphinstone K, Heiser G, Andronick J, Cock D, Derrin P, Elkaduwe D, Engelhardt K, Kolanski R, Norrish M, Sewell T, Tuch H, Winwood S (2009) seL4: formal verification of an OS kernel. In SOSP, pp 207–220
[KG08] Kuhlmann M, Gogolla M (2008) Modeling and validating Mondex scenarios described in UML and OCL with USE. Formal Aspects Comput J 20(1):79–100
[Kle09] Klein G (2009) Operating system verification—an overview. Sādhanā 34(1):27–69
[Kle10a] Klein G (2010) A formally verified OS kernel. Now what? In Kaufmann M, Paulson LC (eds) Interactive theorem proving, first international conference, ITP 2010, Edinburgh, UK, July 11–14, 2010. Proceedings. Lecture notes in computer science, vol 6172. Springer, Berlin, pp 1–7
[Kle10b] Klein G (2010) From a verified kernel towards verified systems. In: Ueda K (ed) Programming languages and systems—8th Asian Symposium, APLAS 2010, Shanghai, China, November 28–December 1, 2010. Proceedings. Lecture notes in computer science, vol 6461. Springer, Berlin, pp 21–33
[Kle10c] Klein G (2010) The L4.verified project—next steps. In: Leavens GT, O'Hearn PW, Rajamani SK (eds) Verified software: theories, tools, experiments, third international conference, VSTTE 2010, Edinburgh, UK, August 16–19, 2010. Proceedings. Lecture notes in computer science, vol 6217. Springer, berlin, pp 86–96
[KOF07] Kong W, Ogata K, Futatsugi K (2007) Algebraic approaches to formal analysis of the Mondex electronic purse system. In: Davies J, Gibbons J (eds) Integrated formal methods, 6th international conference, IFM 2007, Oxford, UK, July 2–5, 2007, Proceedings. Lecture Notes in Computer Science, vol 4591. Springer, Berlin, pp 393–412
[Lab02] Labrosse JJ (2002) MicroC OS II: The real time kernel. Newnes
[Lin10] Lin Y (2010) Formal analysis of FreeRTOS. Master's thesis, University of York
[MF11] Muehlberg JT, Freitas L (2011) Verifying FreeRTOS: from requirements to binary code. In: Bendisposto J, Jones C, Leuschel M, Romanovsky A (eds) Proceedings of the 11th international workshop on automated verification of critical systems (AVoCS 2011). Electronic communications of the EASST, vol 10, pp 1–2
[Mis11] Mistry J (2011) FreeRTOS and multicore, September. MSc Dissertation, Department of Computer Science, University of York
[ML10] Mühlberg JT, Lüttgen G (2010) Symbolic object code analysis. In: van de Pol J, Weber M (eds) Model checking software—17th international SPIN Workshop, Enschede, The Netherlands, September 27–29, 2010. Proceedings, Lecture Notes in Computer Science, vol 6349. Springer, berlin, pp 4–21
[MNW13] Mistry J, Naylor M, Woodcock J (2013) FreeRTOS and multicore
[MS97] Meisels I, Saaltink M (1997) Z/Eves 1.5 Reference Manual. ORA Canada, TR-97-5493-03d
[OSR12] Owre S, Shankar N, Rushby J (2012) PVS. http://pvs.csl.sri.com/
[Pau12] Paulson L (2012) Isabelle. http://www.cl.cam.ac.uk/research/hvg/isabelle/
[Pro10] Pronk C (2010) Verifying FreeRTOS: a feasibility study. Technical Report TUD-SERG-2010-042, Delft University of Technology, Software Engineering Research Group
[Ram08] Ramananandro T (2008) Mondex, an electronic purse: Specification and refinement checks with the Alloy model-finding method. Formal Aspects Comput J 20(1):21–39
[Saa99a] Saaltink M (1999) Z/Eves 2.0 Mathematical Toolkit. ORA Canada, TR-99-5493-05b
[Saa99b] Saaltink M (1999) *Z/Eves 2.0 User's Guide*. ORA Canada, TR-99-5493-06a
[Spi92] Spivey JM (1992) The Z notation: a reference manual. Series in Computer Science, 2nd edn. Prentice Hall International
[SWG+11] Sewell T, Winwood S, Gammie P, Murray TC, Andronick J, Klein G (2011) seL4 enforces integrity. In: Marko C, van Eekelen JD, Geuvers H, Schmaltz J, Wiedijk F (eds) Interactive theorem proving—second international conference, ITP 2011, Berg en Dal, The Netherlands, August 22–25, 2011. Proceedings. Lecture notes in computer science, vol 6898. Springer, Berlin
[WD96] Woodcock J, Davies J (1996) Using Z: specification, refinement, and proof. International series in computer science. Prentice-Hall, Englewood Cliffs
[Woo06] Woodcock J (2006) First steps in the Verified Software Grand Challenge. IEEE Comput 39(10):57–64
[WSC+08] Woodcock J, Stepney S, Cooper D, Clark J, Jacob J (2008) The certification of the Mondex electronic purse to ITSEC Level E6. Formal Aspects Comput 20(1)

Received 15 October 2012
Revised 10 May 2014
Accepted 24 June 2014 by Cliff Jones
Published online 20 August 2014