Program Verification of FreeRTOS using Microsoft Dafny

MATTHEW J. MATIAS

Bachelor of Business Administration in Information Systems

Cleveland State University

December 2010

submitted in partial fulfillment of the requirements for the degree

MASTERS OF SCIENCE IN SOFTWARE ENGINEERING

at the

CLEVELAND STATE UNIVERSITY

May 2014

We hereby approve this thesis for

Matthew J. Matias

Candidate for the MASTERS OF SCIENCE IN SOFTWARE ENGINEERING

degree from the

Department of Electrical and Computer Engineering

and the CLEVELAND STATE UNIVERSITY

College of Graduate studies

Thesis Committee Chairperson, Dr. Nigamanth Sridhar

Department of Electrical and Computer Engineering, 05/06/2014

Thesis Committee Member, Dr. Chansu Yu

Department of Electrical and Computer Engineering, 05/06/2014

Thesis Committee Member, Dr. Yongjian Fu

Department of Electrical and Computer Engineering, 05/06/2014

To my family, friends, neighbors, colleagues, and acquaintances ...

ACKNOWLEDGMENTS

I thank Dr. Nigamanth Sridhar for his wisdom, advisement, patience, research and experience. He introduced me to formal methods and made this research possible from his academic endeavors and brilliance. I extend my gratitude to Dr. Sridhar's colleagues from the Resolve/Reusable Software Research Group for feedback and advisement on verifying data structures. Thank you Indian Institute of Science for sharing their research on formalizing FreeRTOS. The Z model is mentioned and referenced many times throughout this thesis. I thank Dr. Yongjian Fu for several hands-on software engineering courses in which documentation, specification, and implementation were covered in detail. These concepts are used frequently in this research. My gratitude to Sheari Rice for reviewing early revisions of my thesis along with her studies of various formal methods tools. Thank you Dr. Chansu Yu for your academic and leadership role in our department. I am grateful to Rustan Leino and Microsoft developers for creating Dafny and providing support on the discussion threads. I appreciate my friends for their moral support. Thank you to my family for their support, love, and kindness because they are simply the best.

Program Verification of FreeRTOS using Microsoft Dafny

MATTHEW J. MATIAS

ABSTRACT

FreeRTOS is a popular real-time and embedded operating system. Real-time software requires code reviews, software tests, and other various quality assurance activities to ensure minimal defects. This free and open-source operating system has claims of robustness and quality [26]. Real-time and embedded software is found commonly in systems directly impacting human life and require a low defect rate. In such critical software, traditional quality assurance may not suffice in minimizing software defects. When traditional software quality assurance is not enough for defect removal, software engineering formal methods may help minimize defects. A formal method such as program verification is useful for proving correctness in real-time software. Microsoft Research created Dafny for proving program correctness. It contains a programming language with specification constructs. A program verification tool such as Dafny allows for proving correctness of FreeRTOS's modules. We propose using Dafny to verify the correctness of FreeRTOS' scheduler and supporting API.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

Introduction

Software engineering attempts to solve the practical problems of developing software. It is a young field, but our society is increasingly dependent on software that is released commonly with defects such as bugs, faults, and failures. The branch of software engineering that directly attempts removing or preventing defects is software quality assurance (SQA). SQA identifies quality assurance activities that are necessary in creating low-defect software. Documentation, specifications, reviews, inspections, standards, and testing are activities common in removing software defects. SQA does not guarantee defect-free software, but the goal is an acceptable level of software quality [9].

The demand for SQA resulted from user intolerance to software defects. Some defects are tolerable in desktop and productivity software. If a word processor crashes, restarting the program will most likely remedy the crash and the user may continue working on a document. On the other hand, real-time software is not fixed by a simple restart. For example, an air traffic control system must continue running

and operating indefinitely; this is true especially if there are airplanes in mid-flight. It is unlikely human injury occurs from productivity software defects, but real-time systems require constraints to prevent accidents and danger to human life. There is a high cost for software defects in other types of real-time systems too. On September 23, 1999, NASA lost its Mars Climate Orbiter (MCO) due to a software defect in which English units were used rather than metric units [28]. The MCO's trajectory was miscalculated and the satellite's signal was lost that day. NASA's MCO project cost the MCO satellite's destruction, time, and money, but the result was ensuring the next Mars satellite, Mars Polar Lander (MPL), would land safely.

NASA's MCO Mishap Report mentions one of the contributing causes to MCO's software failure was an inadequate verification and validation process [28]. This is a common problem in software development. Specifications are written mostly in ambiguous natural language which may lead to defects. Inadequate validation and verification coverage may not show the specification was followed. This may also lead to defects. As a result of this common problem, there is a need for an unambiguous specification, adequate validation, and verification process.

Software engineering formal methods attempts to create unambiguous specifications and adequate validation and verification coverage. Formal specifications use different logics as an unambiguous language. Tools such as program verifiers prove the specification is correct. Formal methods are not a "silver bullet" [5], but it provides another SQA activity that may minimize software defects.

1.1 Problem

Fortunately, software defects in NASA's MCO did not cost human life or injury. Real-time and embedded software are used often in critical systems. FreeRTOS is an operating system used in embedded systems. While this software has claims of robustness from rigorous testing, there are research projects formalizing FreeRTOS [7] [8] [6].

Projects such as the Indian Institute of Science's (IISc) Z model formally specifies FreeRTOS in which the data structures and scheduler are specified [7] [8]. Z is a formal specification language and there is tool support for validating Z schema such as ProZ [24]. The IISc research converts FreeRTOS' API documentation into a formal model. This is quite difficult because natural language is converted into a formal language, Z. Natural language is ambiguous and the translation to formalization is open to interpretation. In addition, the Z model is not coupled to any program implementation. The implementation may consist of pen-and-paper proofs or a complete guess. Also, there are tools to refine a formal model into implementation code [6]. A common goal in formal methods is to refine the formal model or specification into executable code.

Various formal methods tools can formalize FreeRTOS including model-checkers and program verifiers. A model checker can check the Z model for consistency. If the Z model was given to a model checker, it will show if the model's attributes hold. Although using this tool is feasible, there is a disconnect between the formal model and program implementation [13]. In addition, the Z schema needs to translate into the model checker's language. Refining a formal model to executable code is not a trivial task, but model-driven design methods exist which allows for refining the model and eventually creating an implementation [6]. Although model checkers can formalize

FreeRTOS, there is a better tool. A program verifier can formalize FreeRTOS based on a Z model or documentation. Microsoft Research has several tools used to prove functional correctness such as Dafny, VCC, SPEC#, and HAVOC [1]. Each tool contains a programming language and specification constructs. The API documentation needs a conversion into the verifier's specification constructs and constructing code is required. The implementation is proven correct based on the specification in the program verifier process.

Model checking or program verification are sufficient formal methods tools that may verify FreeRTOS's API. Both tools can create a formal specification. However, a program verifier such as Dafny supports refinement into program code without the use of additional tools. This verifier contains specification constructs and a programming language for formalizing FreeRTOS. Dafny is the tool used in this thesis because it allows for creating a specification and code for verifying FreeRTOS's scheduler and supporting data structures.

1.2 Thesis

This thesis creates a formal specification and verified implementation of FreeRTOS's scheduler with a program verifier. The scheduler's supporting data structures and API are specified in Dafny. Therefore, the data structures and API have an implementation in which it is correct in respect to the specification. The Dafny specification is based on the existing API documentation in FreeRTOS's code base and IISc's Z model. The challenge is converting natural language into a specification and creating an implementation from the API documentation. IISc's Z model is referenced also providing an unambiguous specification.

[1] see http://research.microsoft.com/en-us/projects/boogie/

1.3 Solution Approach

The overall process starts with documentation and it is finalized with an implementation. Formalizing FreeRTOS requires selecting a tool in which a creating a specification is possible. Dafny is used because it provides specification constructs for formalizations and programming language to construct an implementation. In addition, references from existing documentation must provide guidance for creating a specification and translating it into a formal methods tool. After the formalized specification is created, an implementation is constructed. Dafny verifies the code correct based on the specification.

The API documentation is reviewed and converted into Dafny along with simultaneously referencing IISc's `Z` model. The functional and behavioral requirements are captured. This provides the what and how the task lists and scheduler operates. The API documentation describes each function and provides examples how each is called. Constraints are stated in the documentation in natural language also. Some functions have preconditions before a call occurs. The `Z` model further specifies the constraints in an unambiguous language and formalizes the operations. Each formalized schema contains the behavioral attributes of an operation without containing implementation details. The schema do not directly convert into Dafny and API documentation is used when needed. Classes and methods are created to contain class invariants, framing annotations, annotated loops, preconditions, and postconditions. Modules allow refining a class containing only specifications into an implementation. Each method signature is kept as close as possible to its associated API function signature. Again, the API documentation is referenced to closely model the functional and behavioral requirements. As a result of reviewing and converting the API documentation and `Z` model, the final artifact of this step contains a Dafny specification without an

implementation.

A formalized specification is refined into an implementation. Code is created which corresponds to the formalized specification. This is accomplished in Dafny. Some of the code is based on the existing FreeRTOS implementation and API functions are referenced. In contrast, other code is constructed only from the Dafny specification. This code may or may not execute, but the scheduler data structures and algorithms are captured and formalized. The resulting and final artifact is a formal specification and verified construction in Dafny.

1.4 Contributions

The contributions include a formalization of FreeRTOS and practical application of the Dafny program verifier. This is a software engineering formal methods experiment that applies software verification to a real-time system. This is a directional step related to "verified software's grand challenge" (Jones et al) where formal methods allow developers to build trustworthy, reliable, robust, and low-defect software [16]. A verified FreeRTOS contains an unambiguous formal specification and correct implementation. The Dafny community may benefit from observing how well the tool verifies a software system. The FreeRTOS community may benefit by expanding the existing documentation and following more quality assurance activities to minimize and eliminate defects.

The following statements summarize the contributions:

- FreeRTOS is formalized into a Dafny specification and implementation. This documents the behavior required and constraints ensured. `xList` and the scheduler API are specified and verified code is constructed.

- The Dafny code captures the data structures, algorithms, and scheduler operations which can translate into another language or formal methods tool. For example, the code can convert to another programming language in which it can execute; or it is translated into a tool such as `VCC`, an annotated C language.

- This work shows examples of using Dafny. It provides a practical example of the Dafny program verifier. Common constructs are used including classes, methods, modules, framing, invariants, and contracts. The examples may expose improvements needed in the tool.

- A possible intermediate step for IISc's work on formalizing FreeRTOS where the `Z` model is translated into Dafny, then `VCC`. Converting `Z` to `VCC` is difficult because C is much lower-level than the modeling notation. Dafny is higher-level than `VCC` and contain similar specification constructs. It may be simpler to translate `Z` to Dafny to `VCC` rather than `Z` to `VCC`.

1.5 Organization of Thesis

This chapter introduces the thesis in which the problem, topic, solution, and contributions are described. The second chapter explains FreeRTOS's design and architecture. The scheduler and task list behavior are described along with task states and timing constraints. A list of API calls are presented and object-oriented design techniques for the scheduler and data structures are shown. Next, chapter three explains common specification and programming constructs used in Dafny. The language and annotations utilized in verification are shown with tutorial-like examples. Chapter four verifies the task lists built on the high-level designed `xList`. The list data structure specifications and implementations are verified. The specified data structures include a priority queue, FIFO queue, and unordered list. Afterwards,

chapter five verifies the specification and construction of FreeRTOS's scheduler. The chapter introduces the scheduler module and class declaration. Each method is shown in a figure and described sequentially. The specifications are quite detailed and presented with figures and textual descriptions. Large method contracts are written in separate logic formulas. The final chapters include lessons learned from Dafny and related work. Notable Dafny features and improvements are mentioned along with other formal methods projects. The final chapter describes the accomplishments and future works.

CHAPTER II

FreeRTOS

This chapter provides background information on FreeRTOS. The purpose is demonstrating the system aspects required for a formal specification and verification in later chapters. FreeRTOS is a real-time operating system and must guarantee events occur in a time interval. The proper management of temporal events is common in real-time software [30]. These events include scheduling tasks and running the highest priority task. The scheduler algorithm must follow such a guarantee. The scheduler utilizes several data structures to accomplish this guarantee. In addition, several functions are included in the operating system's API for managing tasks.

Section 1 provides a brief introduction defining FreeRTOS. Section 2 discusses FreeRTOS's Architecture which includes the scheduler, tasks, list data structures, and API calls. Section 3 discusses modeling FreeRTOS with an object-oriented design.

2.1 What is FreeRTOS?

FreeRTOS is abbreviated *Free Real-time Operating System* because it is embedded software and a real-time operating system. *Embedded software* is defined as software used in embedded devices such as microcontrollers and an *embedded device* refers to a cost-effective and low-resource computing electronic. *Real-time* guarantees events happen in a specific time interval. This guarantee makes FreeRTOS a hard real-time system [30]. In addition, a real-time operating system such as FreeRTOS supports multitasking, memory management, scheduling, prioritizing tasks, and real-time events. Approximately thirty-four hardware platforms are supported. The supported hardware architectures include microcontrollers such as ARM, Atmel, and PIC. In order for FreeRTOS to fit on a microcontroller, the kernel size is small: it is approximately 3-9 KLOC depending on the port. The kernel is licensed by GNU GPL, free, and open-source. It has also grown in popularity being downloaded 103,000 times in 2012. FreeRTOS is actively supported by its community and claims robustness and dependability [1].

2.2 FreeRTOS Architecture

Figure 1 shows the software architecture of FreeRTOS. There are port and non-port modules in FreeRTOS. Port modules include device drivers and any software specific to either a processor or microcontroller hardware implementation. Modules are written in assembly language or C [15]. Each port has its own configuration which is shown as a module in Figure 1. Some header files contain macros for configuring and compiling for specific microcontrollers. For example, CPU clock and system tick rate macros sets the frequencies specific for an ARM processor. The implementation is different depending on the microcontroller and compiler used. The port code is

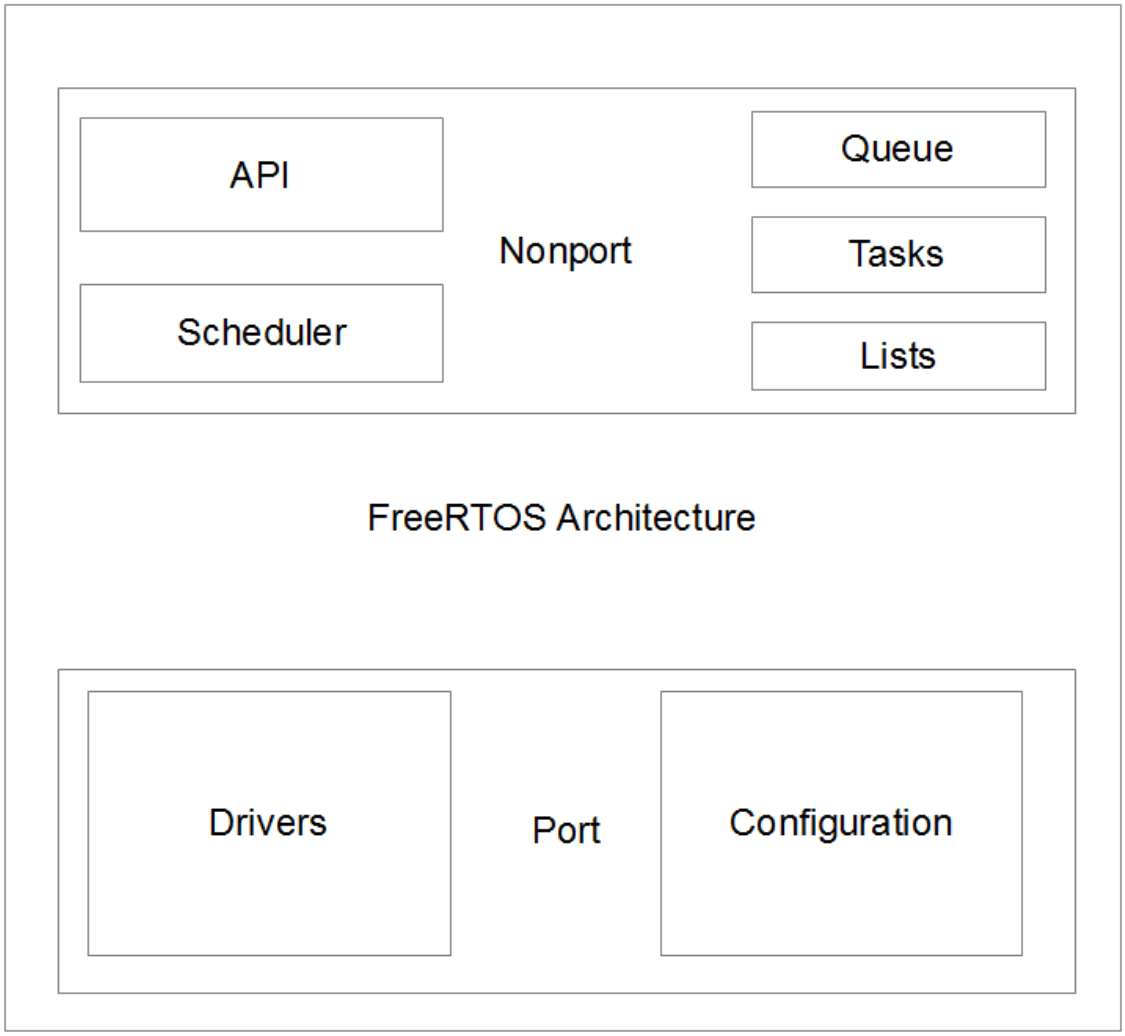

Figure 1: FreeRTOS Architecture

found in the source folder in FreeRTOS's code.

Non-port modules include code such as tasks, scheduler, queues, and lists. The scheduler is FreeRTOS's core component [29]. The scheduler must guarantee each task is given a fair-share of processor time by managing tasks. In this case, fair-share of processor refers to a properly scheduled task: highest priority tasks schedule and run before lower priority tasks. Task scheduling is a critical role of any operating system, but a real-time system must guarantee priority tasks are given processor time. Also, FreeRTOS is hard real-time software: events must occur in a specific time. It is not a soft real-time system where timed events can occasionally miss deadlines [30]. Most of FreeRTOS's code base is ensuring tasks are given the processor and time guarantee. The non-port code is found in every FreeRTOS port and includes the files list.c, tasks.c, and queues.c [25].

An application programmer interface (API) is available regardless of the system FreeRTOS is compiled. This API presents a set of functions for accessing the scheduler, queues, tasks, and lists. FreeRTOS can execute application programs on the system. Application programmers can write programs to run on FreeRTOS.

2.2.1 The Scheduler

The highest priority tasks must schedule and run first. FreeRTOS's scheduler is configured either as preemptive or non-preemptive scheduling. In preemptive scheduling, a task runs for a specific time interval. Once the time interval reaches zero, the task is changed to a suspended state [30]. Low priority tasks are interrupted by high priority tasks in preemptive scheduling, but non-preemptive scheduling ensures low or high priority tasks will run until completion, then high priority tasks may run afters. Low priority tasks are not interrupted and run until completion in non-preemptive

scheduling [30].

Preemptive scheduling is available for architectures which contain a clock interrupt. However, some systems may not have this interrupt. In this case, non-preemptive scheduling is used because a clock interrupt is not required for this scheduling algorithm. Also, the lack of a clock interrupt is the reason tasks are not interrupted in non-preemptive scheduling.

Tasks must run in a certain time interval. FreeRTOS must have a timed heartbeat [29]. Any time the operating systems has a heartbeat, the scheduler must run the highest priority task. This heartbeat is an interrupt called the system tick shown in Figure 2. Thus, when the system tick interrupts, the highest priority task is selected and the task is scheduled as the currently running task. When the system tick interrupts, the highest priority task enters the running state. The FreeRTOS scheduler's main job is to ensure the highest-priority task is always scheduled and running. The highest-priority task is any task from a ready list array where the ready list contains all of the highest-priority tasks. However, tasks with the same priority (i.e. tasks in the same ready list) follow round-robin scheduling.

Figure 2 shows how the system tick affects scheduled tasks. Assume there are three tasks: `Task1-1` and `Task1-2` have a priority of one and `Task2-1` has a priority of two. `Task1-1` and `Task1-2` have higher priority then `Task1-2`. `Task1-1` is scheduled and running in the first period of Figure 2 while `Task1-2` and `Task2-1` are not running. In the second period, `Task1-2` is running since `Task1-1` and `Task1-2` have equal priority and are scheduled in a round-robin fashion. `Task2-1` is not running. The third period has `Task1-1` as the running task again while `Task1-2` and `Task2-1` are not running. In period four, `Task1-2` is running. `Task1-1` and `Task1-2` are completed by the fifth

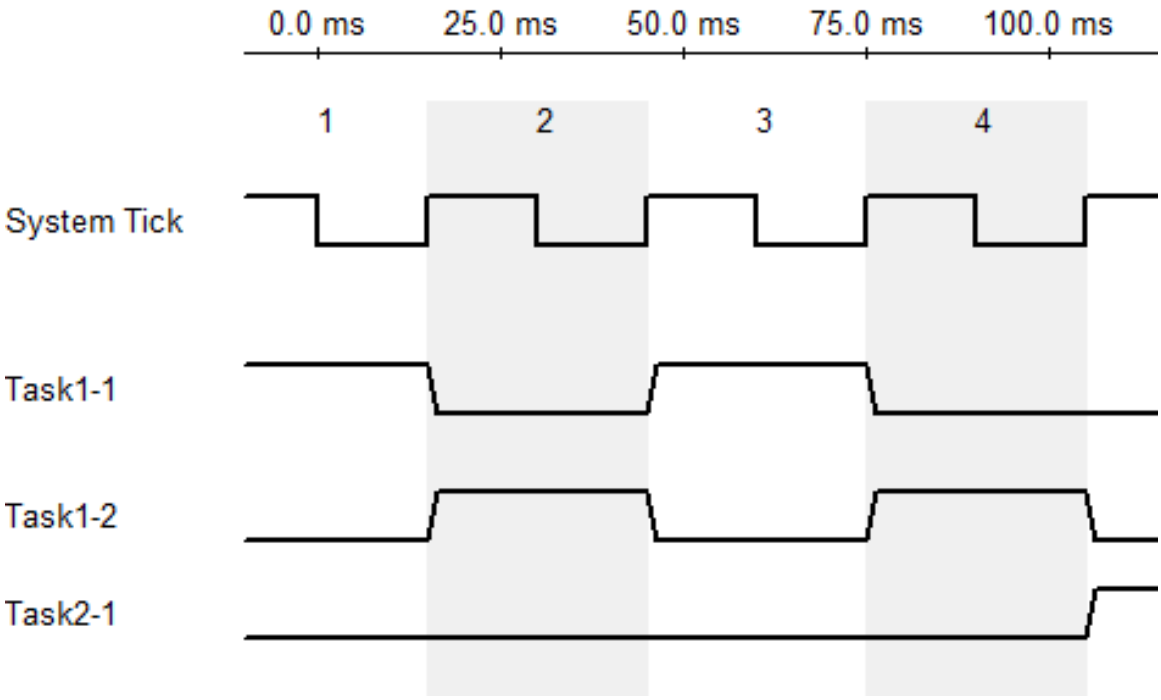

Figure 2: System Tick and Scheduled Tasks

period. `Task2-1` runs in period five because it is the highest priority task.

Each FreeRTOS port contains a system tick. It is essential for real-time software. The system tick measures timed events. At the end of a tick, the running task and ready list is checked. If the running tasks's time out period expires, it is inserted into the suspended list. Whether or not the running task expires, the next task in the ready list becomes the running task. Tasks in the same ready list run in a round-robin fashion. This is the same tasks having the same priority. Lower priority tasks do not enter the running state until higher priority tasks are completed.

2.2.2 Tasks

A task is the main unit of work in an operating systems. An operating system's main role is properly scheduling tasks. FreeRTOS has a task control block (TCB)

which is a placeholder for tasks. A TCB is a struct containing task data. TCB contains a pointer to the list owning the task, a task priority integer, and a name for the task.

Tasks have several states: running, ready, delayed, suspended, and blocked/waiting. However, a task does not have a variable which explicitly assigns the task state, but a task's state is known by the list in which it is inserted. There are several types of lists including ready, suspended, delayed , and waiting lists. In regards to a tasks state, if a task is entering the ready state, this task is inserted into a ready list. When a task's state changes from ready to waiting, the task is removed from the ready list and inserted into the waiting list.

Figure 3 models the task states. Tasks enter the ready state after a task is created. The transition from the ready to run state occurs when a task is scheduled. A running task is either suspended, delayed, or waiting as the next transition. A waiting task requests a resource and block until it is available. Tasks may delay for a duration then transit into a ready state at the duration's deadline. Suspended tasks may enter a suspension state indefinitely. This may happen when an interrupt request resources and all other tasks are suspended until the request is finished.

Most task states have an associated list type. The exception is the running state which is marked with the pointer to the running task, `pxCurrentTCB`. The current running task is not a list, but a single item using processor resources. There are several states and list types (see Table I).

Each task has a priority. The default priorities are integers 0 (lowest), 1, 2, 3, and 4 (highest priority). A task's priority allows the scheduler to pick the highest priority

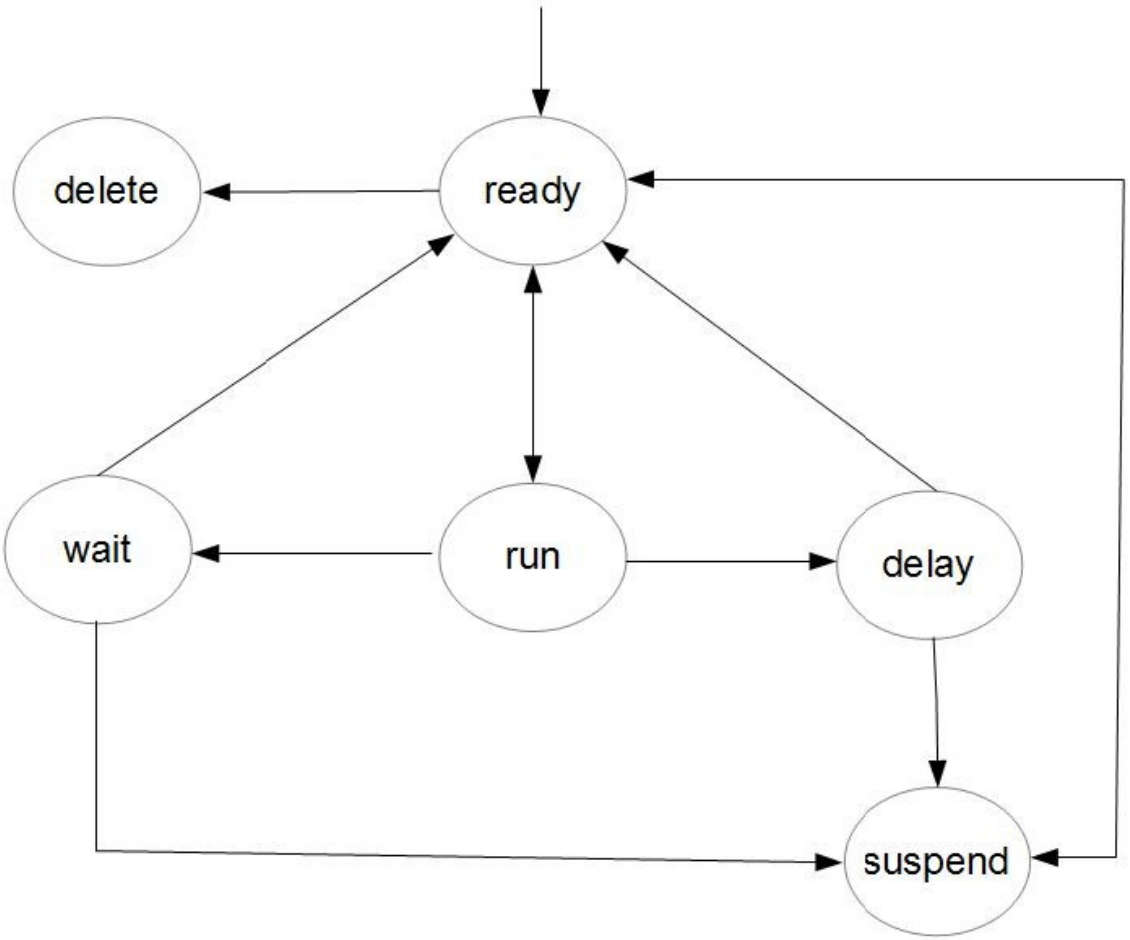

Figure 3: Task States Model

State	**List**	**Description**
Running	none	Current running task
Suspended	SuspendedTaskList	Implemented as an unordered list
Waiting (blocked)	WaitingTaskList	Implemented as a priority queue. Order is determined by task priority. After the waiting state, the highest priority task becomes ready, then the task will run
Delayed	DelayedTaskList	Implemented as a priority queue. Tasks are blocked for a timeout period which is defined by the user. The scheduler checks this task list at every tick to determine if the task timed out. Order is determined by timeout period
Ready	ReadyTasksLists	Newly created tasks are immediately placed in the ready list. Ready lists track tasks currently ready to run. It is implemented as an array of FIFO queues.
Deleted	none	The idle tasks may cleanup deleted tasks

Table I: Tasks states and associated lists

task. For example, a task with the priority of 4 will always run before a task priority of 2. Additionally, the preemptive scheduler allows a task of priority 4 to interrupt a priority 2 task. The non-preemptive scheduler allows a currently priority 3 task to run to completion while a priority 4 task will run next.

A FreeRTOS macro, `config_maxPriorities`, allows setting the number of priorities. Since the default priorities is 5, the priority integers range from 0 to `config_maxPriorities`-1. Also, there are API calls available for manipulating tasks such as accessing a task, changing a task priority, or inserting a task into another list. The list data structures containing the tasks are discussed in the next section.

2.2.3 System Tick

Although the system tick was described in the previous section, aspects of its implementation and abstraction shall be emphasized. The system tick is a software abstraction: hardware implementation details are not accessed by the scheduler. Microcontrollers operate at different tick frequencies. For example, an Atmel AVR Atmega 323 IAR and PIC18 MPLAB tick oscillates at 1000 Hz while an Arm9 STR91X IAR tick operates at 100 Hz. The scheduler has no responsibility to know the tick frequencies of each implementation. However, the scheduler must know when a tick cycle is complete in order to assign a task processor time. The tick allows the scheduler to manage tasks.

The system tick is needed for time slicing. In ordered to schedule tasks correctly, each task receives a slice of processor time. Tasks with the same priority are swapped and receive processor time after each tick cycle. This allows tasks of the same priority to schedule in a round-robin ordering. If there is a single high priority task, it will run to completion. In conclusion, this tick interrupts the processor which allows tasks to

get a fair-share of processor time from time-slicing.

2.2.4 List Data Structures

Lists are used in tasks and scheduling. As mentioned earlier, a task's state is tracked by the list in which it was inserted. As shown in Table I, each list is implemented as a data structure: a suspended list is an unordered list, waiting list is a priority queue, delayed list is a priority queue, and a ready list is an array of FIFO queues. All FreeRTOS lists are implemented as a data structure known as an `xList`. An `xList` is implemented as a linked list containing nodes called `xListItem`. Every task control block is owned by an `xListItem`. The main list types are queues and unordered lists.

Queues There are several uses for queues. FreeRTOS allows tasks communication and synchronization with the use of queues. Also, interrupt service routines (ISRs) use queues for synchronization and communication. The two types of FreeRTOS queues include a FIFO queue and priority queue. Both queues support both blocking and nonblocking inserts and removals. Blocking inserts and removals have a timeout. The user can define a timeout period which the task is alive for a certain amount of time. For example, a delayed task has a timeout period of ten seconds and the task is inserted into a ready list when timeout occurs [10]. A nonblocking insert or removal returns `true` or `false` based on its success or failure.

Priority Queue Waiting and delayed tasks are implemented with a priority queue. This is stated in Table I. Waiting tasks are assigned to a task priority which the lowest priority task is scheduled last and contains the integers equal or closest to zero. The highest priority tasks are scheduled first and assigned with the highest priority integers or closest to `maxPriorities`-1. Delayed tasks have a time-to-wake period. For example, a task is delayed for eight seconds then it is assigned to a ready

list. The task is awake when it is removed from the delayed task list and inserted into the ready list.

FIFO Queue As stated in Table I, ready lists are implemented as FIFO queues. The ready list reference is an array of ready lists in the FreeRTOS kernel. Each ready list index represents the priority level of the task. The scheduler iterates through the ready list (i.e. array of FIFO queues) starting from the highest priority and finishes with the lowest priority list [10].

Unordered List The suspended task list is implemented as an unordered list (see Table I). Task order is not important in this list. An unordered list supports insert and removal of tasks. All tasks that are not currently running may be inserted into the suspended task list. These tasks may be suspended for an unspecified time period [10]. Tasks that are no longer suspended are removed from the suspended task list and inserted into a ready list.

2.2.5 API Calls

Operating systems commonly have an application programming interface (API) [30]. This allows a program to interact with the operating system. Operating system API calls include activities such as manipulating tasks, processes, and files. Table II is a list of scheduler API calls.

API Call	**Description**
vTaskSwitchContext()	selects highest priority task and assigns task to pxCurrentTCB (currently running task)
xTaskCreate()	creates a task
vTaskStartScheduler()	starts the scheduler
vTaskEndScheduler()	terminates the scheduler
ListData	struct containing ready, delayed and suspended/blocked lists
Task	contains a task name and priority. It may contain other members including task data, list data, priority date, etc.)
vTaskDelete()	deletes a running, ready, delayed, or suspended task
vTaskDelayUntil()	delays a task for a given number of clock ticks
vTaskDelay()	delays a task indefinitely
vTaskIncrementTick()	increments the system clock
vTaskPrioritySet()	change the priority of an existing task in the system
vTaskSuspend()	suspends a task
vTaskResume()	updates the suspended and ready lists as required. Resumes a task to the running state
uxTaskPriorityGet()	returns the task priority
xGetCurrentTaskHandle()	return a pointer to the current running task
xTaskGetCurrentTaskHandle()	returns the scheduler state of either executing, suspended, or initialized
uxTaskGetNumberOfTasks()	returns the number of tasks
xTaskGetTickCount()	returns the time elapsed since scheduler initialization
vTaskSuspendAll()	suspends the scheduler
xTaskResumeAll()	resume scheduler

Table II: FreeRTOS Scheduler API

2.3 Object-Oriented Design in FreeRTOS

One goal of this thesis is to specify FreeRTOS's scheduler behavior into Dafny. Along with converting the code base, a formal specification is created. A formal specification with an implementation can be verified for functional correctness in Dafny. Most of the code is converted from the API documentation and code to Dafny directly. However, this is not always straightforward and some code is converted from C to C++ before converting to Dafny. C++ is utilized because it is an object-oriented language and contains C language constructs. This intermediary step helps us translate C to Dafny code. This also allows debugging because Dafny does not have a program debugger that supports execution tracing. The Visual C++ IDE allows for tracing the program execution. Regardless of how the code is translated from C to Dafny, an object-oriented design must be incorporated since Dafny is an object-oriented language.

C is not an object-oriented programming language, but the FreeRTOS scheduler is modeled in an object-oriented language in this project. This is because the Dafny programming language (see Chapter 3) is an object-oriented language. The FreeRTOS scheduler is ported into an object-oriented design in which it is converted into Dafny code.

This section demonstrates the tactics for converting FreeRTOS into an object-oriented design. Some code is converted from C to C++. A class diagram is presented in Figure 8.

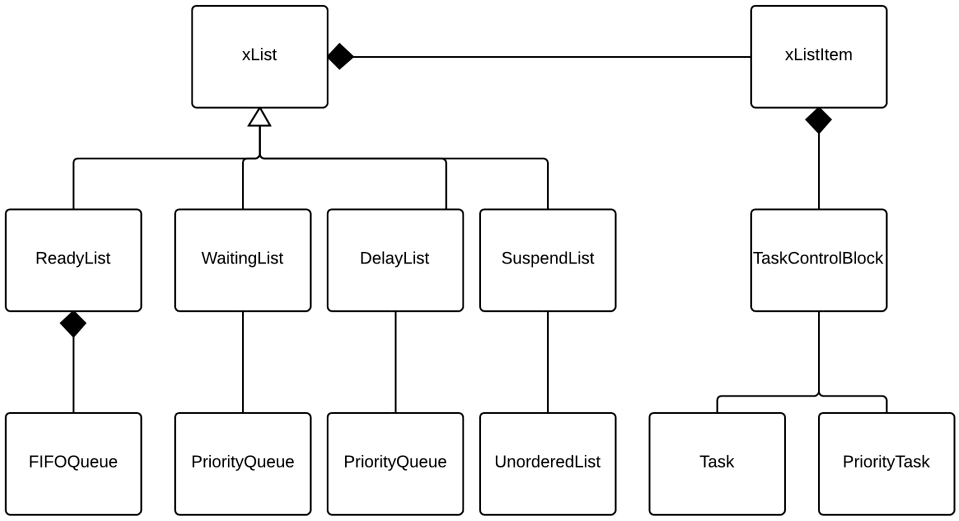

Figure 4: Class diagram of xList

2.3.1 Converting C to an Object-Oriented Design

The API calls and data structures in FreeRTOS are written in C. The code follows modular programming: the program is composed of many different modules. Each module contains variables, functions, or structures. Modules may also contain abstract data structures. The data structures are placeholders for data and the API calls manipulate the data structures. For example, a task is created with the `xTaskCreate()` function and assigned to a task. The task is inserted into a ready list which is represented using a FIFO queue.

C code is converted to an object-oriented design because Dafny (used in later chapters) is an object-oriented programming language. Some of the FreeRTOS code has a struct-and-function idiom: it contains structs for structured data and functions to manipulate data. Structs are placeholders for data and often provide a reference to a data structure. In C, a common idiom includes declaring a struct and using functions

to modify the struct. For example, a queue struct has an insert function which allows appending an item to the queue. The insert function contains two parameters: the queue reference and inserted item. When the insert method is called, the inserted item is appended to the queue via the struct reference to the queue.

This idiom is followed when the C code is converted to an object-oriented language. This struct and function idiom is converted into a class. The struct data members are declared as data members in the class while the functions manipulating the struct are methods owned by the class. Instead of declaring a struct and passing the struct to a function, an object is instantiated from a class and method calls modify the object. The idiom is used to convert a task control block from C to C++. As mentioned earlier, the task control block is a placeholder for a task. A TCB is a struct modified with functions such as `xTaskCreate()`, `uxTaskPriorityGet()`, and `vTaskPrioritySet()` where task creation, priority access, and priority mutation occurs, respectively.

The TCB is converted to a class in Figure 5. Line 2 declares the Task class. There is a default constructor on Line 9 and another constructor which accepts a task name on Line 11. There are two accessors on Line 15 and Line 19 which set and get a task name. Line 24 declares a TaskControlBlock class. This class extends the functionality of the Task class: a task can be declared with a key (i.e. priority) in the TaskControlBlock class.

Let us assume the scheduler's responsibilities includes creating tasks. Figure 6 is a partial class which allows task creation. The `createTask()` method accepts parameters for a name and priority on Line 17. A task control block is created on Line 19 and inserted into a ready list on Line 20. A pointer to the task is returned

```
1  // C++ code for task and task control block
2  class Task {
3
4  private:
5          int _name;
6
7  public:
8
9          Task() { }
10
11         Task(int name) {
12                 _name = name;
13         }
14
15         void setName(int name) {
16                 _name = name;
17         }
18
19         int getName() {
20                 return _name;
21         }
22 };
23
24 class TaskControlBlock {
25 private:
26         int _key;
27         Task _task;
28
29 public:
30         TaskControlBlock() { }
31
32         TaskControlBlock(Task task, int key) {
33                 _task = task;
34                 _key = key;
35         }
36
37         void setTask(Task task) {
38                 _task = task;
39         }
40
41         void setKey(int key) {
42                 _key = key;
43         }
44
45         Task getTask() {
46                 return _task;
47         }
48
49         int getKey() {
50                 return _key;
51         }
52
53 };
```

Figure 5: A task control block (TCB) as a class

at the end of the method (Line 22). Tasks are created by calling the scheduler constructor and invoking the `createTask()` method. Figure 7 shows the creation of a task.

```
1  class Scheduler {
2
3          FIFOQueue* readyList;
4
5          private:
6                  void init_readyList(int maxSize) {
7                          for (int i = 0; i < MAX_PRIORITY; i++)
8                                  readyList[i] = *new FIFOQueue(maxSize);
9                  }
10
11         public:
12                 Scheduler() {
13                         readyList = new FIFOQueue[MAX_PRIORITY];
14                         init_readyList(FIFO_MAX_SIZE);
15                 }
16
17                 TaskControlBlock createTask(int taskName, int taskPrior) {
18                         Task* task = new Task(taskName);
19                         TaskControlBlock* tcb = new TaskControlBlock(taskName, taskPrior);
20                         readyList[taskPrior].enqueue(tcb);
21
22                         return *tcb;
23                 }
24
25                 FIFOQueue getFifoQueue(int priority) {
26                         return readyList[priority];
27                 }
28
29 };
```

Figure 6: Create a task method

```
1
2          Scheduler* sc = new Scheduler();
3          TaskControlBlock tcb = sc->createTask(22222, 1);        // name, priority
4          TaskControlBlock tcb2 = sc->createTask(11111, 0);
5          TaskControlBlock tcb3 = sc->createTask(91231, 1);
6          TaskControlBlock tcb4 = sc->createTask(20812, 1);
7          TaskControlBlock tcb5 = sc->createTask(10899, 0);
8          TaskControlBlock tcb6 = sc->createTask(71987, 1);
```

Figure 7: Instantiate a scheduler and create a task.

Figures 5, 6, and 7 implement task creation in an object-oriented fashion. The scheduler, task and task control block are composed as classes. This is a simple example of converting C code into an object-oriented design and implementation in C++.

2.3.2 Class Diagram of an xList and Scheduler

Figure 4 is a class diagram of the `xList` data structure. The `xList` is the abstraction for ready, delayed, waiting, and suspend lists. Each of the lists inherits from abstract class, `xList`. It has a composition relationship with `xListItem`: an `xList` can have one or many `xListItems`. These are composed of a task control block which this TCB either contains a task or priority task. A ready list is composed of several FIFO queues. Delayed and suspend lists are priority queues and a delayed list is an unordered list.

Also, there is scheduler class. It follows the singleton pattern since there is only one instance of the scheduler. A scheduler has a current task which is a place holder for the running task. The other data members include several lists seen in Figure 4: ready, delayed, waiting, and suspend lists. A ready list is declared as an array. The API calls listed in Table II are methods in the scheduler class.

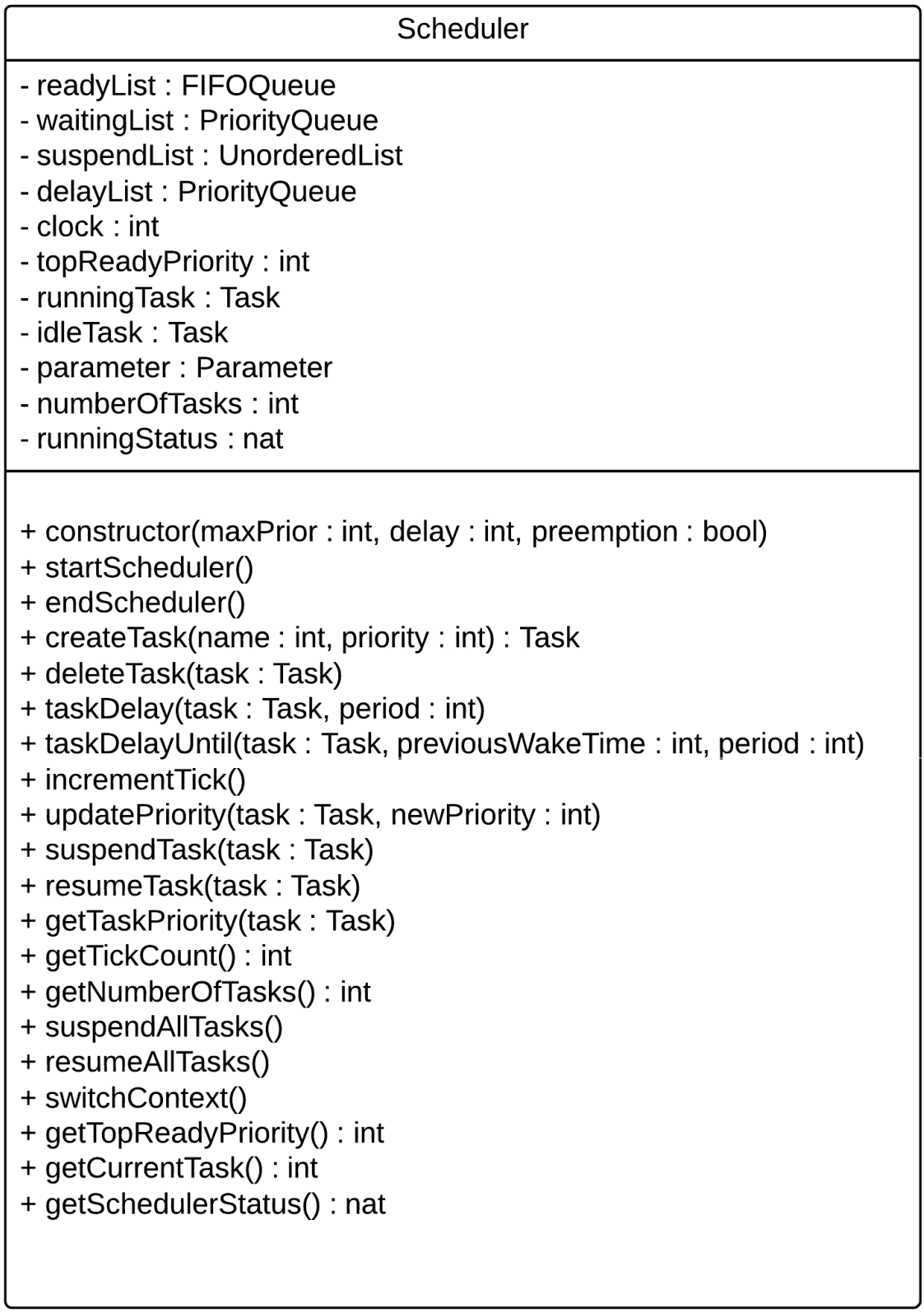

Figure 8: The scheduler class

2.4 Summary

This chapter covered FreeRTOS's architecture including the scheduler's data structures and API. The data structures include the `xList` which is an abstraction for a FIFO queue, priority queue, or unordered list. An `xList` is composed of one or many tasks. The API calls are listed in Table II. The scheduler API calls modify these data structures and tasks. An object-oriented design is presented. FreeRTOS's C code base is translated into an objected-oriented language. A class diagram presents the components of the scheduler and `xList`.

Now that FreeRTOS was discussed, we can consider creating a formal specification of FreeRTOS in Dafny. Dafny's specification and programming language constructs are covered in Chapter 3.

CHAPTER III

Dafny

Until recently, program verification tools included only pencil and paper proofs and assisted theorem provers [19]. These proofs require predefined proof rules and significant mathematics. In addition, it may cost much time and effort. Verifying modern software systems using this method is impractical because the costs are high. Assisted theorem provers reduced time and effort of verification with minimal results because verification proofs were not very automated [19].

Proving program correctness is more automated than its predecessor tools. Modern program verifiers do not require a pencil and paper proofs. Instead, users create annotated programs within a developer environment. Current research efforts have produced a list of benchmarks for program verifiers [31]. Verification tools have grown popular and recent benchmarks guide and measure the tools' limitation. Alone, Microsoft has several program verifiers including SPEC#, HAVOC, `VCC`, Chalice, and Dafny [1].

[1] see http://research.microsoft.com/en-us/projects/boogie/

Dafny met the verification benchmarks with varying success, but improvements were made as a result of benchmark attempts [22]. This project used some features added from the benchmark attempts including termination metrics, sets, and sequences. Despite the benchmark claims, Dafny may require various lemmas to guide the theorem prover. It is not a mature nor perfect tool, but the verification benchmarks have shown Dafny as a useful program verifier since it passed benchmarks of verifying data structures and common math operations [22]. There is an active and supportive community built around Dafny. This tool has regular updates, improvements, and builds available to the public. Dafny is a continuously improving verification tool.

In this chapter, the first section covers basic concepts behind Dafny. Section 2 provides a high-level explanation of verifying programs in Dafny. The role of an intermediary language, theorem prover, and verification conditions are shown. Section 3 shows Dafny's programming language and the language constructs used in this project. This language allows the user to create an implementation which is proven correct given a specification. Section 4 explains the specification constructs used in Dafny.

3.1 What is Dafny?

Microsoft Dafny is a program verification tool which includes a programming language and specification constructs [20]. Users create and verify specifications and implementations. After these are created, the verifier proves correctness of the implementation in regards to the specification.

An advantage to Dafny is a relationship between the specification and program's code because its users must create an implementation and specification for verification. In comparison, other software engineering formal methods tools such as model checkers are disconnected from the software's implementation since the model is a representation not easily ported to programming code [13]. A Dafny program needs an implementation and specification for verification, but the verification may require much effort to verify implementations and verification. Also, translation to an implementation language may require effort. Programmers may find it less cumbersome to port a Dafny program to an implementation language such as C++ rather then translating a model to program code.

Dafny is an object-based language and does not support sub-typing or inheritance [22], but the FreeRTOS specification can utilize an object-oriented design. It can verify object-oriented programs which may seem an odd choice for verifying modules from FreeRTOS which are written in C. Mapping object-oriented design to a non-object-oriented language is feasbile [27] [2]. Dafny encourages using best-practice programming styles and following object-oriented design is our approach to following best practices. Our goal in this project is verifying FreeRTOS's hardware-independent modules and Dafny is the verifier tool used in this project to achieve this goal.

3.2 The Toolset and Architecture

A Dafny program includes specifications and implementations. Users create methods which have precondition and postcondition specifications. Classes may have an invariant which specifies aspects of the class that do not change regardless of the method called in the class. After the Dafny program specification and implementa-

[2]Chapter 16 in *Object-Oriented Modeling and Design* shows the transistion from an object-oriented design to a programming language that is not object-oriented

tion is created, the program verification process occurs.

Dafny contains several components involved in the verification process shown in Figure 9. Users write specifications and code in a Dafny program. When a program is verified, the specification constructs and program code are sent to the program verifier. The Dafny program is translated into an intermediate verification language known as Boogie. Boogie creates the verification conditions (VCs). Verification conditions are generated from the translated Boogie code. The verification conditions are assertions used to prove if the program is correct. These verification conditions are given to the Z3 SMT solver and either the program is correct or an error model is generated. If all the verification conditions are true, the program is proven correct in respect to the specfication. An error model is generated for incorrect programs. An error model is a counterexample stating a verification condition was not proved. The Boogie Verification Debugger (BVD) can display and trace the error model that was generated. As of Dafny v1.7, the Dafny Visual Studio add-in displays the BVD and allows for interacting with the error model.

Z3 is an satisfiability modulo theory (SMT) solver in which it is used as Dafny's reasoning engine. Dafny is considered an auto-active verifier [18]: users will interact with the proofs, but there is automation. This SMT solver contains a collection of theories, theorems, and proof rules. Z3 can solve declarative calculations, inductive and co-inductive proofs. In Dafny, users can use the specification and programming constructs to create the proof. The SMT solver, Z3, contains facts on specification constructs. The SMT solver's components allows Z3 to solve proofs.

The rectangles in Figure 9 are Dafny's components while the circle items are inputs or outputs of each component. Each component is standalone software. For

example, a user can prove theorems without using Boogie or Dafny. Boogie can verify programs as a standalone tool, but this is not recommended because it is an intermediate verification language. Several program verifiers were built using Boogie as an intermediary language such as `VCC` [3]. Boogie also contains the verification condition (VC) generator. The VC generator is shown separately in Figure 9 to emphasize verification conditions are generated after the Boogie code translation and before Z3 proves the VCs.

Microsoft offers a web interface for Z3 [4], Dafny [5], and Boogie [6]. Dafny's web interface is good for small programs, but cumbersome for larger programs. Web interfaces for program verification tools are mostly for demonstration. The Dafny Language Service add-in for Visual Studio is recommended to write programs.

[3] see `http://research.microsoft.com/en-us/projects/boogie/`
[4] see `http://research.microsoft.com/en-us/um/redmond/projects/z3/old/`
[5] see `http://research.microsoft.com/en-us/projects/dafny/`
[6] see `http://research.microsoft.com/en-us/projects/boogie/`

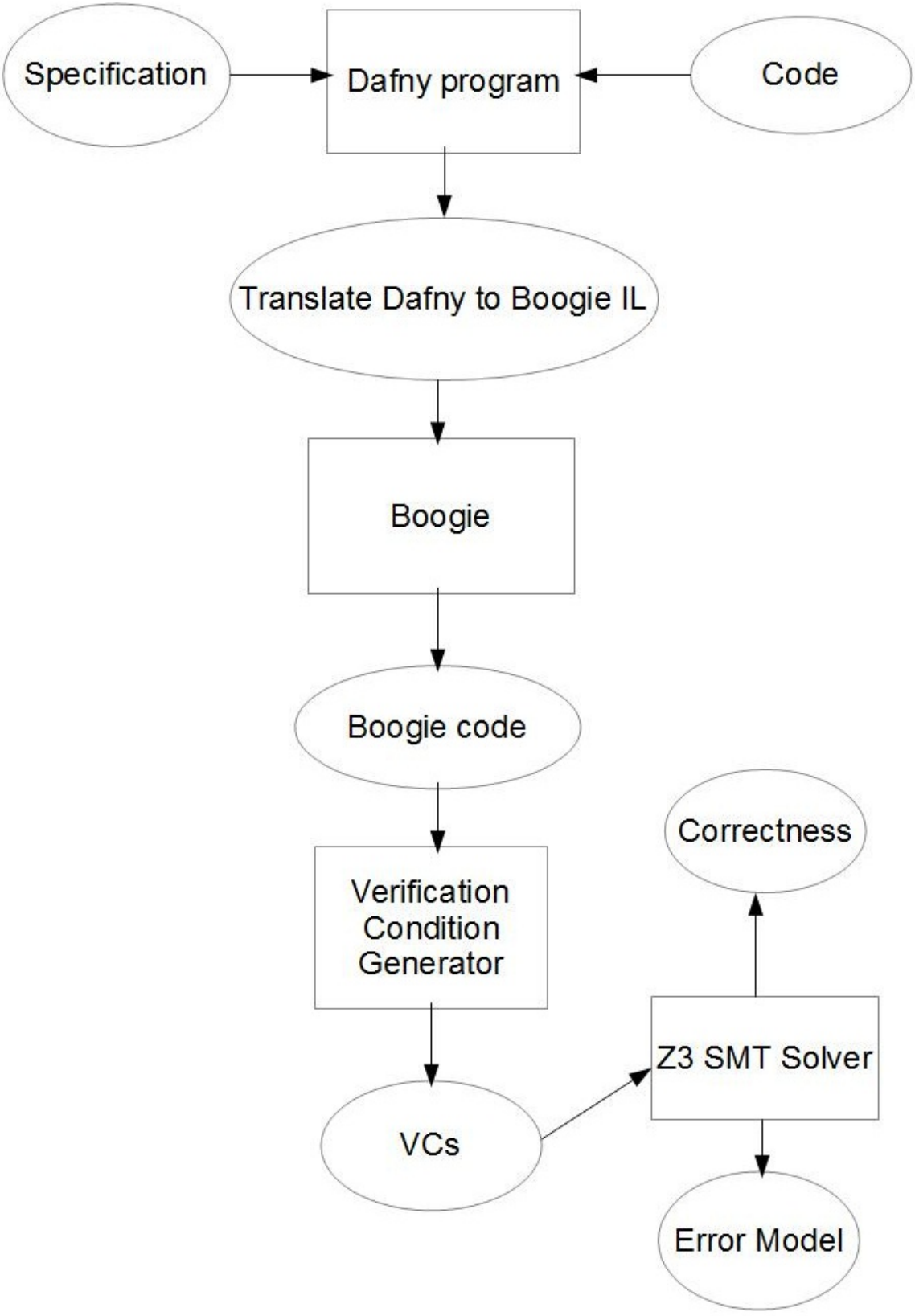

Figure 9: Dafny Verification Process

3.3 The Dafny Programming Language

Dafny can model an object-oriented program [22]. Classes are created and instantiated as objects and classes may contain data members, constructors, accessors, and methods in an object-based language. However, Dafny does not support inheritance, but composition relationships are used to model inheritance if necessary.

Tricks such as using composition to model inheritance is needed in the Dafny programming language. It is a minimal programming language and does not have many language constructs. It does contain weak or strong typed variables, branching, loops, methods, dynamic data types, and classes common in many modern programming languages [20]. There are also ghost types, functions, invariants, lemmas, assertions, and assumptions used for specifications [20].

3.3.1 Assignment, Equality, and Data Types

`:=` is used for assignment. Dafny allows for both strong and weak typed variables. Strong type variables state the data type explicitly and weak type variables state the data type implicitly. Assigning an integer in a strong and weak typed manner is as follows in Figure 11.

The `setInteger( )` method demonstrates variable assignment. A method declaraction must contain the `method` keyword and name. Optionally, `method` may contain parameters, return value, body, post conditions, and preconditions. The following examples in Figure 12 contains legal method declarations:

Methods do not always have a body when writing specifications and may only contain preconditions and postconditions. A Dafny user may attempt to verify and

```
1  // Java example
2  public class A {
3          public A ( ) { }
4
5          public void printString(String s) {
6                  System.out.println(s);
7          }
8  }
9
10 // inheritance
11 public class B extends A {
12
13         public B ( )  { }
14
15         public void printString(String s) {
16                  super.printString(s);
17         }
18 }
19
20 // composition
21 public class B {
22         private A a;
23
24         public B ( ) {
25                 this.a = new A( );
26         }
27
28         public void printString(String s) {
29                 this.a.printString(s);
30         }
31 }
```

Figure 10: Java inheritance and composition

```
1 // assign two integers a value
2 method setInteger( ) {
3         var i : int := 5;       // strong typed
4         var j  := 6;            // weak typed
5 }
```

Figure 11: Integers in Dafny

```
1 // method follows the form of method keyword,
2 // method name, return value, postcondition, and body
3 method getInteger() returns (i : int)
4         ensures i >= 0; // i must be positive
5 {
6         var i := 5;
7         return i;
8 }
9
10 // method follows the for method keyword, method name,
11 // parameters, return value, precondition, postcondition, and body
12 method setIntegerToFive(i : integer) returns ( j : int )
13         requires i >= 0;
14         ensures i == 5;
15 {
16         var j := i
17         return j;
18 }
19
20 // method with no body, but only specifications
21 method getPositiveInteger( ) returns ( i : int )
22         ensures i >= 0;
```

Figure 12: Methods in Dafny

test the specification before writing an implementation. This is further described in Section 3.3.2.

The `==` symbol compares to values for equality and `!=` is the inequality symbol. Comparing two integers will compare two integer values and does not compare the references. The example (below) shows comparing integers `i == 5` which compares a variable to a literal value for equality and `i != j` for inequality. Each `i`, `j`, and 5 are compared by value.

Primitive and generic data types are supported. Primitive types include integers, booleans, and natural numbers. Boolean types contain either `true` or `false`. Integers include signed integers where `N` is the maximum integer such that $-N \leq i \leq N - 1$.

```
1 method compareIntegers( ) {
2         var i : int := 5;
3         var j  := 6;
4
5         assert i == 5;  // i == 5 evaluates as true
6         assert i != j;    // i != j compares two values, but not two references
7 }
```

Figure 13: Comparing values in Dafny

Natural numbers are unsigned integers where `N` is the maximum integer such that $0 \leq i \leq N - 1$. The method in Figure 14 shows valid primitive types:

```
1 method showDataTypes( ) {
2         var i : int := -55;     // integer
3         var j : int := 55;      // integer
4         var k : nat := 99;      // natural number
5 }
```

Figure 14: Signed and unsigned integers in Dafny

Dafny also allows for generic types. A class must be created to use generic types. The class created will contain a constructor, data members, and methods. Suppose class `A` is created and this class will accept the type `Data`. When `A` is instantiated, an integer type is assigned to the generic data type, `Data`, in the main method of the example (see Figure 15).

3.3.2 Branching, Loops, and Arrays

Branching and loops are supported by the `if..else` statement and `while` loop, respectively. Loops commonly iterate through arrays: the usual ordered, fixed memory sized, and list-styled construct. The declaration and initialization of an array and its respective elements is as follows in Figure 16.

An `if..else` statement requires brackets in the body. `else..if` is supported. An example of a branching statement shown in Figure 17.

```
class A<Data> {
  var k : Data;

  constructor(s : Data)
    modifies this;
  {
    k := s;
  }

  method getK() returns (k : Data)
  {
    return k;
  }
}

method Main() {
  var i : int := -5;
  var a := new A<int>(i);
  var b := a.getK();
  //e := new A(i);
}
```

Figure 15: Classes and generic types in Dafny

```
        var a  := new int[10];
        a[0] := 5;
        a[1] := 3;
        a[2] := 89;
        a[3] := -2;
        a[4] := 9;
```

Figure 16: Initializing an array and its values in Dafny

`while` loops contain a loop condition and body. Loops often contain specifications for termination metrics and loop invariants, but this is discussed in the next section. Here is an example which finds the first instance of a negative number in the array `a` in Figure 18.

```
1         bool success := false;
2
3         if ( a.Length > 0) {
4                 success := true;
5         }
6         else if ( a.Length == 0) {
7                 success := false;
8         }
9         else {
10                success := false;
11        }
```

Figure 17: If statement

```
1  method findNegative(a : array<int>) returns (negativeInt : int)
2         // specifications
3         requires a != null;
4         requires a.Length > 0;
5         ensures  negativeInt < 0 ==>
6          exists i :: 0 <= i < a.Length ==> negativeInt == a[i];
7  {
8         var i := 0;
9
10         while ( 0 <= i < a.Length)     // notice the loop condition,
11                                                //0 <= i < a.Length
12                invariant 0 <= i <= a.Length;   // specification
13                decreases a.Length - i;         // specification
14         {
15                if ( a[i] < 0 || i == a.Length) {
16                        break;
17                 }
18                else {
19                        i := i + 1;
20                }
21          }
22
23          if (i == a.Length) {
24                 i := i - 1;
25          }
26
27          negativeInt := a[i];  // could also be return a[ i ];
28 }
```

Figure 18: Find the first negative number in the array

3.4 Specifications in Dafny

Dafny contain annotations for specifying programs. Each program does not always declare a method implementation, but each method is annotated with a specification. The code is verified in respect to the specification in which the method's preconditions and postconditions are used to prove correctness. This section discusses the annotations used in Dafny.

3.4.1 Postconditions and Preconditions

The most common specification constructs in Dafny are method preconditions and postconditions. Method preconditions state conditions which are true before the method is called and method postconditions state true conditions after the method is invoked. A good example is the `findNegative( )` method from the previous section, but it is shown in Figure 19.

```
1
2 method findNegative(a : array<int>) returns (negativeInt : int)
3         requires a != null;                      // precondition
4         requires a.Length > 0;                   // precondition
5         ensures  negativeInt < 0 ==>
6                 exists i :: 0 <= i < a.Length ==>
7                         negativeInt == a[i];     // postcondition
```

Figure 19: Finding negative method specification

Regardless of the method's implementation [7], the specification shows the method's behavior. `findNegative( )` requires an initialized array (i.e. an array that is not null) and an array with a length greater then zero. There is some reasoning for the preconditions. This method needs an array which contains elements for searching. First, the array requires initializations because an uninitialized array does not have accessible elements. In most programming languages, accessing an uninitialized array such as `a[0] := 4` will cause a crash (i.e. either a thrown exception or compilation

[7] The emphasis is on the specification of this method and not the method body.

error). Second, an array length of more then zero is required because searching an array with no elements is quite useless. One or more elements in the array allows for searching for a negative number.

The postcondition is one statement which is an implication followed by an existential quantifier (discussed more in the next sections). This postcondition states the main behavior of this method: if the negative integer is found, then there is a bounded i where the negative integer is the same as the *ith* element in the array.

3.4.2 Invariants and Termination

An invariant is defined as an attribute remaining constant and unchanged across several states [8]. Dafny requires loop invariant annotations in `while` loops. Consider the loop in the finding negative method in Figure 20.

```
1  var i := 0;
2
3  while ( 0 <= i < a.Length)              // notice the loop condition,
4                                          // 0 <= i < a.Length
5          invariant 0 <= i <= a.Length;   // loop invariant annotation
6          decreases a.Length - i;                 // termination metric
7  {
8          if ( a[i] < 0 || i == a.Length) {
9                  break;
10         }
11         else {
12                 i := i + 1;
13         }
14 }
```

Figure 20: A while loop's invariant

At a first glance, the loop condition is similar to the invariant. The only difference between the loop condition and invariant is `0 == i == a.Length` is valid in the invariant at the end of the loop, but not the loop condition. i is assigned to zero

[8] This is my definition.

before the loop and after the loop, i might equal the array length, a. This reasoning helps, but this does not show the reason the invariant is required.

The loop invariant is concerned with three states of a loop:

1. The state before the loop (i == 0)
2. The state during the loop ($0 \leq i \leq a.Length$)
3. The state after the loop terminates ($i \leq a.Length$)

The variable, i, is assigned to zero before the loop starts. Therefore, $i == 0$ is true and it satisfies the loop invariant, $0 \leq i \leq a.Length$. The loop is entered when $0 \leq i < a.Length$ based on the loop condition. Inside the loop, i is incremented until $a[i]$ is negative or $i == a.Length$ and the loop invariant is still satisfied. The `while` loop will exit if the loop condition is false. Once the loop exits and $a[i]$ is negative, then the $0 \leq i \leq a.Length$ is satisfied since i is located in the array. Alternatively, $i == a.Length$ satisfies the invariant too.

This reasoning is showing the possible states reached before, during, and after the `while` loop's iterations. Dafny cannot handle proving an infinite amount of reachable states in a loop [20], but it can prove a finite amount of reachable states with the `while` loop. The loop invariant provides Dafny with a finite model of reachable states. As a result, it is verified.

Dafny also needs a termination metric for loops. Notice in our previous reasoning consisted of statements such as "the loop will exit if the condition is false." The `decreases` clause is the termination metric in which it specifies the loop exits at a specific condition. When `a.Length`$-i == 0$ is true, the loop will terminate. A termination metric is needed since Dafny needs to prove the loop will exit. Dafny

attempts to guess the termination metric if the user does not annotate it [22]. As useful as this may seem, it is not as challenging as creating a loop invariant. The termination metric contains a stating when the loop terminates, but loop invariants must show the unchanging state of a loop. An invariant table is often created to find the unchanging attributes of a loop.

3.4.3 Predicates and Functions

Predicates and functions are specification constructs which contain one statement and return a value. Predicates return a boolean value while functions return any data type. These functions represent mathematical functions [20] An example of a function is using an `isEmpty()` function for specifying an array-implemented stack. As stack is empty when $items == 0$.

```
1
2 function isEmpty() : bool       // or, this could be ``predicate isEmpty()"
3         reads this;
4 {
5         items == 0
6 }
```

Figure 21: A function checking if the stack is empty

A precondition of a stack's `pop()` method requires the stack is not empty. The `isEmpty()` function is used in Figure 22.

```
1
2 method pop() returns (d : Data)
3         requires !isEmpty();
4         ensures Contents == old(Contents)[0..old(items)-1] &&
5                 d == old(Contents)[old(items)-1];
6 {
7         items := items - 1;
8         d := a[items];
9         Contents := Contents[0..items];
10 }
```

Figure 22: A pop() method in Dafny

The `pop()` method's precondition requires the stack is not empty. Instead of

using `isEmpty()`, the precondition could be written as *items* == 0. Functions are used because readability is improved and the specification is portable. Any method can call `isEmpty()` in any specification annotation such as preconditions or postconditions.

`isEmpty()` could be declared as a predicate since it returns a boolean value. However, predicates are often used in class invariant specifications.

3.4.4 Specifying Classes

Dafny must have annotations for specifying classes for verifying object-oriented programs. Classes commonly contain constructors, data members, and methods, but a class invariant and representation are also included in Dafny. Consider a partial stack class in Figure 23.

```
1  class {:autocontracts} Stack<Data> {
2          ghost var Contents : seq<Data>; // representation
3          var a : array<Data>;
4          var items : int;
5          var maxSize : int;
6
7          predicate Valid          // class invariant
8                  reads this;
9                  reads a;
10         {
11                 a != null &&
12                 0 <= items <= |Contents| < maxSize+1 <= a.Length &&
13                 Contents == a[0..items]
14         }
15 /* .. functions and methods .. */
16 }
```

Figure 23: The beginning of a stack class

The stack needs a representation that is decoupled from the implementation. Sequence `Contents` represents the stack. Sequences model ordered data structures in Dafny. `Contents` is also declared a ghost variable because it is called only in specification annotations and not used in the implementation. The sequence contains append and cardinality operations needed in our stack specification. This is shown

in the `push()` and `pop()` specifications.

Predicate `Valid` is Dafny's class invariant [20]. Since the `:autocontracts` keyword is called in the class, `Valid` is automatically a postcondition for constructors and a precondition and postcondition for all methods in the stack class. The class invariant contains bounds-checking and a mapping of the implementation to the abstract specification. `a != null` states the array is initialized since methods cannot modify an uninitialized array. $0 \leq items \leq| Contents |< maxSize + 1 \leq a.Length$ states the valid bounds of the stack. `Contents == a[0..items]` maps the array implementation to the abstract specification. `a[0..items]` is Dafny's slicing notation which allows assigning elements an array to a sequence. This slicing notation allows the abstract specification and array mapping needed for this specification.

```
1  constructor(size : int)
2          modifies this;
3          requires 0 < size;
4          ensures maxSize == size;
5          ensures isEmpty();
6          ensures maxSize > 0;
7          ensures a.Length > 0;
8  {
9          maxSize := size;
10         a := new Data[maxSize+1];
11         items := 0;
12         Contents := a[0..items];        // initialize Contents sequence
13 }
```

Figure 24: Stack constructor

The constructor (Figure 24) initializes the class, ghost variables, and data members. The array implementation is initialized to a discrete size and initialized. This follows the usual idiom for initializing classes in object-oriented programming. However, the ghost variable, `Contents`, is initialized in the constructor. The `Valid` predicate is implicitly a postcondition since `:autocontracts` was declared. The constructor ensures `Contents` is initialized with the array implementation values. As a result, the entire array is assigned to `Contents` using Dafny's slicing notation. Also,

slicing may remove items from the sequence as shown in the `pop()` method.

```
1  method push(d : Data)
2          requires !isFull;
3          ensures Contents == old(Contents) + [d];
4          ensures maxSize == old(maxSize);
5  {
6          a[items] := d;
7          items := items + 1;
8          Contents := Contents + [d];
9  }
10
11 method pop() returns (d : Data)
12         requires !isEmpty();
13         ensures Contents == old(Contents)[0..old(items)-1] &&
14                 d == old(Contents)[old(items)-1];
15         ensures maxSize == old(maxSize);
16 {
17         items := items - 1;
18         d := a[items];
19         Contents := Contents[0..items];
20 }
```

Figure 25: Stack operations

As shown in Figure 25, `push()` and `pop()` are the usual operations for a stack: append an item to the top of the stack and remove an item from the top of the stack, respectively. The `push()` method appends a d item to the `Contents` sequence using the old sequence (i.e. before the method call) with the appended new item, d. This is stated in the postcondition, `Contents == old(Contents) + [d]`. The `pop()` method uses the slicing notation to remove an item from the stack. Dafny does not have a "unappend" sequence operation. Therefore, the slicing notation is used to remove sequence items. The postcondition `Contents == old(Contents)[0..old(items)-1]` states the new sequence is all of the old sequence items except for the last item, `old(items)-1`. The last item is the removed item.

The following figure, Figure 26 shows the entire stack. A main method is declared to provides a demonstration that the stack is usable in Figure 27.

```
class {:autocontracts} Stack<Data> {
        ghost var Contents : seq<Data>;
        var a : array<Data>;
        var items : int;
        var maxSize : int;

        predicate Valid
                reads this;
                reads a;
        {
                a != null &&
                0 <= items <= |Contents| < maxSize+1 <= a.Length &&
                Contents == a[0..items]
        }

        constructor(size : int)
                modifies this;
                requires 0 < size;
                ensures maxSize == size;
                ensures isEmpty();
                ensures maxSize > 0;
                ensures a.Length > 0;
        {
                maxSize := size;
                a := new Data[maxSize+1];
                items := 0;
                Contents := a[0..items];
        }

        function isEmpty() : bool
                reads this;
        {
                items == 0
        }

        predicate isFull
                reads this;
        {
                items == maxSize
        }

        method push(d : Data)
                requires !isFull;
                ensures Contents == old(Contents) + [d];
                ensures maxSize == old(maxSize);
        {
                a[items] := d;
                items := items + 1;
                Contents := Contents + [d];
        }

        method pop() returns (d : Data)
                //requires items >= 1;
                requires !isEmpty();
                ensures Contents == old(Contents)[0..old(items)-1] &&
                        d == old(Contents)[old(items)-1];
                ensures maxSize == old(maxSize);
        {
                items := items - 1;
                d := a[items];
                Contents := Contents[0..items];
        }
}
```

Figure 26: A verified stack

```
1  method main() {
2
3          var st : Stack<int> := new Stack<int>(10);
4
5          st.push(20);
6
7          var temp := st.pop();
8          st.push(temp);
9
10         st.push(40);
11         st.push(60);
12
13         assert 40 in st.Contents;
14         assert 20 in st.Contents;
15         assert 60 in st.Contents;
16
17         var item := st.pop();
18         assert item == 60;
19
20         var item2 := st.pop();
21         assert item2 == 40;
22
23         var item3 := st.pop();
24         assert item3 == 20;
25
26         assert item == 60;
27         assert item2 == 40;
28         assert item3 == 20;
29
30 }
```

Figure 27: Main method which tests the stack

3.5 Summary

This chapter presented the Dafny programming language and specification constructs. Verifying a stack data structure and "finding negative" algorithm was also presented. These verifications were discussed in order to show the possible applications to verifying FreeRTOS's scheduler. Verifying the scheduler requires verifying several data structures and API calls. FreeRTOS's data structures are verified in Chapter 4 and API calls are verified in Chapter 5.

CHAPTER IV

Verifying the xList in Dafny

An operating system scheduler manages tasks by guaranteeing each task receives a fair-share of processor time. Task management in FreeRTOS includes list data structures. The underlying scheduler's data structure is known as `xList`. This data structure is implemented as a doubly-linked list in FreeRTOS. An is composed of one or many `xList` items consisting of a task. Also, the is an abstraction for ready, delayed, waiting, and suspended lists. These lists are special cases of `xList`. The is vital to scheduling correctly. The scheduler's implementation must guarantee tasks are scheduled correctly in its design. Therefore, the scheduler's data structures must allow for proper scheduling to occur. The data structure is the foundation of FreeRTOS's scheduler.

This chapter will show the verification of FreeRTOS's using Dafny. The first section provides a high-level explanation of eliciting specifications. The trailing sections explain the task class, FIFO queue, priority queue, and unordered list. The specification and refined code is shown. Finally, the last section concludes the data structure

verification.

4.1 Purpose

The scheduler uses `xList` for managing tasks. Tasks can have several states (see Chapter 2) which are represented as data structures. Tasks change state from being inserted or removed from ready, suspended, delayed, and waiting lists. An is either a ready, waiting, delayed, or suspend list. A ready list is composed of several FIFO queues and a suspend list is an unordered list. Additionally, delay and waiting (blocked) lists are priority queues.

The verification is a starting point for verifying the FreeRTOS scheduler. The scheduler's foundation is composed of underlying data structures. The must allow operations including insertion and removal of tasks. Creating a Dafny specification and implementation for the are the materials used for showing functional correctness of `xList`. The purpose of verifying the is to show its behavior is functionally correct.

In showing the `xList`'s behavior is correct, functional correctness is defined as proving the data structure is correct in respect to the Dafny specification. If the data structures are verified as correct, the scheduler API's verification comes next (see Chapter 5). Proving the data structure is the foundation for verifying the functional correctness of the FreeRTOS scheduler.

4.1.1 Z Schema to Dafny Specification

The Indian Institute of Science created a `Z` model of the `xList` [8] and scheduler API [7]. These Dafny specifications are based on the `Z` schema provided by IISc. The specification constructs in `Z` are utilized in Dafny because both are based on set theory

and formal logic. IISc's `Z` schema are very detailed and thorough.

Many concepts from the `Z` model [8] are used in the Dafny specification. There is a task schema in the `Z` model [8] because a scheduler manages tasks. The task schema includes a task key (i.e. priority) and name. The Dafny specification uses this schema and declares a task class with `name` and `key` as data members. However, `name` is declared an integer in Dafny because string data types do not exist. The data structures follow the `Z` model also. The `xList`'s underlying data structures including a FIFO queue and priority queue are modeled as sequences. Unordered list is a set in the `Z` model [8], but it is a sequence in Dafny. Both queues have insertion and removal operations traditionally found in lists and queues. Additionally, the `Z` model contained a random-access `remove()` operation in all data structures [8]. This is convenient for moving tasks to different lists in the scheduler. The Dafny specification includes a `remove()` method and it is called frequently in the scheduler. This method is adapted to input a single parameter of data type `Task` rather than the task and key used in the `Z` model [8].

Sequences are utilized in both the `Z` model and Dafny specification. The concepts behind the operations are similar, but the notation is different. `Z` contains several symbols for sequence and set concatenation, but it is the + symbol in Dafny. This allows for appending elements, but random-access insertion is also possible. The sequence is sliced into two parts and the new element is concatenated between the two pieces. Removing elements utilizes slicing a sequence into two pieces and concatenating the two parts. Sequences support accessing or removing the first element (i.e. the head). Sequence element `seq[0]` references the head and `seq[1..]` retrieves the tail. `Z` supports those operations with `head()` and `tail()`. In summary, Dafny and `Z` support similar sequence operations.

$$
\begin{array}{|l}
\hline
[ListData] \\
PQ : \text{seq } TASK \times \mathbb{N} \\
FIFO : \text{seq}(\text{iseq } TASK) \\
unordered : \mathbb{P}\ TASK \\
\hline
\#PQ \leq BOUND \\
\#FIFO \leq BOUND \\
\forall\, i : \mathbb{N} \bullet i \in \text{dom } FIFO \Rightarrow \#FIFO(i) \leq BOUND \\
\#unordered \leq BOUND \\
\text{dom } FIFO = \{1 \,..\, MAXPRIO\} \\
\forall\, i : \mathbb{N} \bullet i \in \text{dom } PQ \Rightarrow PQ(i).2 \leq BOUND \\
\forall\, i, j : \text{dom } PQ \bullet i \neq j \Rightarrow PQ(i).1 \neq PQ(j).1 \\
\forall\, i, j : \text{dom } PQ \bullet i < j \Rightarrow PQ(i).2 \leq PQ(j).2 \\
\hline
\end{array}
$$

Figure 28: `ListData` schema from IISc

The `Z` model contains a schema `ListData` modeling the `xList` [8]. The schema is shown in Figure 28 in which priority queue, FIFO queue, and unordered list are referenced respectively as `PQ`, `FIFO`, and `unordered`. The Dafny specification contains both queues modeled as sequence, but the unordered list is a sequence. In contrast, IISc's unordered list is a set of tasks. The Dafny specification does not declare a `ListData` class. In an object-oriented language, priority queue, FIFO queue, and unordered list would contain an `xList` interface or abstract class and follow polymorphism. Dafny does not support inheritance, but the `xList` could declare queues and lists as data members. Dafny does not handle several levels classes well due to framing issues. As a result, priority queue, FIFO queue, and unordered list will retain the respective data type when declared as a data member.

The Dafny specification is influenced by the `Z` model, but the annotations use different characters, operations, and operators. The `Z` model utilizes a strict mathematical notation. Dafny contains both mathematical and programming constructs. This specification is a based on the `Z` model, but the API documentation is referenced

also.

4.1.2 API Documentation to Dafny Specification

The API documentation is a sufficient high-level reference, but it is written in natural language and subject to ambiguity. Formal specifications may remedy the ambiguities of natural language, but the API documentation is utilized as a reference. This documentation contains comments for public functions and shows examples of how to use the functions. The `xList` and task control block (TCB) structures were referenced. The TCB's members were translated into data members for class `Task`. `xList`'s behavior was described sufficiently in the API documentations, but `Z` model is referenced more for verifying the data structures. However, the scheduler class (Chapter 5) references the API documentation more extensively.

4.2 Task Class

Tasks are an essential component to an operating system scheduler. Each task has a name or priority. The task priority and name follow setter and getter patterns common in object-oriented design. Task priorities commonly change in a scheduler. Therefore, a task becomes a class in Dafny because it is the main object that is managed by the scheduler.

The next sections contain the task class specification and refinement into a Dafny implementation. The specification is contained in a module and refined into code. The task class in the task module is refined into an implementation after the specification is created.

```
1  module TaskModule {
2          class Task {
3                  var name : int;
4                  var key : int;
5                  ghost var g_name : int;
6                  ghost var g_key : int;
7
8
9
10                 predicate Valid()
11                         reads this;
12                 {
13
14                         name == g_name &&
15                         name >= 0 &&
16                         g_name >= 0 &&
17                         g_key == key &&
18                         key >= 0 &&
19                         g_key >= 0
20                 }
21
22                 constructor(newName: int, newKey : int)
23                         requires newName >= 0;
24                         requires newKey >= 0;
25                         modifies this;
26                         ensures g_name == newName;
27                         ensures g_key == newKey;
28                         ensures name == g_name;
29                         ensures key == g_key;
30                         ensures Valid;
31
32
33         }       // end class Task
34 }
```

Figure 29: Task Specification Module

4.2.1 Specification

The Figure 29 contains the code for the task specification module. The first line in the task specification contains the module declaration. A Dafny module allows specification refinement into an implementation. This module is refined later to include an implementation that uses this specification. The second line contains the class name, `Task`. Next, `name` and `key` are declared and used as setters and getters in Dafny (Lines 3-4). `g_name` and `g_key` are ghost variables used only in a specification context (Lines 5-6). The key is the priority of the task. Notice `name` is an integer and not a string. This is because Java-like strings do not exist in Dafny.

The predicate, `Valid` (Line 10), is used as the class invariant. This is a common specification pattern in Dafny classes. `Valid` is a postcondition for a constructor and a precondition and postcondition for all methods in a class. This allows method calls not to change the data members of the existing class [1]. The next line, `reads this`, states the memory locations accessible by the predicate. This allows access to the data members in the `Valid` predicate's body. The `Valid` predicate body (Lines 12 to 20) maps the implementation to the ghost variables such as `name` == `g_name`, `g_key` == `key`. The predicate body also contains bounds-checking because these integers are natural numbers and should not contain negative integers[2].

The constructor is declared (Line 22). `newName` and `newKey` set the new values for the task's name and key. These values require natural numbers (Lines 23 and 24). `modifies this` allows mutating an instance of this class (Line 25). In Dafny, `this` refers to this specific class. Lines 26-29 are postconditions ensuring `name` and

[1] Dafny will assume values change in between method calls if Valid is not used as an invariant. This is because the SMT solver attempts verification with arbitrary values, unless stated in the specification.

[2] Dafny contains a natural number, `nat` data type, but there is a known bug which sequences cannot be referenced by natural numbers, but signed integers, `int`, work perfectly. See extended Dafny tutorial at `http://rise4fun.com/Dafny/tutorial/Sequences`

```
1  module TaskImplementation refines TaskModule {
2          class Task {
3
4                  constructor(newName: int, newKey : int)
5                  {
6                          name := newName;
7                          g_name := name;
8                          key := newKey;
9                          g_key := key;
10                         Repr := {this};
11                 }
12
13                 function method getName() : int
14                         reads this;
15                 {
16                         name
17                 }
18
19                 function method getKey() : int
20                         reads this;
21                 {
22                         key
23                 }
24
25
26         }       // end class Task
27
28 }
```

Figure 30: Task Specification Module

`key` values are assigned to the respective ghost variables and implementation values. `ensures Valid` is our class invariant. Data members will not change between the constructor and next method call. Since this constructor contains the postcondition `Valid`, any other method with precondition `Valid` may be called after the constructor because it is a valid object state.

4.2.2 Refinement to an Implementation

Figure 30 shows the refinement of the task specification from Figure 29. This refinement contains mostly code. The specifications are implicitly included from the `refines TaskModule` declaration.

The first line contains the module declaration. `refines TaskModule` imports the specification including `Valid`, method preconditions, and postconditions into this module. All classes, data members, and methods are also imported. Next, the `Task`

class is declared (Line 2). The constructor declaration (Line 4) contains both concrete variables and specification-only constructs such as ghost variables. `name` := `newName` are the concrete values of the task's name (Line 6), but `g_name` := `name` assigns the concrete variable to a ghost variable. Assigning a value to both the concrete and ghost variables is a common specification pattern in Dafny [22]. Ghost variables allow for decoupling the implementation and allow for a modular specification. (Although this is a simple example, this pattern is useful when mapping abstract values to a realization, such as an array assigned to a sequence.)

`key` := `newKey` assigns the new key value to the concrete `key` data member (Line 8). `g_key` := `key` assigns the concrete value of key to the respective ghost variable. This follows the same assignment pattern as `name` where `newKey` is assigned to the data member (Line 8) and concrete value is assigned to the ghost variable (Line 9).

Function methods `getName()` and `getKey()` are declared (Lines 13 and 19, respectively). If `task` is an object of type `Task`; `task.name` or `task.key` can be called publicly. `getName()` and `getKey()` also have a public scope. These getter methods are included for convenience. Lines 14 and 20 contain `reads this` allows reference data members of this class in this function method. The return values include `name` (Line 15) and `key` (Line 22) for `getName()` and `getKey()`, respectively. Function methods are called in either implementation or specification contexts.

4.3 FIFO Queue

A FIFO queue is a first-in first-out data structure. The first inserted item is the first item removed. FreeRTOS's FIFO queue contains task objects which are instantiated from the `Task` class in Figure 30 or 29. Tasks are removed from the queue's head

and inserted at the tail-end. The scheduler's ready list is implemented as an array of FIFO queues.

The next sections contain the FIFO queue specification and implementation modules. In comparison to the task modules, the FIFO queue has a specification module and the implementation module is refined from the specification.

4.3.1 IISc's FIFO Queue Schema

Figure 31 shows IISc's `Z` schema for FIFO queue's enqueue operation [8]. `taskIn` and `prioIndex` are input parameters. The Dafny specification follows this in the method signature by allowing a task as a parameter. The enqueue operation will not have two parameters in the method specification. Instead, a `task` object contains a priority as a data member. The Dafny specification also follows concatenating a new task to the FIFO queue. This is shown in the `Z` schema in the second last line where `FIFO‘` is referenced in Figure 31. `Z` language contains different symbols for concatenation when compared to Dafny in which the `+` symbol appends to a sequence.

Figure 32 displays IISc's dequeue operation in `Z` [8]. `taskOut` is the return variable of this schema. The `head` assigns `taskOut` from the front of the queue. In addition, `tail` returns the tasks behind the queue's head. These annotations are followed in the Dafny specification, but the language contains a different notation. However, the queue's head is referenced with `fifo[0]` where `fifo` is a sequence and the tail-end is a sliced with `fifo[1..]`.

4.3.2 FIFO Queue Specification

Figures 33 and 34 displays the FIFO queue specification module. Starting with Part 1, the module is declared (Line 1). The `TaskImplementation` module 30 is

$[FIFO\_enqueue]$

$\Delta ListData$
$taskIn? : TASK$
$prioIndex? : \mathbb{N}$

$prioIndex? \in \operatorname{dom} FIFO$
$\#FIFO(prioindex) < BOUND$
$taskIn? \in \operatorname{ran} FIFO(prioIndex?)$
$PQ' = PQ$
$FIFO' = FIFO \oplus \{prioIndex? \mapsto FIFO(prioIndex?) \frown \langle taskIn? \rangle\}$
$unordered' = unordered$

Figure 31: IISc's FIFO queue enqueue operation

$[FIFO\_dequeue]$

$\Delta ListData$
$taskOut! : TASK$
$index : \mathbb{N}$

$index = max\{i \in \{1 \,..\, \#FIFO \mid FIFO(i) \neq \langle\rangle\}\}$
$taskOut! = head\ FIFO(index)$
$PQ' = PQ$
$FIFO' = FIFO \oplus \{index? \mapsto tail\ FIFO(index?)\}$
$unordered' = unordered$

Figure 32: IISc's FIFO queue dequeue operation

```
module FIFOQueueModule {

        import T = TaskImplementation;

        class FIFOQueue<Task> {
                ghost var fifo : seq<T.Task>;

                var q : array<T.Task>;
                var m : int;
                var n : int;
                var maxSize : int;

                predicate Valid
                        reads this, q;
                {
                        q != null &&
                        Convention(q) &&
                        Correspondence(q)
                }

                predicate Convention(q : array<T.Task>)
                        requires q != null;
                        reads this, q;
                {
                        maxSize >= |fifo| &&
                        0 <= n < maxSize <= q.Length &&
                        0 <= m < maxSize <= q.Length
                }


                predicate Correspondence( q : array<T.Task>)
                        requires q != null;
                        reads this, q;
                {
                        (q.Length >= maxSize > m >= n >= 0 ==>
                                fifo == q[0..n] + q[m..]) ||
                        (0 <= m < n < maxSize <= q.Length ==>
                                fifo == q[m..n] ) ||
                        ( m <= n < maxSize <= q.Length ==>
                                fifo == q[m..n]) ||
                        (q.Length >= maxSize > m >= n ==>
                                q[0..n] + q[m..] == fifo)
                }

                constructor(size : int)
                        modifies this;
                        requires size > 0;
                        ensures Valid;
                        ensures maxSize == size;
                        ensures fifo == [];
                        ensures |fifo| == 0;
```

Figure 33: FIFO Queue Specification Module Part 1

```
1                   method enqueue(task : T.Task)
2                           requires Valid;
3                           modifies this, q;
4                           requires |fifo| < maxSize;
5                           requires task != null;
6                           ensures Valid;
7                           ensures fifo == old(fifo) + [task];
8                           ensures task in fifo;
9                           ensures maxSize == old(maxSize);
10                          ensures |fifo| == old(|fifo|)+1;
11
12                  method dequeue() returns (task : T.Task)
13                          requires Valid;
14                          requires |fifo| != 0;
15                          modifies this;
16                          ensures Valid;
17                          ensures fifo == old(fifo)[1..];
18                          ensures task == old(fifo)[0];
19                          ensures maxSize == old(maxSize);
20
21                  method remove(task : T.Task)
22                          requires Valid;
23                          requires task != null;
24                          modifies this;
25                          ensures Valid;
26                          ensures 0 <= task.key < old(|fifo|) ==>
27                                  fifo == removeAt(old(fifo), task.key);
28                          ensures task !in fifo;
29                          ensures |fifo| == old(|fifo|)-1;
30
31                  function method removeAt(s: seq<T.Task>, index: int): seq<T.Task>
32                          reads this;
33                      requires 0 <= index < |s|;
34                  {
35                     s[..index] + s[index+1..]
36                  }
37
38                  function method isEmpty() : bool
39                          requires Valid;
40                          reads this;
41                          ensures Valid;
42                  {
43                          m == n
44                  }
45
46          }
47  }
```

Figure 34: FIFO Queue Specification Module Part 2

imported because the queue contains task objects (Line 2). This queue is composed of tasks. The FIFO queue class is declared (Line 4). Next, a sequence of type `Task` is a specification construct (Line 5). The FIFO queue is an ordered data structure modeled by this sequence. Each sequence is referenced by an index. Sequences also support concatenation and slicing operations for creating a new sequence. When items are inserted or removed, a new sequence is created from concatenating or slicing the old sequence. `fifo` also contains a data type, `Task`. However, Dafny's module system requires the `T` in `T.Task` because it distinguishes names from other imported modules [3]. This prevents any naming conflicts between modules.

`q` represents an array-based queue (Line 7). This couples the implementation as array-based, but it does not require a bounded queue. If this specification is refined into an unbounded queue, it is possible to copy all existing array elements into a new and resized array [20]. On the other hand, `q` may define the array bounds by calling `q.Length` in the `Valid` predicate. If the queue is bounded, it must be specified in `Valid`. The predicate constrains the class to several reachable states. For example, `q != null` is a valid state, but `q==null` is not valid. Any state not defined in `Valid` will not verify in this class because it is invalid.

Data members `m`, `n`, and `maxSize` couple this specification to a circular and bounded queue (Lines 8-10). `m` points to the first index of the queue while `n` points to the last element. `m` is incremented when an item is removed while `n` is incremented for task insertion. This allows for the wrap-around to behave as a circular queue. `maxSize` limits the capacity of this bounded queue.

[3] see the Dafny module tutorial `http://rise4fun.com/Dafny/tutorial/Modules` for more information

FIFO queue's `Valid` predicate (Line 12) allows reads to data members and `q` with `reads this, q`. Although `q` is a data member, arrays need explicit declaration in `read` clauses. This specifies that elements can be accessed in this predicate. `q` is not `null` (Line 15). For this postcondition to verify, `q` must be initialized.

Two predicates are called in `Valid`. This includes `Convention` and `Correspondence`. Both predicates require `q` is not null and another `reads` clause which is needed because predicates and functions must specify a memory object that may be accessed. Dafny needs to know this for every predicate or function specified. The rest of `Convention` contains the array and sequence bounds that are valid. `maxSize` >= `|fifo|` enforces the size of `fifo` sequence is not larger than the maximum size. `|fifo|` is the cardinality of the sequence which grows and shrinks with the elements inserted or removed from a sequence As stated earlier, inserting or removing elements into a sequence is accomplished by concatenating and slicing sequences. $0 <= n <$ `maxSize` $<= q.Length$ and $0 <= m <$ `maxSize` $<= q.Length$ enforce that `n` and `m` are within the array bounds. For instance, `n` and `m` being outside the array bounds is an invalid state in this specification. `Convention()` prevents `m` and `n` from referencing indexes outside `q`. Both of these variables are referenced in the `Correspondence` predicate (Lines 30-38). This predicate maps the concrete and abstract values (i.e. implementation is mapped to ghost variables). For example, the `fifo` sequence is equal to the sliced array, `q`, within bounds of the circular queue. This disjunction describes how the array implementation corresponds to the specified sequence. Table III lists the valid states of the circular queue.

Valid State 1 in Table III is constrained by a bound where `n` is between zero and `maxSize` along with `m` is equal or greater than `n`. This models the scenario where `m` points to the head, but `n` is less than `m`. This is the wrap-around. `n` only equals `m`

Valid State	Bound	Mapping
1	$q.Length >= maxSize > m >= n >= 0$	$fifo == q[0..n] + q[m..])$
2	$0 <= m < n < maxSize <= q.Length$	$fifo == q[m..n]$
3	$m <= n < maxSize <= q.Length$	$fifo == q[m..n])$
4	$q.Length >= maxSize > m >= n$	$q[0..n] + q[m..] == fifo$

Table III: Tasks states and the associated list

during initialization. **Valid State 2** displays the state the queue will exist during and iterations before the wrap-around. `m` is less than `n`. **Valid State 3** is similar to **Valid State 2**, but both variables are the same during and after initialization. **Valid State 4** also models the wrap-around, but `n` may not equal zero.

`Correspondence` is a disjunction and uses implications which is different from the conjunctive statements in `Valid` and `Convention`. A disjunction loosens the specification since one, several, or all statements may evaluate as true. The implication statements may short-circuit. If the left-side of the implication evaluates as false and the right-side evaluates as true, then this statement evaluates as true. For example, if the wrap-around occurred in the queue, $q[0..n] + q[m..]$ and $q[m..n]$ is not desired, then the $q[m..n]$ will short-circuit and the wrap-around will evaluate as true. This short-circuiting of implication statements is a common pattern in Dafny [23].

`constructor` is declared (Line 40) and accepts the parameter, `size`. The constructor requires the queue's size is one or greater (Line 42). `modifies this` allows the modification of the data members in this class. `Valid` ensures the class is invariant after initialization. `maxSize == size` constrains the queue will have a constant capacity (Line 44). This is the upper bound of the queue's array. `fifo == []` (Line 45) and `|fifo| == 0` (Line 46) ensures the initialization of the sequence modeling the queue. `fifo == []` sets the sequence to empty. `|fifo| == 0` strengthens the

postcondition ensuring zero items in the queue.

`FIFOQueue` includes the operations `enqueue()`, `dequeue()`, and `isEmpty()` which are shown in Figure 34. `enqueue()` inserts at the end of the queue and `dequeue()` removes the head of the queue. In addition to these operations, there are `remove()` and `removeAt()`. Most of the queue operations contain `Valid` as the class invariant. The exception is `removeAt()` only utilized in a specification context as a postcondition in `remove()` (Line 26). `modifies this` is specified in `enqueue()`, `dequeue()`, and `remove()` because each method may change the data members in class `FIFOQueue`.

The first operation is the `enqueue()` method which accepts `task` as a parameter in Figure 34 (Line 1). The preconditions `|fifo|` $<$ `maxSize` (Line 4) and `task` $!=$ `null` (Line 5) require the queue to have fewer items than the maximum size and `task` parameter is initialized. Uninitialized tasks are not part of the queue. The parameter, `task`, is concatenated with the old sequence `fifo` as a postcondition (Line 7). The old value of `fifo` concatenated with `task` (right-side of ==) creates the new `fifo` sequence (left-side of ==). The concatenation defines the ordering of the queue by inserting the item after the last element of the queue. `task in fifo` states that the task is an element in sequence `fifo` (Line 8). `maxSize` == `old(maxSize)` preserves the value and enforces the maximum size does not change after the method call (Line 9). In comparison, Dafny can change the value of `maxSize` after the method call without this statement because the SMT solver will assume a new value for `maxSize` unless a post condition preserves it. The last postcondition, `|fifo|` == `old(|fifo|)` +1, ensures the number of tasks is incremented by one.

The second operation is `dequeue()` which returns a task (Line 12). `|fifo|` $!= 0$ states the queue is not empty. Notice `isEmpty()`'s negation could replace this post-

condition. However, the reference to sequence `fifo` modularizes the specification and does not contain implementation details of the specification. `isEmpty()` references `m` and `n` which are coupled to this implementation of a circular queue. The postcondition, `fifo == old(fifo)[1..]`, removes the head of the queue (Line 17). The `fifo`'s old value is referenced and sliced. `old(fifo)[1..]` keeps all the values of `fifo` and removes the first element, `old(fifo)[0]`. The next postcondition, `task == old(fifo)[0]`, ensures the return value `task` is from the old queue head (Line 18). `maxSize == old(maxSize)` preserves itself in the same fashion as `enqueue()`.

The next operation is `remove()` which provides random-access to an element and removes it from the queue (Line 21). While this is not a typical FIFO queue operation, FreeRTOS's queues may remove a specific task and insert it into another list at certain times. For example, if a task is delayed, it is removed from the ready list regardless of priority and inserted into the delay list. This provides a clean method of removing items when tasks change state (i.e. a task moving from one list to another). As a precondition, `task` contains some value by stating it is not null (Line 23). The postconditions (Lines 26-28) ensures `task` is not in the queue and all other tasks are preserved in the queue. `task !in fifo` states the task is not an element in the new sequence, `fifo` (Line 28). The latter postcondition, `0 <= task.key < old(|fifo|) ==> fifo == removeAt(old(fifo), task.key)`, ensures `task.key` is between bounds 0 and `fifo`'s old size (Line 26). If true, `task` is removed at the specific index. All other tasks are not removed from the sequence except for `task`. `|fifo| == old(|fifo|)−1` ensures the number of queue items is decremented by one (Line 29).

There are two function methods shown: `removeAt()` and `isEmpty()` (Lines 30-43). `removeAt()` accepts a sequence and an index as parameters. The precondition `requires 0 <= index < |s|` constrains the index is in the sequence `s` bounds (Line

32). An item is removed by slicing and concatenating sequence `s` (Line 34). This ensures `s[index]` is removed [4]. The returned sequence will not contain the task at `index`, but it will preserve all other tasks. `isEmpty()` states that if `m == n`, then the queue is empty.

4.3.3 FIFO Queue Implementation

This section describes the FIFO queue specification refinement into an implementation. This follows a similar pattern to the Task specification and implementation. The FIFO queue implementation includes the data members and specifications from the `FIFOQueueModule` import. This section concentrates on Figure 35 which contains the refined code.

`FIFOQueueImplementation` refines the FIFO queue specification from Figure 33 and 34 (Line 2). Throughout this module, the `Valid` predicate, preconditions, and postconditions are implicitly included in this refined module. `FIFOQueue` class is declared (Line 5). All data members from the `FIFOQueueModule` are implicitly included in this refined module. The constructor performs the usual initialization of data members and ghost variables (Lines 6-13). `size` is assigned to `maxSize` (Line 8) which conforms to the postconditions `size == maxSize` (see Figure 33). `q` is instantiated as an array of Tasks (Line 9). This follows `Valid` as a postcondition since `q` is not `null` and the array's bounds are valid. `n` is assigned from `maxSize`-1 (Line 10) and initializes to zero when `enqueue()` reaches the `if` block (Line 17). `n` assigned to `m` initializes the queue as empty (Line 11). The postcondition `m == n` signifies the queue is empty. Sequence `fifo` is initialized as empty with `[]` (Line 12). Ghost variables also need initialization in the constructor.

[4] If you are not satisfied this is true, see `https://dafny.codeplex.com/discussions/529249`. This discusses updating sequences

```
1
2  module FIFOQueueImplementation refines FIFOQueueModule {
3          import opened TaskModule;
4
5          class FIFOQueue<Task> {
6                  constructor(size : int)
7                  {
8                          maxSize := size;
9                          q := new T.Task[maxSize];
10                         n := maxSize-1;
11                         m := n;
12                         fifo := [];
13                 }
14
15                 method enqueue(task : T.Task)
16                 {
17                         if ( n == maxSize-1) {
18                                 n := 0;
19                         } else {
20                                 n := n + 1;
21                         }
22
23                         q[n] := task;
24                         fifo := fifo + [task];
25                 }
26
27                 method dequeue() returns (task : T.Task)
28                 {
29                         if ( m == maxSize-1) {
30                                 m := 0;
31                         } else {
32                                 m := m + 1;
33                         }
34
35                         task := q[m];
36                         DequeueLemma(task);
37                 }
38
39                 ghost method {:axiom} DequeueLemma(task : T.Task)
40                         requires |fifo| > 0;
41                         ensures fifo == old(fifo)[1..] && task == old(fifo)[0];
42
43         }
44 }
```

Figure 35: FIFO Queue Implementation Module

Lines 15 to 25 contain the declaration and body of `enqueue()`. The `if-else` block (Lines 17-21). When the queue is first initialized, `n` == `maxSize`−1 `n` is zero. Afterwards, `n` is incremented by 1 until `n` must wrap-around to the first element in the array. This wrap-around enforces the circular array behavior. `task` is assigned to `q[n]` which is the next item in the array (Line 23). `task` concatenates to the end of the sequence. This allows the array to maintain the queue's ordering because tasks are inserted at the queue's tail-end.

`dequeue()`'s implementation is declared and shown (Lines 27-37). The `if-else` block is the wrap-around (Lines 29-33) for `m` (the queue's front index). If the queue was initialized, contains one element, and `dequeue()` has not been called, `m` is assigned to zero from the `if` branch. `m` will increment by one until the wrap-around occurs and the circular queue changes to `fifo` == `q[m..n]`. `task` := `q[m]` returns the head of the queue (Line 35). `DequeueLemma` helps Dafny show the head of the sequence is the return value, `task` (Line 36). This lemma helps prove the head of the sequence and private array implementation are the same. Dafny's sequence theories are incomplete [5] and requires a lemma to help prove specifications utilizing sequences. This lemma connects the sequence to the corresponding array because Dafny recalls `task` := `q[m]` removes the head task from the array and not the sequence. The additional ghost annotation tells Dafny the queue's head, `task`, is removed from the sequence also. As a result, the `dequeue()` method concludes with removing `task` from both implementation and specification constructs.

[5]Sequence incompleteness is described in this discussion thread: `http://dafny.codeplex.com/discussions/529249`. Lemmas remind Dafny which assertions are still true within the method scope.

4.4 Unordered List

FreeRTOS's suspended list is an unordered list. Any tasks inserted into this list are in the suspended state. Operations including `insert()` which appends a task to the list and `remove()` which provides a random-access removal of a task. This section shows the specification and refined implementation of an unordered list.

4.4.1 IISc's Unordered List Specification

Figure 36 shows the insert and remove schema for unordered list [8]. `taskIn` is the return value. The last lines in both `Unordered_insert` and `Unordered_remove` update `unordered` with an union and removal symbol. Other annotations specify the priority and FIFO queues are maintained which is needed only in `Z`. In Dafny, a task is returned also. Updating the unordered list is different from the IISc schema because it is modeled in Dafny as a sequence. The `insert()` method appends a task to the sequence with concatenation and `remove()` removes the task by slicing the sequence.

4.4.2 Unordered List Specification

Figure 37 and 38 contains the specification for the unordered list module. The `TaskImplementation` module is declared and the task class is imported (Lines 2-3). The import allows `Task` to be referenced throughout the module. Class `UnorderedList` is declared (Line 5). The data members consist of a ghost sequence `unorderedList`, array `u`, `items`, and `maxSize` (Lines 6-9). An unordered list does not need ordering, but a sequence may models this list. Although a sequence preserves the elements' ordering, `unorderedList` is used because it can model a bounded list and supports concatenation needed for insertion and removal. Sequence `unorderedList` is declared (Line 6). The private array implementation is declared as `u` (Line 7). `items` tracks

$$
\begin{array}{|l}
\hline
[Unordered\_insert] \\
\Delta ListData \\
taskIn? : TASK \\
\hline
taskIn? \notin unordered \\
\#unordered < BOUND \\
PQ' = PQ \\
FIFO' = FIFO \\
unordered' = unordered \cup \{taskIn?\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
[Unordered\_remove] \\
\Delta ListData \\
taskIn? : TASK \\
\hline
taskIn? \in unordered \\
PQ' = PQ \\
FIFO' = FIFO \\
unordered' = unordered \setminus \{taskIn?\} \\
\hline
\end{array}
$$

Figure 36: IISc's insert and remove operations for unordered list

the number of tasks in the list (Line 8). `maxSize` is the capacity of the list.

The class invariant `Valid` is declared (Line 11). `reads this, u` allows access to this classes data members and the array, `u`. `u ! = null` specifies `u` must have a value. `Convention()` and `Correspondence()` define the bounds and mapping of the sequence and array elements (Lines 15-16 in `Valid`). `Convention()` is declared (Line 19). `UnorderedList`'s array is passed as a parameter and contains an initialized value (Line 19-20). This specifies the valid state of the unordered list's array, `u`. `items` and `maxSize` are between 0 and the array length (Line 23). This predicate constrains the bounds of the list. `Correspondence` contains an array parameter that is not `null` and within bounds (Lines 26-28). `reads this, u` allows access to this class and array. Sequence `unorderedList` and array `u` are mapped and must contain the same values (Line 31). The ordering is also preserved. The sequence `unorderedList` is less than

```
1  // unordered list module
2  module UnorderedListModule {
3          import opened TaskImplementation;
4
5          class UnorderedList<T> {
6                  ghost var unorderedList : seq<Task>;
7                  var u : array<Task>;
8                  var items : int;
9                  var maxSize : int;
10
11                 predicate Valid()
12                         reads this, u;
13                 {
14                         u != null &&
15                         Convention(u) &&
16                         Correspondence(u)
17                 }
18
19                 predicate Convention(u : array<Task>)
20                         requires u != null;
21                         reads this, u;
22                 {
23                         0 <= items <= maxSize <= u.Length
24                 }
25
26                 predicate Correspondence(u : array<Task>)
27                         requires u != null;
28                         requires 0 <= items <= maxSize <= u.Length;
29                         reads this, u;
30                 {
31                         unorderedList == u[0..items] &&
32                         |unorderedList| <= maxSize
33                 }
34
35                 constructor (s : int)
36                         modifies this;
37                         requires 0 < s;
38                         ensures Valid;
39                         ensures unorderedList == [];
40                         ensures |unorderedList| == 0;
41                         ensures s == maxSize;
```

Figure 37: Unordered List Specification Module Part 1

```
1                   method insert(task : Task)
2                           requires Valid;
3                           modifies this, u;
4                           requires |unorderedList| < maxSize;
5                           requires task != null && task.Valid;
6                           ensures Valid;
7                           ensures unorderedList == old(unorderedList) + [task];
8                           ensures maxSize == old(maxSize);
9
10                  method remove(task : Task)
11                          requires Valid;
12                          modifies this, u;
13                          requires task != null;
14                          ensures Valid;
15                          ensures 0 <= task.key < old(|unorderedList|) ==>
16                             unorderedList == removeAt(old(unorderedList), task.key);
17                          ensures task !in unorderedList;
18                          ensures |unorderedList| == old(|unorderedList|)-1;
19                          ensures maxSize == old(maxSize);
20
21                  function method removeAt(s: seq<Task>, index: int): seq<Task>
22                          reads this;
23                     requires 0 <= index < |s|;
24                  {
25                     s[..index] + s[index+1..]
26                  }
27
28          }
29  }
```

Figure 38: Unordered List Specification Module Part 2

or equal to the capacity, `maxSize` (Line 32). This keeps the sequence within bounds. The list is never larger than `maxSize`.

The constructor initializes the data members (Line 35). The parameter `s` defines the capacity of the unordered list and must be greater than one (Lines 35 and 37). `modifies this` allows access to data members in this class (Line 36). This allows mutation of this class because the data members are set to default values. `unorderedList` is initialized as an empty sequence with no elements (Lines 39-40). The capacity, `s`, must equal `maxSize` since it specifies the list's upper bound (Line 41). Notice the precondition $0 < s$ constrains `maxSize` as greater than zero. This constraint allows the postcondition to follow the same bounds.

Figure 38 displays the second part of the unordered list module. The `insert()` appends `task` to the list (Line 1). The usual class invariant and framing are declared

(Line 2-3). `Valid` constrains properties that do not change in this class and `modifies this, u` allows changing data member values. The `|unorderedList|` $<$ `maxSize` precondition requires the quantity of tasks is less than the upper-bound, `maxSize` (Line 4). The parameter `task` is initialized and valid (Line 5) because `null` tasks are forbidden in the list. `Valid` maintains the invariant properties of this class (Line 6). The postconditions include concatenating the inserted task into this list (Line 7) and preserving `maxSize` (Line 8). The task is properly inserted into the list and `maxSize`'s value is constrained to not change after the method call.

The `remove()` method is shown in Figure 38 too. `Valid` establishes the class invariant and `modifies this, u` allows changes to the data members. The precondition requires the parameter `task` in a initialized state (Line 13). The postconditions show `task` is removed from the unordered list (Lines 15-18). The left side of the implication (Line 15) states the task is removed if it is in bound of sequence `unorderedList`'s previous value. The implication's right side ensures the new `unorderedList` is the old `unorderedList` without parameter `task`. The removal is accomplished with the `removeAt()` function method. The next postcondition, `task !in unorderedList`, guarantees the task is not in the list (Line 17). In case the removal does not occur (Line 15-16), `task` guaranteed to not exist in the list. This is necessary because if `task` is not in the list, `remove()` ensures the task is removed. This is needed for both moving a task from suspended to a different list and deleting tasks. The `unorderedList` is decremented (Line 18) and `maxSize` is preserved because the upper bound must not change. The `removeAt()` function method (Line 21) is the same as its counterpart in the FIFO queue. A sequence is concatenated and sliced at `index`. A new sequence is returned without the item at `s[index]`.

4.4.3 Unordered List Implementation

Figure 39 shows the unordered list's implementation. Along with `remove()`, the `constructor` and `insert()` methods are displayed. The implementation for the constructor and `insert()` are concise, but `remove()` contains two loops which requires invariant and termination annotations. The remove operation also calls the `tailAndRemoveLemma()`. The mentioned figure is described and these upcoming paragraphs explain the annotated code.

Figure 39 shows the `UnorderedListImplementation` module is a refinement of `UnorderedListModule` (Line 1). `Valid`, data members, method preconditions, and postconditions are called implicitly in this class. `UnorderedList`'s constructor requires parameter `size` which is assigned to `maxSize` (Line 6). This follows the postcondition where `maxSize` == `s`. This is the upper bound of the unordered list. Sequence `unorderedList` is initialized to an empty sequence (Line 7). This follows the postcondition where the sequence is initialized as empty. The private implementation array, `u`, is initialized as an array of tasks (Line 8). `u` is not null according to the postcondition. `items` is assigned as zero (Line 9) which follows the postconditions `items` == 0.

The implementation for `insert()` is concise (Lines 12-17). `task` is assigned to the last item of array `u` by `u[items]` := `task` (Line 14). This inserts `task` at the end of `u`. `items` is incremented by one (Line 15). This tracks the number of tasks in list. `unorderedList` := `unorderedList` + `[task]` appends the task to the end of the sequence.

Lines 19-44 list `remove()`'s implementation in which the first loop searched `index` and second loop removes the task. The first `while` loop searches for the `task`'s index

```
module UnorderedListImplementation refines UnorderedListModule {
        class UnorderedList<T> {

                constructor (s : int)
                {
                        maxSize := s;
                        unorderedList := [];
                        u := new Task[s];
                        items := 0;
                }

                method insert(task : Task)
                {
                        u[items] := task;
                        items := items + 1;
                        unorderedList := unorderedList + [task];
                }

                method remove(task : Task)
                {
                        var index := 0;

                        while(u[index] != null &&
                                task != u[index] && 0 <= index < u.Length-1 )
                                invariant 0 <= index < u.Length;
                                decreases u.Length-index;
                        {
                                index := index + 1;
                        }

                        var i := index;
                        while (0 <= i < u.Length-1)
                                invariant u != null && 0 <= i <= u.Length-1;
                                modifies u;
                                decreases u.Length-i;
                        {
                                u[i] := u[i+1];
                                i := i + 1;
                        }

                        items := items - 1;
                        unorderedList := u[0..items];

                        tailAndRemoveLemma(task, index);
                }


                ghost method {:axiom} tailAndRemoveLemma(taskOut : Task, index : int)
                        requires Valid;
                        ensures Valid;
                        ensures taskOut != null &&
                                        0 <= index < |unorderedList| &&
                                        0 <= taskOut.key < old(|unorderedList|) &&
                                        taskOut !in unorderedList;

        }
}
```

Figure 39: Unordered List Implementation

in the unordered list's array (Line 23). The loop exits when the array is either a null value (`u[index] == null`, `task` is found (`task == u[index]`), or the loop's end `index == u.Length-1` (Line 23). The invariant `0 <= index < u.Length` states that `index` ranges between zero and `u`'s length (Line 24). This required for searching for `index` because it is restricted to the unordered list's bounds. The `decreases` clause enforces the loop termination (Line 25). When `u.Length-index` is zero, the loop terminates because it is the last iteration of the loop. `index` is incremented by one until the loop exits (Line 27). `index` is assigned to `i` (Line 30) because it is modified in the loop and `index` is used later. Inside the loop, `task` is overwritten at `i`'s initial value (Line 36). `i` is incremented by one (Line 37) and array items are shifted left (see Line 36). Since `u`'s items are shifted, `modifies u` allows the loop to edit array values (Line 33). Other loop annotations include the invariant which constrains `u` is initialized (Line 32). In addition, the bounds of `i` range from zero to `u.Length`-1 (Line 32) because `i` cannot reference an array element outside the range. However, `i <= u.Length`-1 is valid because the invariant must include a valid state when the loop exits. Another loop annotation is the termination metric in which `u.Length`$-i$ decreases until it equals zero (see Line 34). After the loop exits, `items` is decremented because `task` was removed in the previous loop (Line 40). The `unorderedList` sequence is updated from the modified array values since `u` no longer contains `task` (Line 41). Although the sequence and array were updated, a lemma is needed to verify `taskOut` is removed from the sequence (Line 43). `taskOut` was removed from `u` in the second loop and `unorderedList` was updated, but Dafny does not know the task was removed from the sequence. This is required because sequence theories are incomplete [6].

[6] In this example, the task is removed from shifting in the second loop. Dafny cannot recognize that shifting deletes the task from the array. Furthermore, updating the sequence from the array values does not prove the removal. The lemma allows Dafny to verify the item is removed. Again, see Dafny discussion thread on sequence incompleteness `http://dafny.codeplex.com/discussions/529249`.

4.5 Priority Queue

This section describes the priority queue module containing the specification and code. The implementation consists of a FIFO queues array in which tasks are inserted into the array index associated with the respective priority. For example, a priority two task is inserted with this method call: `q[2].enqueue(task)`. `q` is an array of FIFO queues. This implementation associates a task with a priority without explicitly sorting the queue. This implementation is used due to the difficulty of annotating and verifying a sorted queue [4] [22].

4.5.1 IISc's Priority Queue Schema

Figures 40 and 41 show IISc's `Z` schema. `PQ_enqueue` specifies a task is inserted into the queue in a sorted order. `taskIn` and `key` are parameters for this operation where the key is a priority. `tmpSeqPre` and `tmpSeqSuf` allow for inserting a task somewhere in the sequence. This is similar to concatenating or slicing sequences in Dafny. However, the Dafny specification maintains the sorted order differently. The implementation contains several arrays for each priority. Higher priority arrays are sliced and assigned to a sequence which preserves the sorted order (see next section).

`PQ_dequeue` is similar to the Dafny specification. While the `Z` schema uses `head` and `tail` to return and remove the highest-priority task, Dafny returns the task at the first index and slices the queue. IISc's `PQ_dequeue` operation maintains values for `FIFO` and `unordered` in which this is not needed in Dafny.

4.5.2 Specification and Implementation

This implementation contains the usual enqueue and dequeue operations. In addition, there is a `remove()` method specification, but the implementation is not included for brevity. This method contains a similar specification to unordered list's `remove()`

$[PQ\_enqueue]$

$\Delta ListData$
$taskIn? : TASK$
$key? : \mathbb{N}$
$tmpSeqPre : \text{seq } TASK \times \mathbb{N}$
$tmpSeqSuf : \text{seq } TASK \times \mathbb{N}$

$taskIn? \notin \text{dom } PQ$
$key? \leq BOUND$
$\#PQ < BOUND$
$\forall (t, i) : TASK \times N \bullet (t, i) \in \text{ran } PQ \Rightarrow t \notin taskIn?$
$tmpSeqPre = PQ \upharpoonright \{t : TASK, i : \mathbb{N} \mid i \leq key? \bullet (t, i)\}$
$tmpSeqSuf = PQ \upharpoonright \{t : TASK, i : \mathbb{N} \mid i \leq key? \bullet (t, i)\}$
$PQ' = tmpSeqPre \frown \langle (taskIn?, key?) \rangle \frown tmpSeqSuf$
$FIFO' = FIFO$
$unordered' = unordered$

Figure 40: IISc's enqueue schema for a priority queue

$[PQ\_dequeue]$

$\Delta ListData$
$taskOut! : TASK$

$PQ \neq \langle\rangle$
$taskOut! = head\,(PQ).1$
$PQ' = tail(PQ)$
$FIFO' = FIFO$
$unordered' = unordered$

Figure 41: IISc' dequeue schema for a priority queue

and its implementation is the same for the unordered list. In addition to brevity, the priority queue module contains specifications and method implementation. This is because separating priority queue modules resulted in a failed sanity check. Early version of the priority queue attempted creating two separate modules, but the implementation module failed a sanity check where "assert false" evaluated as true. When the specification and implementation were combined, it passed the sanity checks. Hopefully, improving module support is in future plans for Dafny. Figures 42 and 43 show the priority queue specification and implementation .

`PriorityQueueModule` is declared along with the imported task module and priority queue's class declaration (Lines 1-3). Importing `TaskImplementation` allows referencing `Task` throughout this class. The sequence `pq` models the priority queue and is declared a ghost variable (Line 9). This is needed for specifying an ordered data structure. The sequence will allow for ordering, inserting and removing tasks. `q` provides the private array implementation of this queue (Line 6). In comparison with the FIFO queue (see Figures 33 and 34), this priority queue maps the array values to the sequence in `Valid`. `items` contains the number of tasks in the priority queue (Lines 8). `maxSize` is the number of FIFO queues in `q` (Line 8) where each index is corresponds with a priority. For example, priority three tasks are inserted into `q[3]`.

`Valid` specifies the class invariant (Line 11) which follows the specification pattern similar to classes `Task`, `UnorderedList`, `FIFOQueue`. This invariant describes properties which do not change throughout the class. `reads *` ignores framing constraints. While framing annotations are prefered, this is necessary to avoid strange `read` errors [7]. `q` must be initialized (Line 17). `Convention()` and `Correspondence()` specify

[7]Dafny needs improvement in this area. In ordered for `Valid` to verify, it requires preconditions stating `q`'s array elements are not `null`. However, `Valid` should not have preconditions because it constrains where the predicate can be called. `Valid` should not have any preconditions because its the class invariant and can be called in specifications anywhere its needed. In conclusion, `Valid`

```
module PriorityQueueModule {
        import T = TaskImplementation;
        import F = FIFOQueueModule;

        class PriorityQueue {
                var q : array<F.FIFOQueue<T.Task>>;
                var items : int;
                var maxSize : int;
                ghost var pq : seq<T.Task>;

                var prior : int;
                var topPriority : int;

                predicate Valid
                        reads *;
                {
                        q != null &&
                        Convention(q) &&
                        prior == 5 &&
                        prior <= q.Length &&
                        q[0] != null && q[1] != null && q[2] != null &&
                        q[3] != null && q[4] != null &&
                        q[0].Valid && q[1].Valid && q[2].Valid &&
                        q[3].Valid && q[4].Valid &&
                        Correspondence(q) &&
                        prior >= topPriority >= 0
                }

                predicate Convention(q : array<F.FIFOQueue<T.Task>>)
                        requires q != null;
                        reads this;
                {
                        0 <= items <= |pq|
                }

                predicate Correspondence( q : array<F.FIFOQueue<T.Task>>)
                        requires q != null;
                        requires prior == 5;
                        requires prior <= q.Length;
                        requires q[0] != null && q[1] != null && q[2] != null &&
                        q[3] != null && q[4] != null;
                        reads this, q;
                        reads q[0], q[1], q[2], q[3], q[4];
                {
                        // highest priority is concatenated first
                        pq == q[4].fifo[..]  + q[3].fifo[..] +
                        q[2].fifo[..] + q[1].fifo[..] + q[0].fifo[..]
                }

                constructor(size : int)
                        requires size > 0;
                        modifies this;
                        ensures Valid;
                        ensures items == 0;
                        ensures pq == [];
                        ensures |pq| == 0;
                {
                        items := 0;
                        pq := [];
                        maxSize := 5;
                        q := new F.FIFOQueue[maxSize];
                        q[0] := new F.FIFOQueue(size);
                        q[1] := new F.FIFOQueue(size);
                        q[2] := new F.FIFOQueue(size);
                        q[3] := new F.FIFOQueue(size);
                        q[4] := new F.FIFOQueue(size);
                        prior := 5;
                        topPriority := 0;
                }
```

Figure 42: Priority Queue Part 1

```
method enqueue(task: T.Task) returns (position : int)
        requires Valid;
        requires |pq| < maxSize;
        requires task != null;
        requires 0 <= task.key <= prior < q.Length;
        requires q[task.key] != null && q[task.key].Valid;
        requires |q[task.key].fifo| < q[task.key].maxSize;
        modifies this, q, q[task.key];
        ensures Valid;
        ensures 0 <= position < old(|pq|);
        ensures pq == insertAt(old(pq), position, task);
        ensures task in pq;
{
        q[task.key].enqueue(task);
        position := task.key;
        enqueueLemma(position);
        pq := insertAt(pq, position, task);
        if (task.key > topPriority) {
                topPriority := task.key;
        }
        enqueueLemma(position);
        items := items + 1;
}

ghost method {:axiom} enqueueLemma(position : int)
        ensures Valid;
        ensures  0 <= position < |pq|;
        ensures 0 <= position < old(|pq|);

// remove from the front of the queue
method dequeue() returns (task : T.Task)
        requires Valid;
        requires |pq| != 0;
        requires 0 <= topPriority < q.Length;
        requires q[topPriority] != null && q[topPriority].Valid;
        requires |q[topPriority].fifo| != 0;
        modifies this, q[topPriority];
        ensures Valid;
        ensures task == old(pq)[0];
        ensures pq == old(pq)[1..];
{
        task := q[topPriority].dequeue();
        items := items - 1;
        dequeueBoundsLemma();
        dequeueLemma(task);
        if (task.key <= topPriority && q[task.key].q.Length == 0) {
                topPriority := topPriority - 1;
        }
}

ghost method {:axiom} dequeueBoundsLemma()
        ensures Valid;
        ensures 0 < |pq|;

ghost method {:axiom} dequeueLemma(task : T.Task)
        requires |pq| > 0;
        ensures Valid;
        ensures pq == old(pq)[1..] && task == old(pq)[0];
        ensures task != null;

method remove(task : T.Task)
        requires Valid;
        requires task != null;
        modifies this;
        ensures Valid;
        ensures 0 <= task.key < old(|pq|) ==>
                pq == removeAt(old(pq), task.key);
        ensures task !in pq;
        ensures |pq| == old(|pq|)-1;
```

Figure 43: Priority Queue Part 2

the valid bounds and values mapped between the sequence and array, respectively (Lines 18, 25). This priority queue assume five priorities because it is the default number of priorities in the scheduler (Line 19). In addition, `prior` is less than or equal to `q.Length` since the number of priorities is not larger than the queue length (Line 20). All five queue elements are initialized values and valid (Lines 21-24). The bounds for `topPriority` are defined (Line 26). This tracks which queue element is dequeued. For example, if `topPriority` $== 1$, then `q[1].dequeue()` is called when the priority queue calls its `dequeue` method.

`Convention()` is declared (Line 29). Although it is specified in `Valid`, `q` requires initialization as a precondition (Line 30). `reads this` allows this predicate to access the data members in this class (Line 22). The priority queue's bounds are required as between zero and `|pq|` (Line 24). This defines sequence `pq`'s bounds as it grows and shrinks. Also, `items` is constrained within the array bounds.

`Correspondence()` is declared (Line 36). The preconditions require `q` is initialized and the queue's bounds are valid (Lines 37, 40-41). The `reads` clause provides the same role as in `Convention()` and `Valid` by allowing access to the data members (Line 42). Line 43 allows this predicate to reference `q`'s elements. The `pq` sequence and `q` array values are mapped to each other (Line 46-47). This states the priority queue is the concatenation of all the FIFO queues. The highest priority element (`q[4]`) is concatenated first while lower priority items are appended after. This preserves the sorting order in the priority queue.

`constructor` accepts an integer `s` (Line 50). Each FIFO queue is assigned this value. The precondition requires the size is zero or greater (Line 51). `modifies this` allows for changing values of the data members associated with this class (Line 52).

should not have preconditions in which framing is ignored.

`Valid` is the usual class invariant which states the unchanging properties of this class. The postcondition `pq == []` ensures the sequence is initialized as empty (Line 55) and `|pq| == 0` strengthens the initialization by constraining the sequence size is zero (Line 56). `items == 0` ensures the priority queue is empty (Line 54). `constructor`'s implementation is between Lines 58-68 and starts with initializing `items` to zero. This satisfies the postcondition in Line 54. `pq` is initialized as an empty sequence (Line 59). In addition, `maxSize` is assumed the value of five (Line 60) because there are five priorities. `q` is initialized to five array elements (Line 61). This is followed by initializing each array element (Line 62-66). `topPriority` is initialized to zero since the queue is still empty (Line 68).

Figure 43 contains the other methods for the priority queue. `enqueue()` inserts the parameter, `task`, into the priority queue (Line 1). `Valid` is specified in the precondition and postcondition for preserving the class invariant. `task` requires an initialized value (Line 4). The `|pq| < maxSize` postcondition requires the priority queue is not full (Line 3). The `pq` is referenced to keep the specification modular. This precondition is the same regardless of the implementation because the sequence `pq` is used. The tasks priority, `task.key`, is located in the range between zero and `q.Length` (Line 5). Lines 6-7 allows a valid call to `FIFOQueue`'s enqueue operation. `q[task.key]` is initialized, valid and not empty. There is one framing annotation where `this`, `q`, and `q[task.key]` may change values (Line 8). The next annotations are the postconditions (see Lines 9-12). The `task` is an element in sequence `pq` (Line 12), but a stronger postcondition appends the task to the priority queue (Line 11) by inserting `task` into `pq`. `insertAt()` inserts `taskIn` at `position` in which a new sequence containing the new and old items is returned. These annotations ensures priority queue contains `task`.

`enqueue()`'s implementation starts with inserting `task` into a FIFO queue (Line 14). The task is inserted into the correct priority because `q[task.key].enqueue()` inserts an item into a FIFO queue associated with `task.key`. If `task.key` is two, the task is inserted into `q`'s FIFO queue element at `q[task.key]`. `position` returns the task's priority (Line 15). `enqueueLemma()` allows Dafny to verify that `position` is between the ranges of zero and `pq`'s old and new lengths (Line 16). On Line 17, assigning the return value of `insertAt()` to `pq` satisfies the postcondition on Line 11. The `if` statement updates `topPriority` if `task`'s priority is greater than the current `topPriority` (see Lines 18-20). This updates which FIFO queue contains the highest priority tasks and it is used later in `dequeue()`. `enqueueLemma()` is called again to remind Dafny `position` is in bounds (Line 21). Incrementing `items` updates the number of tasks in the priority queue (Line 22).

`dequeue()` removes the task at the head of the priority queue (Line 31). `Valid` enforces the queue is sorted before and after every method call. Therefore, the head is always a highest priority task which preserves the best-in first-out ordering. A postcondition of this method call requires the priority queue is not empty (Line 33). The next three postconditions (see Lines 34-36) reference the index to the highest priority `pq` element: `topPriority`. The highest priority ranges between zero and `q.Length` (Line 34). `q[topPriority]` is initialized and valid because this element is dequeuing a task (Line 35). In addition, `q[topPriority]` is not empty (Line 36) and it may be modified (Lines 37-38). The postconditions start with `Valid` in which this preserves the class invariant properties (Line 38). The next postcondition ensures `task` is the head from the old priority queue sequence, `pq` (Line 39). This also ensures the return value, `task` contains a value. Even if only one element exists in the priority queue before a call to `dequeue()`, `old(pq)[0]` is not null because the `Valid` predicate does not allow uninitialized tasks. `|pq|` gets the size of the sequence.

`pq == old(pq)[1..]` removes the head from sequence `pq` (Line 40). This slicing notation removes the first element, `pq[0]`, and preserves all the other tasks in the sequence.

The implementation is shown between Lines 42-48. The head task is removed from the priority queue (Line 42). `items` is decremented by one because a task is removed (Line 43). The lemmas `dequeueBoundsLemma()` and `dequeueLemma()` state the priority queue's sequence , `pq`, does not contain `task`. These are ghost annotations which reason if `task` was dequeued from `q[topPriority]`, then it is removed from the specification construct, `pq`. `dequeueLemma()` contains a postcondition that slices `pq` (`pq == old(pq)[1..]`) and reasons `task` is the old priority queue head (`task == old(pq)[0]`) (see Line 58). This satisfies `dequeue`'s postconditions on Lines 39-40. The final part of the implementation updates `topPriority` to the next highest priority (Line 46-48).

The next method, `remove()`, accepts a `task` parameter (Line 61). This operation removes the task from the priority queue. `Valid` is a precondition and postcondition preserving the invariant properties (Lines 62, 65). The precondition `task != null` states the parameter must be initialized (Line 63). A postcondition includes the task is in bounds between zero and the size of `old(pq)` (Line 66). The task is removed from sequence `pq` (Line 67) and preserves the values not removed from the sequence. Also, the task removal is strengthened by stating `task` is not an element of `pq` (Line 68). In case `removeAt()` does not remove the task, `task` must not exist as an element in `pq`. `removeAt()` (not shown for brevity) provides the same functionality as in the unordered list 38 and priority queue 34. A task is removed at `index`. The sequence is sliced and concatenated. The new sequence is returned. The priority queue is decremented (Line 69).

4.6 Summary

The chapter started with explaining the references used for specifying the in Dafny. The FreeRTOS API documentation and IISc `Z` model [8] were the basis of the Dafny specification. However, the `Z` model was widely used throughout the data structure specifications. The API documentation was a supplemental reference. It provides examples of how the and tasks are called.

The task class and several data structures were specified in the next sections. The data structures follow a common specification pattern with `Valid`, preconditions, postconditions, and framing. `Valid` is the class invariant and must evaluate as `true` as a precondition and postcondition of every method. `constructor` only needs `Valid` as a postcondition because it is responsible for initialization. The constructor creates the initial valid state of the object. Methods contain preconditions and postconditions to constrain the state in which the method is called and the state after the method call, respectively. The `reads` and `modifies` clauses are framing annotations. Predicates and functions often specify `read` to allow data members access. `modifies` allows data members to change in the class (i.e. object mutation). Methods which only return a value, but do not write changes to data members do not need `modifies` as seen `Task`'s function methods.

The FIFO queue main operations include `enqueue()`, `dequeue()`, and `remove()`. This queue is used the ready list. `enqueue()` appends a task to the end of the queue, `dequeue()` removes a task from the head of the queue, and `remove()` provides a random-access removal. This queue is bounded and array-based. A sequence models this data structure because it allows ordering, appending, and removal.

The unordered list contains methods for insertion and removal. FreeRTOS's suspend list is an unordered list. The `insert()` operation appends a task to the list and `remove()` provides a random-access removal. The list is also bounded and array-based. A sequence models this data structure, but no ordering is not required. The appending and removal is needed from the sequence for modeling an unordered list.

The priority queue contains `enqueue()`, `dequeue()`, and `remove()`. This queue is used in both delay and waiting lists in FreeRTOS. `enqueue()` inserts a task into the priority queue, `dequeue()` must remove the highest-priority task, and `remove()` provides a random-access removal. This list is bounded and array-based implementation. A sequence models this data structure for ordering, appending, and removal. However, the priority queue must `dequeue()` the highest-priority item (i.e. the best item is removed first). The priority ordering is specified in the class invariant to guarantee the best item is removed first.

CHAPTER V

Verifying the Scheduler in Dafny

The previous chapter presented details of the verified `xList` in Dafny. This is the foundation of the FreeRTOS scheduler because the ready list is an array of FIFO queues, suspended list is an unordered list, waiting and delay lists are priority queues. The scheduler manages tasks by inserting or removing from the appropriate list. The scheduler contains a running task and it may undergo a context switch at every clock tick. Tasks are switched at every tick increment. Scheduler task management includes creating, deleting, updating, and changing tasks. This must follow the real-time constraints found in FreeRTOS. As a result, this behavior is included in the Dafny specification. Port-specific details are not included, but clock behavior is abstracted as a tick. The system clock tick is in every port, but since Dafny is not a low-level programming language, the tick is a software abstraction. All common scheduler behavior is included in this Dafny specification.

The specification utilizes ideas from IISc's `Z` FreeRTOS model [7] in which it has common traits for each port including the scheduler's API. The `Z` model specifies the

scheduler API found in every port [7], but it does not include port-specific details such hardware details. If the generic API is specified, port-specific details should behave the same in every port. As in the IISc `Z` model, these commonalities are specified throughout the Dafny specification. This is mentioned in the first section.

This chapter contains the scheduler specification and implementation. All the modules from Chapter 4 are imported and referenced by the scheduler. Unlike the `xList`'s data structures, the specifications and implementations are included in the same class due to module issues [1]. The scheduler module is explained and split into sections. IISc's `Z` model concepts use in the Dafny specification are covered in the first section. The second section covers the module declaration and data members. Specification constructs are described in the third section. The fourth section covers the method specification and implementation for each scheduler operation.

5.1 Eliciting the Scheduler Specification

There are differences between the IISc `Z` model and Dafny specification. While the classes and methods resemble `Z` schema, the Dafny language does not have the same notation. The `Z` model translation is open to interpretation. There is no straightforward process to convert a `Z` model to a Dafny specification. For example, the `ListData` schema contains the ready, delayed, blocked, and suspend lists [7], but this schema is not converted to a Dafny class. This was a design decision to simplify the scheduler specification. Not all of the `Z` model schemas are translated to Dafny specifications, but there are schema converted into a class such as `Parameter`.

[1] Dafny needs improvement in this area. When the scheduler specification was refined into an implementation module, it failed a sanity check where "assert false" verified.

$$
\begin{array}{|l}
\hline
[Parameter] \\
MAXPRIO : \mathbb{N} \\
MAXDELAY : \mathbb{N} \\
PREEMPTION : BOOL \\
\hline
MAXPRIO > 0 \\
MAXDELAY > 0 \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
[Init\_Parameter] \\
Parameter' \\
MP? : \mathbb{N} \\
MD? : \mathbb{N} \\
PREEMPT? : BOOL \\
\hline
MP? > 0 \\
MD? > 0 \\
MAXPRIO' = MP? \\
MAXDELAY' = MD? \\
PREEMPTION' = PREEMPT? \\
\hline
\end{array}
$$

Figure 44: IISc's `Parameter` Schema

The `Parameter` schema follows the same specification in Dafny. Figure 44 shows the schema. The preemption state is a boolean value while the maximum priority and delay are natural numbers [7]. These values model defined macros from FreeRTOS's code. This is constrained with class `Scheduler`'s `Valid` predicate in Dafny. Additionally, the `Init_Parameter` initializes the values [7], but a constructor performs the initialization in Dafny. The `Z` model's priorities scheme follows a the range of one to `MAXPRIO` [7] where $0 < i <=$ `MAXPRIO` when `i` is a valid priority. The specifications are different in Dafny. The priorities range from zero to `MAXPRIO`-1 such that $0 <= i <$ `MAXPRIO` where `i` is a valid priority. The specification's lowest-priority is zero which these priorities range from zero to `maxpriorities-1` because sequences and arrays are allocated from the first index (i.e. the first index is zero).

$$
\begin{array}{|l}
\hline
[TaskData] \\
tasks : \mathbb{P}\ TASK \\
running\_task : TASK \\
\hline
idle \in tasks \\
running\_task \in tasks \\
\hline
\end{array}
$$

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

Figure 45: IISc's `TaskData` Schema

`TaskData` contains the set of all tasks in the `Z` model [7]. This is shown in Figure 45 where `tasks` is a set of tasks. The Dafny specification utilizes this idea as a ghost declaration, `g_allTasks`. This allows for specifying task insertion or removal from the scheduler when a specific list does not need referencing. The `ListData` schema models the `xList` [7], but the Dafny specification considers this as a high-level design decision. The `xList` is a high-level class design and its implementation consists of separate task lists (see Chapter 2 or 3). This schema also contains `running_task` and an idle task (see Figure 45). In Dafny, the scheduler declares data members `runningTask` and `idleTask` to represent the executing and idle tasks, respectively.

In Figure 46, `topReadyPriority` models the index to the ready list's next running task [7]. This index is shown in the `Task` schema. This idea is utilized in the Dafny specification. It is convenient to track the ready priorities as an index. The Dafny specification models a ready list as a sequence and it is simple to reference (i.e. `g_readyList[topReadyPriority]`).

$[Task]$

$Parameter$
$TaskData$
$ListData$
$PrioData$
$ClockData$
$topReadyPriority : \mathbb{N}$

$idle \in ran \frown / (ran\ ready)$
$topReadyPriority \in \mathrm{dom}\ ready$
$\forall i : \mathrm{dom}\ ready \mid ready(i) \neq \langle\rangle \bullet i \leq topReadyPriority$
$running\_task = head\ ready(topReadyPriority)$
$tasks = \mathrm{ran} \frown / (\mathrm{ran}\ ready) \cup \mathrm{ran}\ blocked \cup \mathrm{ran}\ seqFirst(delay) \cup suspended$
$\mathrm{dom}\ priority = tasks$
$\mathrm{dom}\ basePriority = tasks$
$\forall tcn : \mathrm{ran}\ delayed \bullet tcn.2 > clock$
$\forall t : tasks, \forall i : \mathrm{dom}\ ready \mid t \in \mathrm{ran}\ ready(i) \bullet priority(t) = i$
$\forall i, j : \mathrm{dom}\ blocked \mid i < j \bullet priority(blocked(i)) \geq priority(blocked(j))$

Figure 46: Task schema from IISc

The Z schema partially model the API functions where one operation may have several schema [7]. Figure 47 contains two schemas for creating a task. It models different use-cases of the API functions. For example, `xTaskCreate()` has two different scenarios. If a task is created, it is added either to the ready list or executed as the running task. The Dafny specification contains both scenarios in one method because the methods are annotated to have similar calling behavior to the associated API functions. If `xTaskCreate()` is called in another method, both scenarios are possible use-cases. Since Dafny allows method calls, the specification allows both scenarios to occur with one call. Modeling several use-cases for a scheduler function is common in the Z schemas. Along with `xTaskCreate()`, this appears in `vTaskPrioritySet()`, `vTaskIncrementTick()`, `vTaskDelay()`, `vTaskDelayUntil()`, and `vTaskDelete()`.

Although the Z model is thorough, the Dafny scheduler specification references the API documentation for eliciting the required behavior. This is the primary reference. The API comments are resourceful when there no clear translation from Z to Dafny exists. The next sections quote the API documentation when needed.

$[CreateTaskAndAddToReadyQueue]$

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

$prio? \leq MAXPRIO$
$taskIn? \notin tasks$
$priority(running\_task) \geq prio?$
$\Xi Parameter$
$tasks' = tasks \cup \{taskIn?\}$
$running\_task' = running\_task$
$ready' = ready \oplus \{(prio? \mapsto ready(prio?) \frown \langle taskIn? \rangle)\}$
$blocked' = blocked$
$delayed' = delayed$
$suspended' = suspended$
$priority' = priority \oplus \{(taskIn? \mapsto prio?)\}$
$basePriority' = priority \oplus \{(taskIn? \mapsto prio?)\}$
$\Xi ClockData$
$topReadyPriority' = topReadyPriority$

$[CreateTaskAndSchedule]$

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

$prio? \leq MAXPRIO$
$taskIn? \notin tasks$
$priority(running\_task) < prio?$
$\Xi Parameter$
$tasks' = tasks \cup \{taskIn?\}$
$running\_task' = taskIn?$
$ready' = ready \oplus \{(prio? \mapsto \langle taskIn? \rangle)\}$
$blocked' = blocked$
$delayed' = delayed$
$suspended' = suspended$
$priority' = priority \oplus \{(taskIn? \mapsto prio?)\}$
$basePriority' = priority \oplus \{(taskIn? \mapsto prio?)\}$
$\Xi ClockData$
$topReadyPriority' = prio?$

Figure 47: Modeling `xTaskCreate()` from IISc

5.2 Module Declaration and Data Members

Figure 48 contains the scheduler module and include statements. Starting with the `include` statements (Lines 1-5), the `xList` components are called in the scheduler. As mentioned in Chapter 4, the data structure includes the priority queue, FIFO queue and unordered list. The Task module is also needed because an `xList` contains tasks. The parameter class models a few macros including maximum delay, maximum priority, and preemption.

`SchedulerModule` is declared (Line 8) and the imported modules are shown (Lines 9-13). `FIFOQueueModule`, `PriorityQueueModule`, and `UnorderedListModule` are referenced in the scheduler. These modules compose the `xList` and each respective class is used as a data member (shown later). The `xList` modules are all composed of tasks which the `TaskImplementation` module is imported for this reason.

Class `Scheduler` is declared (Line 15) along with several data members and ghost variables (Lines 16-34). `readyList`, `waitingList`, `suspendList`, and `delayList` are the `xList` instances (Lines 16-19). Notice the high-level design in Chapter 3 `xList` is the base class for the list classes, but the implementation is different. It does not contain a parent class because Dafny does not support inheritance. Instead, scheduler is composed of the list classes representing the `xList`.

`readyList` is the private array implementation containing several FIFO queues (Line 16). Each array index holds the respective priority task. For example, a task with a priority of three is inserted into `readyList[3]`. Inserting a priority task into this list follows the delegation pattern: `readyList[3].enqueue(aTask)` inserts a task into its respective FIFO queue. `waitingList` holds the blocking tasks (Line 17). `suspendList` is implemented as an unordered list (Line 18). A task may remain in

```
1  include "tasks_module.dfy"
2  include "fifo_queue_module.dfy"
3  include "unordered_list_module.dfy"
4  include "PriorityQueueModule.dfy"
5  include "Parameter.dfy"
6
7
8  module SchedulerModule {
9          import opened F = FIFOQueueModule;
10         import opened P = PriorityQueueModule;
11         import opened U = UnorderedListModule;
12         import opened T = TaskImplementation;
13         import opened Pa = ParameterModule;
14
15         class Scheduler {
16                 var readyList : array<F.FIFOQueue<T.Task>>;
17                 var waitingList : P.PriorityQueue;
18                 var suspendList : U.UnorderedList<T.Task>;
19                 var delayList : P.PriorityQueue;
20
21                 var clock : int;
22                 var topReadyPriority : int;
23                 var runningTask : T.Task;
24                 var idleTask : T.Task;
25                 var parameter : Parameter;  // maxprior, maxdelay, preemption
26                 var numberOfTasks : int;
27                 var runningStatus : nat;
28
29                 ghost var g_readyList : seq<F.FIFOQueue<T.Task>>;
30                 ghost var g_waitingList : PriorityQueue;
31                 ghost var g_suspendList : UnorderedList<T.Task>;
32                 ghost var g_delayList : PriorityQueue;
33                 ghost var g_allTasks : seq<T.Task>;
34                 ..
35         }
36 }
```

Figure 48: Includes and Module Declaration

this list indefinitely. `delayList` is also a priority queue (Line 19).

The other data members (Lines 21-27) model several scheduler behaviors. `clock` represents a tick, but it does not contain port-specific details. Every port has `clock` ticks and it increments for every tick event. This value is initialized as zero. When a tick occurs, the value of `clock` changes to the next positive integer and a context switch may occur. `topReadyPriority` contains the highest-priority `readyList` index (Line 22). This list is composed of high-priority tasks. Round-robin scheduling is followed for `readyList[topReadyPriority]`. If the scheduler contains several items in `readyList[topReadyPriority]`, a context switch may change the running task to an item contained in this list at the `topReadyPriority`.

`runningTask` is the current executing process (Line 23). This task must be a task from `readyList[topReadyPriority]`. `idleTask` is a low-priority task in the ready list. `parameter` represents macro values found in every port (Line 25) including `maxprior`, `maxdelay`, and `preemption` [7]. `numberOfTasks` is the total quantity of tasks in the scheduler's lists (Line 26). `runningStatus` is a natural number modeling the scheduler's status where zero is the initialized state, one is the running state, and two is the suspended state. One or many tasks are in `readyList` during the running state and all tasks are inserted into the suspended list when the scheduler is suspended.

The ghost variables are specified for the lists (Lines 29-33). `g_readyList` is a sequence modeling the `readyList`. The values from the array to the sequence are mapped to the same values in `Valid`. `g_waitingList` models `waitingList` implementation (Line 30), `g_suspendList` models `suspendList`, and `g_delayList` models the delay list. `g_allTasks` contains all the tasks in the scheduler. This helps specify amount of tasks and task removal.

This section described the data members including implementation and ghost variables (i.e. concrete and abstract). These are referenced extensively throughout the specification. The data members have invariant states defined by `Valid`. This is described in the next section.

5.3 The Class Invariant

`Valid` contains the scheduler's class invariant (see Figure 49). These values specify the reachable and valid states in class `Scheduler`. This follows the common specification pattern in Dafny as seen in previous chapters. `Valid` defines that data members

```
1                predicate Valid
2                        reads this;
3                {
4                        g_delayList != null &&
5                        g_waitingList != null &&
6                        g_suspendList != null &&
7                        readyList != null &&
8                        parameter != null &&
9                        Correspondence1 &&
10                       Convention1 &&
11                       Correspondence2 &&
12                       Convention2 &&
13                       runningTask != null &&
14                       (runningTask in g_allTasks ) &&
15                       idleTask != null
16               }
```

Figure 49: Valid predicate (i.e. Class Invariant)

are initialized, sequences and arrays are in bounds, and ghost sequences are mapped to arrays.

This predicate declares framing with `reads this` (Line 2). This allows the data member access in this predicate and throughout the methods when `Valid` is preserved. `g_delayList` $!=$ `null` constrains this value must be initialized (Line 4). This follows with `g_waitingList`, `g_suspendList`, and `readyList` are constrained as initialized values (Lines 5-7). For comparison, `g_readyList` $!=$ `[]` is not a constraint because the ready list is initialized in the constructor as an empty sequence, `[]`. An empty sequence and `null` are not the same value because `[]` is an initialized and empty sequence, but `null` is an uninitialized value. As a result, Line 7 allows a valid ready list initialization. `parameter` is an initialized value (Line 8). Lines 9-12 specifies several correspondences and conventions. These map data members to the corresponding ghost variable and defines bounds. These predicates are ordered specifically as defined values and bounds are referenced later. For example, the ready list requires a value before `readyList.Length` $!= 0$ is evaluated. `Convention1` specifies `g_readyList` and `readyList` bounds. `Correspondence2`'s preconditions include a call to `Convention1` because the ready list bounds requires declaration before

```
1         predicate Correspondence1
2                 reads this, suspendList, delayList, waitingList;
3                 requires g_delayList != null &&
4                                 g_waitingList != null &&
5                                 g_suspendList != null;
6         {
7                 g_suspendList == suspendList &&
8                 g_delayList == delayList &&
9                 g_waitingList == waitingList
10        }
```

Figure 50: A correspondence predicate

```
1         predicate Convention1
2                 reads this, parameter, readyList;
3                 requires readyList != null &&
4                                 parameter != null;
5         {
6                 clock >= 0 &&
7                 parameter.maxDelay > 0 &&
8                 |g_readyList| <= parameter.maxPriorities  &&
9                 readyList.Length != 0 &&
10                0 <= parameter.maxPriorities < readyList.Length
11        }
```

Figure 51: A convention predicate

mapping the array values to the specification sequence. `runningTask` must contain an initialized value (Line 13). In addition, there is a running task in the all the scheduler's tasks (Line 14). `idleTask` is also an initialized value (Line 15).

Figure 50 shows `Correspondence1` which requires several ghost variables contain a value before specifying the mapping concrete data members to abstract values. These maps the suspend list's implementation to the ghost variable (Line 7). This pattern is also followed for the delay list and waiting list in Lines 8-9. This predicate may access the data members in this instance (Line 2). This allows `Valid` to call this predicate.

`Valid` contains another predicate, `Convention1`, which defines the bounds for `clock`, `parameter` and `readyList` (see Figure 51). These data members are accessed and referenced by this predicate (Line 2) while `readyList` and `parameter` must contain an initialized value (Lines 3-4) as a precondition. `clock` is a natural

```
1         predicate Correspondence2
2                 reads this, readyList, parameter;
3                 requires readyList != null &&
4                                         parameter != null;
5                 requires Convention1;
6         {
7                 g_readyList == readyList[0..parameter.maxPriorities]
8         }
9
10        predicate Convention2
11                reads this;
12
13        {
14                |g_readyList| > topReadyPriority >= 0 &&
15                numberOfTasks >= 0 &&
16                // 0 == initialized, 1 == running, 2 == suspended
17                (runningStatus == 0 || runningStatus == 1 || runningStatus == 2) &&
18                0 <= roundRobinIndex
19        }
```

Figure 52: Another correspondence and convention predicate

number because a clock tick is not negative when incremented (Line 6). A task's maximum delay is greater than zero (Line 7). `g_readyList`'s size is less than or equal to the maximum number of priorities (Line 8). Similar bounds are defined in Line 10 where the `parameter.maxPriorities`'s ranges between 0 and `readyList.Length`. The ready list's length is never zero (Line 9).

`Convention1` must come before `Correspondence2` in `Valid` because the ready list's bounds must be defined before mapping the implementation to a ghost sequence. This is a precondition for `Correspondence2` which is declared in Figure 52. `reads` allows access to the data members, `readyList`, and `parameter` (Line 2). These values require initialization before this predicate call (Line 3-4). The ready list implementation is mapped to its ghost variable in Line 7. `readyList` and `g_readyList` have a range between zero and `parameter.maxPriorities` to represent the valid priority numbers. The maximum priority number is defined with `parameter.maxPriorities`. Notice the ready list's bounds were defined in `Convention1` because Dafny must know the `readyList` and `g_readyList` bounds before it is referenced. This is the reason it is a precondition (Line 5).

`Valid` specifies which values are initialized and maps implementations to ghost variables. The class invariant preserves these values after the constructor call. `Valid` also preserves data members before and after method calls. The constructor and method specifications and implementations are described in the next section.

5.4 Method Specifications and Implementations

This section describes the annotated methods and code. Common specification patterns are applied which includes preconditions, postconditions, class invariant, and framing annotations. An implementation is included in the method bodies and it is verified correct based on the specifications. Each body may contain additional annotations such as lemmas in order to guide the proof. Dafny may "forget" certain facts during verification since the scheduler specification and code-base is much larger than the `xList` data structures. As a result, code fragments may contain additional annotations from assertions or ghost methods to guide the verification. The annotated scheduler is explained and it starts with initialization.

5.4.1 Initializing and Running the Scheduler

The scheduler may transform into three valid states: initialized, running, and suspended. Initialization occurs before the run or suspend states. When FreeRTOS finishes booting, the scheduler is initialized, but it is not running. One or many tasks are created before the scheduler starts the running state. After the scheduler is in the initial state, `vTaskStartScheduler()` will run and execute ready tasks.

In Dafny, object-oriented design enforces the constructor to initialize the scheduler. The method signature and specification for `constructor()` contains max priority, delay, and preemption parameters in which it is shown in Figure 53. The constructor

```
1         constructor (maxPrior : int, delay : int, preemption : bool)
2                 modifies this;
3                 requires maxPrior > 0 && delay > 0;
4                 ensures Valid;
5                 ensures parameter != null;
6                 ensures topReadyPriority == parameter.maxPriorities-1;
7                 ensures runningTask != null;
8                 ensures numberOfTasks == 0;
9                 ensures |g_allTasks| >= 1;
10                ensures runningStatus == 0;             // initialized
11                ensures idleTask != null;
```

Figure 53: Constructor specification

$$
\begin{aligned}
&Valid() \wedge (maxPrior > 0) \wedge (delay > 0) \wedge \\
&(parameter \neq null) \wedge \\
&(topReadyPriority = parameter.maxPriorities - 1) \wedge \\
&(runningTask \neq null) \wedge (numberOfTasks = 0) \wedge \\
&(\mid g\_allTasks \mid \geq 1) \wedge \\
&(runningStatus = 0) \wedge (idleTask \neq null)
\end{aligned}
$$

Figure 54: Postcondition for `constructor`

begins with initializing data members. Parameters `maxPrior` and `delay` are constrained to any integer greater than zero (Line 3). `preemptions` is a boolean data type. It naturally requires no constraint because `bool` contains either `true` or `false`.

The framing annotation, `modifies this`, allows changing the data members of this instance (Line 2). The postcondition is shown in Figure 54 and Figure 53 Lines 4-11 show the ensures clauses which may mutate the data members. `Valid` is the class invariant which constrains this instance to several states (Line 4). `parameter` is ensured as an initialized value because this object holds the max priority, delay, and preemption enabler (Line 5). `topReadyPriority` must point to a highest-priority task (Line 6). This data member allows a context switch to pick the next task to run. For example, `readyList[topReadyPriority]` points to an array containing the next task to run. `runningTask` is initialized as a value (Line 7). Since creating a task

cannot occur before the scheduler initialization, `runningTask := idleTask` will satisfy this specification. This is intentional and explained further in the implementation. The `g_allTasks` sequence contains all tasks in the scheduler and `|g_allTasks| >= 1` accounts for the idle task (Line 9). This postcondition is different from stating `idleTask in g_allTasks`. This is a "looser" specification to allow for minimal conflict with method calls after `constructor` is called. `runningStatus` is set to zero to represent the scheduler's initialized state (Line 10). Line 11 states that `idleTask` contains an initialized value in which this specification is concluded.

Figure 55 shows the implementation for `constructor`. Lines 1-4 initialize a `parameter` object and assign `maxPrior`, `delay`, and `preemption`. The assignment statements are redundant, but necessary to help Dafny verify `parameter` is initialized. This forces the verifier to keep `parameter`'s data member values. Otherwise, Dafny may assume random values in place of the data members during verification. In the next line, `topReadyPriority` is assigned the ready list's highest-priority index (Line 9). `readyList` is assigned to an array of FIFO queues and its length is `parameter.maxPriorities-1`.

Lines 11-22 initialize the `readyList` array and assigns the array to a ghost sequence `g_readyList`. The loop counter, `i`, starts at zero and the loop condition runs until `!(i < readyList.Length)` (Lines 11-12). `while` can modify the `readyList` array contents with `modifies readyList` (Line 13). The loop invariant has three statements (Lines 14-16) in which the bounds are defined and variables are initialized. `readyList` is an initialized value (Line 14). $0 <= i <=$ *readyList.Length* states the valid bounds of `i` before, during, and after the loop (Line 15). $i == 0$ is true before the loop and $0 <= i <$ *readyList.Length* is valid during the loop. $0 <= i <=$ *readyList.Length* is true after the loop. The last invariant annotation

states that every `readyList` previous element is initialized (Line 16). Without this statement, the loop will exit and Dafny only knows the last element was initialized. Lines 18-19 assign `readyList[i]` to a new `FIFOQueue` instance and `i` is incremented. The `readyList` elements hold ten items. The number of items is an arbitrary assumption because there is not documentation stating the item number held in each `readyList` element. `i` is incremented by one in each iteration. This is important for a termination metric (not shown). Dafny guesses[2] a condition in which a loop terminates. The guessed metric is `decreases readyList.Length-i` where the last loop iteration will evaluate the termination metric as zero. After the loop terminates, the `g_readyList` sequence is assigned all of the initialized `readyList` elements (Line 22). The array is sliced and assigned to the sequence.

`idleTask` is assigned a value in Line 24. The task name, 10101, is arbitrary, but the priority states the `idleTask` is a low-priority task. `runningTask` is the same value as `idleTask` (Line 25). As mentioned earlier, the specification ensures both tasks are initialized, but `idleTask` == `runningTask` may become true. `Valid` states `runningTask` and `idleTask` are not `null` and must be true in all object states. Since tasks are not created before scheduler implementation and the initialization must conform to `Valid`'s specification, the running and idle tasks are the same.

`numberOfTasks`, `clock`, and `roundRobinIndex` are initialized (Lines 32, 43-44). `runningStatus` is zero which signifies the scheduler's state is initialized. Lines 29-30 are related to the sequence, `g_allTasks`, which tracks all tasks in the scheduler. The sequence is set to the default value, `[]`, an empty sequence. `runningTask` is appended to `g_allTasks` (Line 30). Lines 34-41 initialize and set the remaining scheduler lists to an associated ghost variable. This includes delay, waiting, and suspend lists. Each

[2]Dafny has built-in heuristics guessing a termination metric.

```
1                parameter := new Parameter(maxPrior, delay, preemption);
2                parameter.maxPriorities := maxPrior;
3                parameter.maxDelay := delay;
4                parameter.preemption := preemption;
5                assert 0 < maxPrior;
6
7                topReadyPriority := parameter.maxPriorities-1;
8
9                readyList := new FIFOQueue<T.Task>[parameter.maxPriorities];
10
11               var i := 0;
12               while (i < readyList.Length)
13                       modifies readyList;
14                       invariant readyList != null;
15                       invariant 0 <= i <= readyList.Length;
16                       invariant forall j :: 0 <= j < i ==> readyList[j] != null;
17               {
18                       readyList[i] := new FIFOQueue<T.Task>(10);
19                       i := i+1;
20               }
21
22               g_readyList  := readyList[0..parameter.maxPriorities];
23
24               idleTask := new T.Task(10101, 0);
25               runningTask := idleTask;
26
27               numberOfTasks := 0;
28
29               g_allTasks := [];
30               g_allTasks := g_allTasks + [runningTask];
31
32               runningStatus := 0;
33
34               delayList := new P.PriorityQueue(10);
35               g_delayList := delayList;
36
37               waitingList := new P.PriorityQueue(10);
38               g_waitingList := waitingList;
39
40               suspendList := new U.UnorderedList(10);
41               g_suspendList := suspendList;
42
43               clock := 0;
44               roundRobinIndex := 0;
45
46               assert Valid;
```

Figure 55: `constructor` implementation

task list has a abstract and concrete value in which the implementation variable is assigned to the ghost variable.

`assert Valid` is used as a lemma (Line 44). Without this lemma, the constructor does not verify. Dafny "forgets" the class invariant in which the assertion is needed to remind the verifier `Valid` holds. One of `constructor`'s role is to mutate the object into a valid state. Along with `Valid`, the postconditions must conform with the implementation. This allows `Valid` and other postconditions to evaluate as true after the constructor call. This assertion statement (i.e. lemma) reminds Dafny that `Valid` still holds.

After `constructor` is called, the scheduler begins running with calling method `startScheduler()` shown in Figure 56 (vTaskStartScheduler() in FreeRTOS). The method runs until the scheduler is suspended. As a precondition, a running task must exist before the scheduler starts (Line 4). Tick processing is started as shown in the postcondition (Line 10). Task creation must occur between the calls to `constructor` and `startScheduler()`. This information is found in the FreeRTOS's API comment:

> Starts the real time kernel tick processing. After calling the kernel has control over which tasks are executed and when. This function does not return until an executing task calls vTaskEndScheduler (). At least one task should be created via a call to xTaskCreate () before calling vTaskStartScheduler (). The idle task is created automatically when the first application task is created.

The Dafny specification splits this role between methods `constructor` and `startScheduler()`. Some of this is accomplished in the `constructor` because `Valid` must verify as a postcondition. Notice `Valid` is also a precondition in `startScheduler()`

and must verify (Line 2) because the values have invariant states that must hold. `constructor` initializes this valid state. The postcondition is shown in Figure 57 and it is written as ensures clauses in Lines 5-11 (see Figure 56). `idleTask` is created in `constructor` and not `createTask()` because the scheduler must be in a valid state after the constructor call and `idleTask` ! = `null` is in `Valid`. `startScheduler()` does initialize the tick processing (Line 10) and change the status to "running" with `runningStatus` == `1` (Line 9). In addition, `runningTask` is appended to `g_allTasks` in which this sequence represents all scheduler tasks(Line 7). As a result, the final ensures clause states that the running task exists in the scheduler.

The implementation is presented on Lines 13-17. A running task is created with a high-priority (Line 14). This satisfies the postcondition in Line 6. Unfortunately, a dummy task is created to satisfy the postcondition because `idleTask` and `runningTask` should not be equal after this method call. `runningTask` is appended to `g_allTasks` (Line 17) which this satisfies the postcondition in Line 7. `clock` is initialized to zero to begin the tick processing (Line 19). As tasks are running, `clock` is incremented and context switching may occur. `runningStatus` is set to one to represent the scheduler's status is now running.

Figure 58 contains the `endScheduler()` method that mimics `vTaskEndScheduler()`. Besides `Valid` (Line 2), there are no preconditions. As a result, the scheduler may stop running at anytime. The clock ticks are halted and `clock` is set to zero (Line 4). This is satisfied by `clock` := 0 in the method body (Line 9). The scheduler status is set to initialized rather than running (Line 5). `runningStatus` := 0 on Line 10 conforms to this postcondition. Lines 11 and 12 clears the tasks, but the running process is kept. Tasks are cleared, but the scheduler goes into "running a single process" (see API comment). This is represented as `g_allTasks` == 1 to show a single process

```
1  method startScheduler()
2          requires Valid;
3          modifies this;
4          requires runningTask != null;           // requires at least one task
5          ensures Valid;
6          ensures runningTask != idleTask;
7          ensures g_allTasks == old(g_allTasks) + [runningTask];
8          ensures old(numberOfTasks) == numberOfTasks;
9          ensures runningStatus == 1;             // running
10         ensures clock == 0;
11         ensures parameter != null;
12 {
13         runningTask := new T.Task(1111, 4);
14         g_allTasks := g_allTasks + [runningTask];
15         clock := 0;
16         runningStatus := 1;
17         roundRobinIndex := 0;
18 }
```

Figure 56: `startScheduler()` specificity and implementation

$$\begin{aligned} &Valid() \wedge (runningTask \neq null) \wedge \\ &(g\_allTasks = old(g\_allTasks) + [runningTask]) \wedge \\ &(old(numberOfTasks) = numberOfTasks) \wedge \\ &(runningStatus = 1) \wedge (clock = 0) \wedge (parameter \neq null) \end{aligned} \tag{5.1}$$

Figure 57: startScheduler() postcondition

```
1  method endScheduler()
2          requires Valid;
3          modifies this;
4          ensures clock == 0;
5          ensures runningStatus == 0;
6          ensures |g_allTasks| == 1;
7          ensures Valid;
8  {
9          clock := 0;
10         runningStatus := 0;
11         g_allTasks := [];
12         g_allTasks := g_allTasks + [runningTask];
13 }
```

Figure 58: `endScheduler()` specification and implementation

can still run.

Scheduler initialization and suspension were covered in this subsection. Although the scheduler may suspend indefinitely, tasks must exist in the scheduler before it starts running. Task creation is needed to start the scheduler.

5.4.2 Creating and Deleting Tasks

This section describes `createTask()` and `deleteTask()` in which the methods provide creating and deleting task operations. In addition to task creation, `createTask()` inserts tasks into the ready list and performs a context switch if a new task is the highest priority item. `deleteTask()` removes the task from the scheduler. Figure 59 shows the specification and implementation of `createTask()` while Figure 62 displays `deleteTask()`. Both methods are annotated with lemmas also presented in Figures 61 and 63.

`createTask()`'s method signature accepts integers `name` and `priority` and returns `task`. Unfortunately, it has a lengthy specification shown in Lines 2-24. The usual `Valid` predicate and `modifies this` is declared (Lines 2-3). Calls to this method preserve the class invariant with `Valid` and `modifies this` allows changing the data members. Lines 4-5 requires `name` and `priority` are in bounds. A new task is inserted into `readyList[priority]` in which 0 <= `priority` < `|g_readyList` is needed before insertion. A valid `name` is anything zero or greater based `Task`'s specification. Another framing clause, `modifies readyList[priority]`, allows modifying the list at index `priority` (Line 6). This `modifies` is declared after the `priority` bounds precondition (see Line 4) to allow `modifies readyList[priority]` is in bounds.

The postcondition is shown in Figure 60 and described as ensures clauses in Lines 7-23 (see Figure 59). The first statement `Valid` declares bounds and states of this object (Line 7). Lines 9-13 specify the use-case where a context switch occurs. If the running task's priority (i.e. `runningTask.key`) is less than `task`'s priority, a context switch occurs. In case `runningTask` was not initialized, the running task is the created task (Lines 11-12). Lines 14-24 describe events that happen regardless of a context switch. `task` is initialized (Line 14). Lines 15-17 ensure the created `task` objected is placed in the ready list. `priority` is between zero and the ready list's length (Line 15). By stating `g_readyList[priority]` $!= null$, the ready list is initialized at `priority` and the `in` operator ensures `task` is placed in the ready list (line 16). Line 17 ensures `priority` is the same as `task.key`. `numberOfTasks` increments by one because a new task is created (Line 18) and `task` is appended to `g_allTasks` (i.e. the sequence of all tasks) (Line 19). Lines 21-23 ensures `task` is appended to the ready list. `priority` is between zero and the old `g_readyList` length (Line 21) in which these ready list values are initialized (Line 22). After the bounds-checking and initialization annotations, `task` is placed in the ready list by appending `old(g_readyList[priority].fifo)` sequence and task object.

Lines 25-35 contains the implementation body. A new task is created (Line 25). When the method returns, `task` will contain the initialized value. The new task is followed by `createTaskLemma()` which (specified in Figure 61) reminds Dafny that `priority` is in bounds and inserting a new task into the ready list is valid. This lemma is needed because verifier forgets `createTasks()`'s preconditions in which allow `enqueue()`'s contracts are satisfied. `createTaskLemma()` reminds Dafny of `readyList[priority].enqueue()`'s preconditions. As a result, the `enqueue()` method call verifies in Line 28. The task is inserted into the ready list at `priority`. This satisfies the postconditions in Lines 15-17 and 21-23 in which `task` is inserted into

the queue. `numberOfTasks` is incremented (Line 30) which follows the postcondition on Line 18. `task` is appended to the `g_allTasks` (Line 19) which ensures Line 19 is true. The branching statement where `runningTask.key` < `priority` allows a context switch to occur (Lines 33-35). The corresponding postconditions are Lines 9-12.

```
1  method createTask(name : int, priority : int) returns ( task : T.Task)
2          requires Valid;
3          modifies this;
4          requires 0 <= priority < |g_readyList|;
5          requires 0 <= name;
6          modifies readyList[priority];
7          ensures Valid;
8          // is context switch ==>
9          ensures old(runningTask.key) < priority ==>
10                 runningTask == task;
11         ensures old(runningTask) == null ==>
12                 runningTask == task;
13         // both
14         ensures task != null;
15         ensures 0 <= priority <= topReadyPriority < |g_readyList| ==>
16           g_readyList[priority] != null && task in g_readyList[priority].fifo &&
17            priority == task.key;
18         ensures numberOfTasks == old(numberOfTasks)+1;
19         ensures g_allTasks == old(g_allTasks) + [task];
20
21         ensures 0 <= priority <= topReadyPriority < old(|g_readyList|) &&
22           g_readyList[priority] != null && old(g_readyList[priority]) != null ==>
23            g_readyList[priority].fifo == old(g_readyList[priority].fifo) + [task];
24 {
25         task := new T.Task(name, priority);
26         createTaskLemma(priority);
27
28         readyList[priority].enqueue(task);
29
30         numberOfTasks := numberOfTasks + 1;
31         g_allTasks := g_allTasks + [task];
32
33         if (runningTask.key < priority) {
34                 runningTask := task;
35         }
36 }
```

Figure 59: `createTask()` specification and implementation

Figure 62 shows `deleteTask()` which removes the task from the scheduler. The method signature takes task as an input and returns a ghost value, `indexOfItem` (Line 1). Generally, the preconditions require `task` is in bounds and the lists are valid while the postconditions ensure `task` is removed from all the lists. As usual, `Valid` requires the class invariant holds and `modifies` allows the object to mutate (Line 11). `task` is initialized and located between the ranges of zero and `g_readyList` (Lines 4-5). After

$$
\begin{aligned}
&Valid() \wedge (old(runningTask.key) < priority \Rightarrow runningTask = task) \wedge \\
&(old(runningTask) = null \Rightarrow runningTask = task) \wedge \\
&(task \neq null) \wedge \\
&(0 \leq priority \leq topReadyPriority < \mid g\_readyList \mid \Rightarrow \\
&\quad g\_readyList[priority] \ \neq null \wedge \\
&\quad task \in g\_readyList[priority].fifo \wedge priority = task.key) \wedge \\
&(numberOfTasks = old(numberOfTasks) + 1) \wedge \\
&(g\_allTasks = old(g\_allTasks) + [task]) \wedge \\
&(0 \leq priority \leq topReadyPriority < old(\mid g\_readyList \mid) \wedge \\
&\quad g\_readyList[priority] \neq null \wedge old(g\_readyList[priority]) \neq null \Rightarrow \\
&\quad g\_readyList[priority].fifo \ == old(g\_readyList[priority].fifo) + [task])
\end{aligned}
$$

Figure 60: `createTask()` postcondition

```
1 ghost method {:axiom} createTaskLemma(priority : int)
2         requires Valid;
3         requires 0 <= priority < |g_readyList|;
4         ensures Valid;
5         ensures g_readyList[priority] != null;
6         ensures 0 <= |g_readyList[priority].fifo| < g_readyList[priority].maxSize;
7         ensures g_readyList[priority].Valid;
```

Figure 61: `createTask()` lemma

```
1  method deleteTask(task : T.Task) returns (ghost indexOfItem : int)
2          requires Valid;
3          requires task != null;
4          requires 0 <= task.key < |g_readyList|;
5          requires g_readyList[task.key] != null;
6          requires g_readyList[task.key].Valid;
7          requires g_waitingList.Valid;
8          requires g_delayList.Valid;
9          requires suspendList != null && suspendList.Valid;
10
11         modifies this; suspendList, readyList[task.key], waitingList, delayList;
12
13         ensures Valid;
14         ensures 0 <= indexOfItem <  old(|g_allTasks|) ==>
15                 g_allTasks == removeAt(old(g_allTasks), indexOfItem);
16         ensures 0 <= task.key < |g_readyList| ==>
17                 g_readyList[task.key] != null && (task !in g_readyList[task.key].fifo);
18         ensures task in old(g_waitingList.pq) ==>
19                 task !in g_waitingList.pq;
20         ensures task in old(g_suspendList.unorderedList) ==>
21                 task !in g_suspendList.unorderedList;
22         ensures task in old(g_delayList.pq) ==> task !in g_delayList.pq;
23         ensures numberOfTasks == old(numberOfTasks)-1;
24         ensures |g_allTasks| == old(|g_allTasks|)-1;
25         ensures task !in g_allTasks;
26 {
27         indexOfItem := getIndexOfItem(task);
28         removeTaskLemma(task, indexOfItem);
29
30         deleteTaskInvariant(task);
31         readyList[task.key].remove(task);
32         g_readyList := readyList[0..parameter.maxPriorities];
33
34         deleteTaskInvariant(task);
35         waitingList.remove(task);
36         g_waitingList := waitingList;
37
38         deleteTaskInvariant(task);
39         suspendList.remove(task);
40         g_suspendList := suspendList;
41
42         deleteTaskInvariant(task);
43         delayList.remove(task);
44         g_delayList := delayList;
45
46         numberOfTasks := numberOfTasks - 1;
47         deleteTaskInvariant(task);
48 }
```

Figure 62: `deleteTask()` implementation and specification

```
ghost method {:axiom} getIndexOfItem() returns (indexOfItem : int)
        requires Valid;
        ensures Valid;
        ensures  0 <=  indexOfItem  <  |g_allTasks|;
        ensures exists k :: 0 <= k < |g_allTasks| &&
                k == indexOfItem ==> g_allTasks[k] == task;

ghost method {:axiom} deleteTaskInvariant(task : T.Task)
        //requires Valid;
        requires task != null;
        requires 0 <= task.key < |g_readyList|;
        requires g_readyList[task.key] != null;
        ensures Valid;
        ensures g_readyList[task.key].Valid;
        ensures g_delayList.Valid;
        ensures g_suspendList.Valid;
        ensures g_waitingList.Valid;

ghost method removeTaskLemma(task : T.Task, indexOfItem : int)
        requires Valid;
        modifies this;
        requires 0 <= indexOfItem <  |g_allTasks|;
        requires task != null;
        ensures Valid;
        ensures 0 <= indexOfItem <  old(|g_allTasks|) ==>
                g_allTasks == removeAt(old(g_allTasks), indexOfItem);
        ensures task !in g_allTasks;
{
        assert Valid;
        g_allTasks := removeAt(g_allTasks, indexOfItem);
        assume Valid;
        assume task !in g_allTasks;
}

/*
* not part of freertos api. remove task at an index for a seq and return new sequence
*/
function method removeAt(s: seq<T.Task>, index: int): seq<T.Task>
requires 0 <= index < |s|;
{
s[..index] + s[index+1..]
}
```

Figure 63: `deleteTask()` lemmas and a function method

$$
\begin{aligned}
&Valid()\wedge \\
&(0 \leq indexOfItem < old(|\ g\_allTasks\ |) \Rightarrow \\
&\quad g\_allTasks = removeAt(old(g\_allTasks), indexOfItem))\wedge \\
&(0 \leq task.key <|\ g\_readyList\ | \Rightarrow \\
&\quad g\_readyList[task.key] \neq null \wedge task \notin g\_readyList[task.key].fifo)\wedge \\
&(task \in old(g\_waitingList.pq) \Rightarrow \\
&\quad task \in g\_waitingList.pq)\wedge \\
&(task \in old(g\_suspendList.unorderedList) \Rightarrow \\
&\quad task \in g\_suspendList.unorderedList)\wedge \\
&(task \in old(g\_delayList.pq) \Rightarrow task \notin g\_delayTask.pq)\wedge \\
&(|\ g\_allTasks\ | = old(|\ g\_allTasks\ |) - 1) \wedge (task \notin g\_allTasks)
\end{aligned}
$$

Figure 64: `deleteTask()` postcondition

`task`'s bounds are defined, the `g_readyList[task.key]` is initialized (Line 6) and the ready list located at `task.key` may change values (Line 11). The object elements in `readyList` are initialized in `constructor` in which `deleteTask()` requires that initialization at `task.key` and `readyList[task.key]` may change values. Lines 7-8 require the ready list is valid at `task.key` and the waiting list is valid. Also, `waitingList`'s values may change (Line 7,11). `g_delayList` requires validity and its associated concrete variable may mutate (Lines 8,11). Line 9 state the suspend list is initialized, valid, and accessible.

The postconditions are shown in Figure 64 and declared as ensures clauses in Figure 62. Line 13-25 contains the postconditions which starts with `Valid` ensuring the class invariant holds (Line 16). Lines 17-18 remove the task from the `g_allTasks`. `indexOfItem` is between zero and `old(g_allTasks)` length while `removeAt()` removes the task. `task` is removed from the ready, waiting, suspend, and delay lists (Lines 19-24). As long as `task` is removed from the lists, this specification holds.

`numberOfTasks` and `g_allTasks` is decremented to ensure a task is removed (Lines 26-27). Additionally, `task` does not exist in `g_allTasks` (Line 28).

Lines 30-50 contain `deleteTask()`'s implementation with annotations. `getIndexOfItem()` retrieves `task`'s index in `g_allTasks` in which it is a ghost method because `indexOfItem` is only assigned from ghost constructs. `removeTaskLemma()` removes `task` at the `indexOfItem` index (Line 31). Lines 33-35 allow removing `task` from the ready list. The `deleteTaskInvariantTask()` lemma allows a valid ready list removal (Line 33-34). Although `deleteTask()` specifies preconditions allowing a valid `remove()` call, Dafny needs a reminder this call is valid. This call is valid because nothing has violated the method preconditions. The task removal from the ready list is shown at Line 34. The updated ready list is assigned to its associated ghost sequence, `g_readyList` (Line 35). The same pattern is followed through Lines 37-47 for waiting, suspend, and delay lists. `numberOfTasks` is decremented (Line 49) and does not violate any preconditions, but the `deleteTaskInvariant()` is called again to ensure the method postconditions verify (line 50).

This subsection covered creating and deleting task operations. `createTask()` adds a new task to the scheduler in which a context switch might occur. `deleteTask()` removes a task indefinitely. The next section describes another task operation: delaying tasks.

5.4.3 Delaying a Task

This section describes methods which delay tasks. This includes two public methods `taskDelay()` and `taskDelayUntil()` along with a private method, `sleep()`. `taskDelay()` delays a task for a duration, but `taskDelayUntil()` delays a task relative to the clock tick. `sleep()` pretends to do nothing for several clock ticks because

Dafny contains no native `sleep()` method. Two lemma are also described.

Figure 66 shows `taskDelay()`'s specification and implementation. In addition, this method's precondition is shown in Figure 65. The method signature accepts `task` and a time period (Line 1). The class invariant must hold (Line 2). `period` contain an integer greater than zero (Line 3). `modifies task` states the delayed task's data members may change (Line 4). This is also true for `delayList`, `readyList`, and `this` (lines 5-7). `task` and `parameter` are initialized values (Lines 8-11). Line 12 are preconditions for inserting a task in the delay list. As a result, the call to `delayList.enqueue()` is valid. `task.key` is bounded between zero and `g_readyList`'s length and the ready list is initialized at the `task.key` element (Line 13). Line 14 enables modifying `g_readyList[task.key]`. The final precondition requires `g_readyList[task.key].Valid` to evaluate as true (Line 15).

The `taskDelay()` postconditions start with ensuring the class invariant, `Valid`, holds (Line 16). The `ensures clock == period` constraint simulates the task delay lasts until this is true (Line 17). `g_delayList` is initialized and the task is in the delay list (Line 18). Lines 19-20 ensures `task.key` is constrained in bounds and `task` is removed from the ready list.

Lines 22-27 contains the implementation. `delayList.enqueue()` inserts `task` into the delay list (Line 22). This call is valid from the Line 12's precondition, `Valid` (Line 2), and `modifies this` (Line 7). The delay list ghost variable, `g_delayList`, is updated with `delayList` because it was modified (Line 23). `removeLemma()` reminds Dafny the `readyList[task.key].remove()` is a valid call (Lines 24-25). This lemma ensures `g_readyList[task.key]` is valid (see Figure 67 because the preconditions are not violated. `task` is removed from the ready list (Line 25). The task simu-

$$
\begin{aligned}
&Valid()\wedge \\
&(period \geq 0) \wedge (task \neq null) \wedge (task.Valid())\wedge \\
&(parameter \neq null) \wedge (parameter.Valid())\wedge \\
&(g\_delayList.Valid()) \wedge (g\_delayList.pq \mid< g\_delayList.maxSize)\wedge \\
&(0 \leq task.key <\mid g\_readyList \mid) \wedge (g\_readyList[task.key] \neq null)\wedge \\
&(g\_readyList[task.key].Valid)
\end{aligned}
$$

Figure 65: `taskDelay()` and `taskDelayUntil()` precondition

lates sleeping for a time period (Line 26). `removalLemma2()` ensures `delayTask()`'s postconditions are met on Line 27.

Figure 68 shows the `taskDelayUntil()` specification and implementation. The method signature contains inputs `task`, `previousWakeTime`, and `period` in which `task` is initialized and `previousWakeTine` and `period` are greater than zero (Lines 1,3,8). Lines 2-16 are the same preconditions as `taskDelay()` (expect Line 3).

The postconditions are the same as `taskDelay()` with exception to Line 18. `clock == previousWakeTime + period` constrains the task is delayed until this is true. Further more, the implementation contains one different call to `sleep()` (the rest of the implementation is the same). In `taskDelay()`, `sleep(0, period)` will delay a task for a time period, but `taskDelayUntil()` delays the task until a specified time where `previousWakeTime + period == clock`.

This subsection covered delaying tasks. The two delay operations different in which `delayTask()` makes a task sleep for a duration and `delayTaskUntil()` sleeps given a starting time and clock tick deadline.

```
method taskDelay(task : T.Task, period : int)
        requires Valid;
        requires period >= 0;
        modifies task;
        modifies delayList;
        modifies readyList;
        modifies this;
        requires task != null;
        requires task.Valid;
        requires parameter != null;
        requires parameter.Valid;
        requires g_delayList.Valid && |g_delayList.pq| < g_delayList.maxSize;
        requires 0 <= task.key < |g_readyList| && g_readyList[task.key] != null;
        modifies readyList[task.key];
        requires g_readyList[task.key].Valid;
        ensures Valid;
        ensures clock == period;
        ensures g_delayList != null && task in g_delayList.pq;
        ensures 0 <= task.key < |g_readyList| &&
                g_readyList[task.key] != null && task !in g_readyList[task.key].fifo;
{
        var index := delayList.enqueue(task);
        g_delayList := delayList;
        removalLemma(task);
        readyList[task.key].remove(task);
        sleep(0, period);
        removalLemma2(task);
}
```

Figure 66: `taskDelay()` specification and implementation

```
/*private*/
method sleep(previousWakeTime : int, period : int)
        requires Valid;
        requires previousWakeTime >= 0 && period >= 0;
        modifies this;
        ensures Valid;
        ensures clock == previousWakeTime + period;
{
        clock := previousWakeTime + period;
}
```

```
ghost method {:axiom} removalLemma(task : T.Task)
        requires task != null &&
                        readyList != null &&
                        parameter != null &&
                        0 <= task.key < |g_readyList| &&
                        g_readyList[task.key] != null;
        ensures g_readyList[task.key].Valid;
        ensures parameter == old(parameter);

ghost method {:axiom} removalLemma2(task : T.Task)
        ensures Valid;
        ensures task != null &&
                        readyList != null &&
                        parameter != null &&
                        0 <= task.key < |g_readyList| &&
                        readyList[task.key] != null;
        ensures 0 <= task.key < |g_readyList| && g_readyList[task.key] != null &&
          task !in g_readyList[task.key].fifo;
        ensures task in g_delayList.pq;
```

Figure 67: Lemmas called in `delayTask()` and `delayTaskUntil`

```
method taskDelayUntil(task : T.Task, previousWakeTime : int, period : int)
        requires Valid;
        requires previousWakeTime >= 0 && period >= 0;
        modifies task;
        modifies delayList;
        modifies readyList;
        modifies this;
        requires task != null;
        requires task.Valid;
        requires parameter != null;
        requires parameter.Valid;
        requires g_delayList != null;
        requires g_delayList.Valid && |g_delayList.pq| < g_delayList.maxSize;
        requires 0 <= task.key < |g_readyList| && g_readyList[task.key] != null;
        modifies readyList[task.key];
        requires g_readyList[task.key].Valid;
        ensures Valid;
        ensures clock == previousWakeTime + period;
        ensures task in g_delayList.pq;
        ensures 0 <= task.key < |g_readyList| && g_readyList[task.key] != null ==>
                task !in g_readyList[task.key].fifo;
        ensures g_delayList != null;
{
        var index := delayList.enqueue(task);
        g_delayList := delayList;
        removalLemma(task);
        readyList[task.key].remove(task);
        sleep(previousWakeTime, period);
        removalLemma2(task);
}
```

Figure 68: `taskDelayUntil()` specification and implementation

5.4.4 Incrementing the Tick

Clock ticks are important for scheduling tasks in FreeRTOS. This tick is counted in each port and it is incremented in every cycle. After each clock cycle, the scheduler checks for blocked (i.e. waiting) tasks that should change to ready. `incrementTask()` provides this functionality. The FreeRTOS API documentation describes `incrementTask()` in a similar fashion:

> Called from the real time kernel tick (either preemptive or cooperative), this increments the tick count and checks if any tasks that are blocked for a finite period required removing from a blocked list and placing on a ready list.

$$
\begin{aligned}
&\mathit{Valid}()\wedge \\
&(g\_\mathit{waitingList} \neq \mathit{null}) \wedge (g\_\mathit{waitingList.pq} \neq [])\wedge \\
&(\forall\, i : \mathit{int} \mid 0 \leq i <| \; g\_\mathit{waitingList.pq} \mid\Rightarrow \\
&\qquad g\_\mathit{waitingList.pq}[i] \neq \mathit{null} \wedge g\_\mathit{waitingList.pq}[i].\mathit{key} = 0\wedge \\
&\qquad g\_\mathit{waitingList.pq}[i] \notin g\_\mathit{waitingList.pq}\wedge \\
&\qquad g\_\mathit{waitingList.pq}[i] \in g\_\mathit{readyList}[\mathit{parameter.maxPriorities} - 1].\mathit{fifo})\wedge \\
&(\mathit{clock} = \mathit{old}(\mathit{clock}) + 1) \wedge (\mathit{parameter} = \mathit{old}(\mathit{parameter}))
\end{aligned}
$$

Figure 69: `incrementTick()` and `checkBlockedTasks()` postcondition

This method must fulfill two roles. The clock tick increments by one for every call and blocked tasks are checked and may become ready (i.e. the tasks are inserted into the ready list). Figure 70 shows `incrementTask()` and its precondition is shown in Figure 69. Starting with the specification, the class invariant is preserved with `Valid` (Lines 2,7). `modifies this` allows incrementing `clock` since it changes in this method. `parameter` must contain a value (Line 4) and also waiting list must contain a value (Line 5). The waiting list is preserved as an initialized value in Line 8. Lines 9-12 contain a rather messy-looking universal quantifier. This quantifier states that any task finished waiting is removed from the wait list and inserted into the ready list. The messy parts include checking i is in bounds defined in 0 <= i < `|g_waitingList.pq|`. `pq` is the sequence modeling the waiting list and `|g_waitingList.pq|` is the length. `g_waitingList.pq[i] ! = null` states every task, `g_waitingList.pq[i]`, is initialized. Any task where `task.key` == 0 is finished waiting (Line 10). Lines 11 and 12 state the finished and previously waiting task is removed from the wait list and inserted into the ready list. Continuing to the remaining post conditions, `clock` is incremented to track the clock ticks (Line 13). The value for `parameter` is preserved (Line 14), concluding the method specification.

```
method incrementTick()
        requires Valid;
        modifies this;
        requires parameter != null;
        requires g_waitingList != null && g_waitingList.pq != [];

        ensures Valid;
        ensures g_waitingList != null  && g_waitingList.pq != [];
        ensures forall i :: 0 <= i < |g_waitingList.pq| ==>
         g_waitingList.pq[i] != null && g_waitingList.pq[i].key == 0 &&
           g_waitingList.pq[i] !in g_waitingList.pq &&
             g_waitingList.pq[i] in g_readyList[parameter.maxPriorities-1].fifo;
        ensures clock == old(clock)+1;
        ensures parameter == old(parameter);
{
        clock := clock + 1;
        checkBlockTasks();
}
```

Figure 70: `incrementTick()` specification and implementation

```
/*private*/
method checkBlockTasks()
        requires Valid;
        modifies this;
        ensures Valid;
        ensures g_waitingList != null;
        ensures g_waitingList.pq != [];
        ensures forall i :: 0 <= i < |g_waitingList.pq| ==>
          g_waitingList.pq[i] != null && g_waitingList.pq[i].key == 0 &&
            g_waitingList.pq[i] !in g_waitingList.pq &&
              g_waitingList.pq[i] in g_readyList[parameter.maxPriorities-1].fifo;
        ensures clock == old(clock);
        ensures parameter == old(parameter);
```

Figure 71: Private method `checkBlockTasks()`

$$
\begin{aligned}
&Valid()\wedge \\
&(task \neq null) \wedge (0 \leq task.getKey() <| \ g\_readyList \ |)\wedge \\
&(0 \leq newPriority <| \ g\_readyList \ |)\wedge \\
&(g\_readyList[task.getKey()] \neq null \wedge g\_readyList[task.getKey()].Valid())\wedge \\
&(g\_readyList[newPriority] \neq null \wedge g\_readyList[newPriority].Valid())\wedge \\
&(| \ g\_readyList[newPriority].fifo \ |< g\_readyList[newPriority].maxSize)\wedge \\
&(task \in g\_allTasks)
\end{aligned}
$$

Figure 72: `updatePriority()` precondition

The `incrementTick()` implementation increments clock by one (Line 16) and checks for completed waiting tasks (Line 17). `checkBlockTasks()` is a private method which checks for completed waiting tasks and transfers these tasks to the ready list (see Figure 71). This method contains the same postcondition (Lines 8-11) as in `incrementsTask()` (Lines 9-12, Figure 70). This is intentional. An implementation for `checkBlockTasks()` would move all completed waiting tasks to the ready list. This operation is performed every clock tick by checking if a task's priority is zero. The next section describes updating task priority.

5.4.5 Updating a Task's Priority

A task's priority is changed eventually to either a higher or lower priority. When a task gets a lower priority, the task remains in the ready list. If a task gets a higher priority, the task remains in the ready list or a context switch occurs. A switch happens if the task becomes the highest priority task. `updatePriority()` must follow this behavior.

`updatePriority()`'s behavior is shown in Figure 73. In addition, this methods precondition is displayed in Figure 72. The method signature accepts `task` and

```
1  method updatePriority(task : T.Task, newPriority : int)
2          requires Valid;
3          modifies this;
4          modifies task;
5          modifies readyList;
6          requires task != null;
7          requires 0 <= task.getKey() < |g_readyList|;
8          requires 0 <= newPriority < |g_readyList|;
9          requires g_readyList[task.getKey()] != null &&
10                 g_readyList[task.getKey()].Valid;
11         modifies readyList[task.getKey()];
12         requires g_readyList[newPriority] != null &&
13                 g_readyList[newPriority].Valid;
14         modifies readyList[newPriority], readyList[newPriority].q;
15         requires |g_readyList[newPriority].fifo| < g_readyList[newPriority].maxSize;
16         requires task in g_allTasks;
17         ensures Valid;
18         ensures task in g_allTasks;
19         ensures task.key == newPriority;
20         ensures runningTask != null;
21         // context switch
22         ensures topReadyPriority < newPriority ==>
23                 runningTask == task;
24 {
25         readyList[task.getKey()].remove(task);
26         updatePriorityLemma(task, newPriority);
27         task.key := newPriority;
28
29         readyList[newPriority].enqueue(task);
30
31         if (topReadyPriority < newPriority) {
32                 runningTask := task;
33                 g_allTasks := g_allTasks + [runningTask];
34         }
35 }
```

Figure 73: `updatePriority()` implementation and specification

`newPriority` as inputs (Line 1). Postconditions and framing clauses are displayed between Lines 2-16. The class invariant is preserved by `Valid` as a precondition and postcondition (Line 2, 17). The framing annotations modifying `this`, `task`, and `readyList` may change in this method (Lines 3-5). `modifies` allows `task`'s priority to change and move to a different ready list location. `modifies this` is annotated because `runningTask` might change. The input parameter, `task`, is an initialized value by stating it is not `null` (Line 6) . `task` and `newPriority` are between zero and the ready list's length, `g_readyList` (Lines 7-8). Lines 9-14 states the ready list is initialized and modified at specific elements. These annotations allow reading and writing to elements in the ready list. `g_readyList[task.getKey()]` is an initialized value (Line 9), but it must contain a valid state (Line 10). Along with `modifies readyList[task.getKey()]`, these annotations allow reading and writing to ready list elements at index `task.getKey()`. The same pattern is followed in Lines 12-14 to allow reading and writing to the ready list at element `newPriority`. Line 15 is precondition allowing a valid call to `readyList[newPriority].enqueue()` because it is also a precondition in `enqueue()`. The final precondition states `task` is located in the scheduler (Line 16), but this is preserved because `task in g_allTasks` is a postcondition (Line 18). After this method call, `task`'s priority is a new priority (Line 19). `runningTask` is an initialized value (Line 20). The running task either remains the same or switched with a higher priority task. A context switch scenario occurs if `newPriority` is greater than the running task's priority (i.e. `topReadyPriority`) (Line 22-23). This context switch specification concludes the method specification.

The implementation is shown on Lines 25-34 along with additional ghost annotations. `readyList[task.getKey()].remove(task)` removes the task from the ready list at the `task.getKey()` index (Line 25). This call is valid due to the preconditions and framing annotations that allow ready list access. `updatePriorityLemma()` allows

```
1  ghost method {:axiom} updatePriorityLemma(taskIn : T.Task, newPriority : int)
2          requires Valid;
3          modifies this;
4          requires 0 <= newPriority < |g_readyList|;
5          requires g_readyList[newPriority] != null;
6          requires taskIn != null;
7          requires taskIn in g_allTasks;
8          ensures Valid;
9          ensures |g_readyList[newPriority].fifo| < g_readyList[newPriority].maxSize;
10         ensures taskIn in g_allTasks;
```

Figure 74: Lemma called in `updatePriority()`

the preconditions to hold after the call to `remove()` (Line 26). Dafny "forgets" the preconditions after the `remove()` call. `task.key` is assigned to a new priority (Line 29). A context switch might occur between the branching statement (Lines 31-34). If the running task's priority (i.e. `topReadyPriority`) is less than `newPriority`, `task` is running (Line 32). A ghost annotation enforces the modified task is in the scheduler (Line 33).

This subsection described changing a task's priority. Other operations, such as suspending tasks, may change a task's state. The next section covers task suspension.

5.4.6 Suspending Tasks

A task may change to a suspended state where it receives no processor time. This task may become suspended indefinitely or until the task is resumed. Therefore, suspended tasks might not resume. Figure 77 and 78 shows the `suspendTask()` method and lemmas needed for `suspendTask()`.

The method signature contains an input parameter, `task`. The specification is included through Lines 2-19. The class invariant, `Valid`, is preserved (Lines 2,14). `this` and `suspendList` are modified in this method (Lines 3-4). The precondition is also shown in Figure 75, but it is described in Figure 77 (Lines 5-13). The `task` parameter is initialized and valid (Line 5). The suspend list requires validity (Line 6)

and `|g_suspendList.unorderedList| < g_suspendList.maxSize` states that the suspend list is not full (Line 7). Both preconditions allow a valid call by fulfilling the `suspendList.insert()`'s preconditions. Additionally, any call to an object's `Valid` predicate allows reading the data members. Lines 8-9 allow writing and reading to the waiting list with `modifies waitingList` and `requires g_waitingList.Valid`. The same concept is applied to the delay list in Lines 10-11. The task's priority bounds are between zero and ready list's length (Line 12). Writes are allowed through `modifies readyList[task.key]` (Line 13).

The postconditions ensure `task` is inserted into the suspend list and removed from other lists (Lines 15-19). It is also shown in Figure 76. The suspend list contains `task` (Line 15). The task is removed from waiting and delay lists (Line 16-17). `task`'s bounds are between the range of zero and ready list's length (Line 18) and removed from the ready list at the `task.key` index (Line 19). `g_readyList[task.key]` cannot equal `null` in order to state the task is removed from the ready list (see first part of Line 19).

Lines 21-36 contain `suspendTask()`'s implementation along with ghost annotations. `task` is inserted into `suspendList` (Line 21). The update `suspendList` is assigned to its associated ghost variable, `g_suspendList` (Line 22). Method `taskSuspendInvariant()` is called for the first time (Line 23). This reminds Dafny the `suspendTask()`'s preconditions continue to hold. `task` is removed from the waiting list and the ghost value is updated (Lines 25-26). Again, `taskSuspendInvariant()` is called as a reminder to Dafny. The delay list contains similar code in which the task is removed and ghost variable is updated (Lines 29-31). Along with waiting and delay lists, the ready list follows the same pattern in Lines 33-35. `task` is no longer in `readyList[task.key]` (Line 33) and the ghost ready list is updated (Line

$$
\begin{aligned}
&Valid() \wedge \\
&(task \neq null) \wedge (task.Valid()) \wedge (g\_suspendList.Valid()) \wedge \\
&(\mid g\_suspendList.unorderedList \mid < g\_suspendList.maxSize) \wedge \\
&(g\_waitingList.Valid) \wedge (g\_delayList.Valid) \wedge \\
&(0 \leq task.key < \mid g\_readyList \mid)
\end{aligned}
$$

Figure 75: `suspendTask()` precondition

$$
\begin{aligned}
&Valid() \wedge (task \in g\_allTasks) \wedge (task.key = newPriority) \wedge \\
&(runningTask \neq null) \wedge \\
&(topReadyPriority < newPriority \Rightarrow \\
&\quad runningTask = task)
\end{aligned}
$$

Figure 76: `suspendTask()` postcondition

34). Two more lemmas are called which includes `taskSuspendInvariant()` and `taskSuspendRemovalLemma()`. The first lemma reminds Dafny the preconditions still hold. The second lemma tells Dafny `task` is removed from delay and waiting lists; also, the suspend list contains `task` (see Figure 78). The updated ready list was not annotated in a lemma because Dafny can infer the ready list was updated and it does not need restating in a lemma.

5.4.7 Resuming a Task

Not all tasks are suspended indefinitely; some might continue into a resumed state. A tasks may resume with calling `resumeTask()`, but resuming a task requires the suspend list contains it. In this case, the task is removed from the suspend list and inserted into the ready list. Figure 81.

```
method suspendTask(task : T.Task)
        requires Valid;
        modifies this;
        modifies suspendList;
        requires task != null && task.Valid;
        requires g_suspendList.Valid;
        requires |g_suspendList.unorderedList| < g_suspendList.maxSize;
        modifies waitingList;
        requires g_waitingList.Valid;
        modifies delayList;
        requires g_delayList.Valid;
        requires 0 <= task.key < |g_readyList|;
        modifies readyList[task.key];
        ensures Valid;
        ensures task in g_suspendList.unorderedList;
        ensures task !in g_waitingList.pq;
        ensures task !in g_delayList.pq;
        ensures 0 <= task.key < |g_readyList|;
        ensures g_readyList[task.key] != null && task !in g_readyList[task.key].fifo;
{
        suspendList.insert(task);
        g_suspendList := suspendList;
        taskSuspendInvariant(task);

        waitingList.remove(task);
        g_waitingList := waitingList;
        taskSuspendInvariant(task);

        delayList.remove(task);
        g_delayList := delayList;
        taskSuspendInvariant(task);

        readyList[task.key].remove(task);
        g_readyList := readyList[0..parameter.maxPriorities];
        taskSuspendInvariant(task);
        taskSuspendRemovalLemma(task);
}
```

Figure 77: `suspendTask()` specification and implementation

```
ghost method {:axiom} taskSuspendInvariant(task : T.Task)
        requires Valid;
        requires task != null;
        ensures Valid;
        ensures g_waitingList.Valid;
        ensures g_delayList.Valid;
        ensures task != null;
        ensures 0 <= task.key < readyList.Length;
        ensures  0 <= task.key < |g_readyList|;
        ensures g_readyList[task.key] != null;
        ensures g_readyList[task.key].Valid;

ghost method {:axiom} taskSuspendRemovalLemma(task : T.Task)
        requires Valid;
        requires task != null;
        ensures Valid;
        ensures task in g_suspendList.unorderedList;
        ensures task !in g_waitingList.pq;
        ensures task !in g_delayList.pq;
```

Figure 78: `suspendTask()` specification and implementation

Figures 79 and 80 shows the precondition and postcondition for `resumeTask()`. Figure 81. Lines 2-18 contain the method specification. Two common specifications are shown. `Valid`, the class invariant, is maintained (Line 2, 14) and `modifies this` allows write access to this object's data members. There are several similar preconditions from `suspendTask()`. `task` is initialized and valid (Line 4) and also, it is preserved in a postcondition (Line 16). The task ranges between zero and ready list's length (Line 5). Lines 6-8 are required preconditions for calling `readyList[task.key].enqueue()` where ready list is initialized, valid, and not full. The suspend list contains `task` before the method call (Line 9). `modifies suspendList` allows write access while a valid suspend list permits reading data members (Lines 10-11). `readyList[task.key]` may change values (Line 12). The postconditions ensure `task` is removed from the suspended list (Line 14), but the ready list contains the task (Lines 17-18). Additionally, suspend list is an initialized value as the final method annotation (Line 16).

The implementation is compact (Lines 20-21). `task` is removed from suspend list and inserted into the ready list. The task is enqueued at `readyList[task.key]`. The removal and insertion accounts for a task to change from suspend to resume states.

This subsection described resuming a task and concludes methods which mutate tasks. The next section covers "getter" operations for retrieving task values.

5.4.8 Getting the Tick Count, Priority, and Number of Tasks

FreeRTOS provides several task "accessor" operations. Three of these are getters for the tick count, task priority, and task quantity. The operations are converted into Dafny methods as "accessor" methods in which the object can retrieve the data member's value. The methods framing specifications are different from other meth-

$$
\begin{aligned}
&Valid() \wedge (task \neq null) \wedge (task.Valid()) \wedge \\
&(0 \leq task.key <| \ g\_readyList \ |) \wedge \\
&(g\_readyList[task.key] \neq null) \wedge (g\_readyList[task.key].Valid()) \wedge \\
&(| \ g\_readyList[task.key].fifo \ | < g\_readyList[task.key].maxSize) \wedge \\
&(task \in g\_suspendList.unorderedList) \wedge (g\_suspendList.Valid)
\end{aligned}
$$

Figure 79: `resumeTask()` precondition

$$
\begin{aligned}
&Valid() \wedge (task \notin g\_suspendList.unorderedList) \wedge \\
&(task \neq null) \wedge (task.Valid()) \wedge \\
&(g\_suspendList \neq null) \wedge \\
&(0 \leq task.key <| \ g\_readyList \ | \wedge g\_readyList[task.key] \neq null \Rightarrow \\
&\quad task \in g\_readyList[task.key].fifo)
\end{aligned}
$$

Figure 80: `resumeTask()` postcondition

```
1  method resumeTask(task : T.Task)
2          requires Valid;
3          modifies this;
4          requires task != null && task.Valid;
5          requires 0 <= task.key < |g_readyList|;
6          requires g_readyList[task.key] != null;
7          requires g_readyList[task.key].Valid;
8          requires |g_readyList[task.key].fifo| < g_readyList[task.key].maxSize;
9          requires task in g_suspendList.unorderedList;
10         modifies suspendList;
11         requires g_suspendList.Valid;
12         modifies readyList[task.key];
13         ensures Valid;
14         ensures task !in g_suspendList.unorderedList;
15         ensures task != null && task.Valid;
16         ensures g_suspendList != null;
17         ensures 0 <= task.key < |g_readyList| && g_readyList[task.key] != null ==>
18                 task in g_readyList[task.key].fifo;
19 {
20         suspendList.remove(task);
21         readyList[task.key].enqueue(task);
22 }
```

Figure 81: `resumeTask()` specification and implementation

ods mentioned earlier. This is because object mutation does not occur: nothing needs write access. Values are retrieved only. Figure 82 shows `getTaskPriority()`, `getTickCount()`, and `getNumberOfTasks()` methods.

Lines 1-9 retrieve a task's priority. The method signature accepts a `task` parameter and returns a priority. `Valid` preserves the invariant states (Lines 2,5). `task` is initialized and valid (Line 3). `task.Valid` allows that `task.key` is an integer greater than zero and read access to its data members. `priority` == `task.key` constrains the return value is the same as the task's priority (Line 4). The implementation contains one assignment in which priority is `task.key` (Line 7). Thus, it satisfies the postcondition.

Lines 10-18 shows the `getTickCount()` method. The method returns the tick count. The class invariant is preserved (Lines 11,15) along with `clock` and `parameter`. Also, `numberOfTicks` is equal to `clock` because the ticks are counted by the clock. The code contains an assignment statement where `clock` is the return value (Line 17). This satisfies the postcondition, but `clock` == `numberOfTicks` very notably proved as the assignment proves this assertion.

Lines 20-27 contain the `getNumberOfTasks()` method which it returns the task quantity. Validity is preserved as usual (Lines 21,24). The return value is the same as the object's `numberOfTasks` (Line 22) and its value does not change (Line 23). The implementation assigns `this.numberOfTasks` to `numberOfTasks` in which the value is returned.

This subsection described several "getter" methods for retrieving number of tasks, tick count, and task priority. The next section describes changing task lists.

```
1  method getTaskPriority(task : T.Task) returns (priority : int)
2          requires Valid;
3          requires task != null && task.Valid;
4          ensures priority == task.key;
5          ensures Valid;
6  {
7          priority := task.key;
8  }
9
10 method getTickCount() returns (numberOfTicks : int)
11         requires Valid;
12         ensures clock == numberOfTicks;
13         ensures clock == old(clock);
14         ensures parameter == old(parameter);
15         ensures Valid;
16 {
17         numberOfTicks := clock;
18 }
19
20 method getNumberOfTasks() returns (numberOfTasks : int)
21         requires Valid;
22         ensures numberOfTasks == this.numberOfTasks;
23         ensures old(this.numberOfTasks) == this.numberOfTasks;
24         ensures Valid;
25 {
26         numberOfTasks := this.numberOfTasks;
27 }
```

Figure 82: `getTaskPriority()`, `getTickCount()`, and `getNumberOfTasks()` specifications and implementations

5.4.9 Suspending and Resuming All the Tasks

This section describes two methods which suspend and resume all scheduler tasks. `suspendAllTasks()` moves all tasks into the suspended list in which the scheduler status is changed to suspended. Except for the suspended list, all task lists are emptied. `resumeAllTasks()` moves all suspended tasks into the ready list. This returns the scheduler to a running state. `resumeAllTasks()` calls a private method, `sendToReadyList()` which moves all tasks to the ready list.

Figure 85 contains the `suspendAllTasks()` method specification and implementation. Lines 2 and 11 preserve the `Valid` predicate. The precondition is shown in Figure 83. The requires clauses start with the requiring a valid and initialized suspend list (Line 3). This list is not full (Line 4). Similar to the suspend list, the waiting and delay list is valid and initialized (Line 5-6). Framing annotations begin with allowing

write access to `this` in which all object data members are modifiable (Line 7). The waiting and delay lists may mutate also (Lines 9-10). The postcondition is shown in Figure 84. Figure 85 displays the declared ensures clauses in which the suspend list contains all tasks and the scheduler is suspended (Lines 12-18). `runningTask` contains a value (Line 12). All tasks are inserted into the scheduler and it is in the suspended state (Lines 13-14). The waiting and delay lists are empty (Lines 15-16). The ready list is empty too, but the universal quantifier states every element of `g_readyList` does not contain tasks (Line 17-18). This leads to the implementation body.

The implementation starts with a ghost annotation in which the suspend list is the concatenation of the current `g_suspendList` and all tasks (`g_allTasks`) (Line 20). The scheduler transits to the suspended state (Line 21). After the transition, the next line calls `validPriorityQueueLemma()` in which Dafny is reminded the waiting and delay lists are valid (Line 22). Both valid annotations are declared as method preconditions in `suspendAllTasks()`, but the lemma is needed since Dafny forgets these preconditions from Lines 5-6. Figure 85 also shows the `clearLists()` private method and `removeFromAllListsLemma()` are shown. The private method `clearLists()` removes all tasks from ready, waiting, and delay lists to the suspend list (Line 23). `removeFromAllListsLemma()` reminds Dafny the scheduler is in the suspended state and all task lists are cleared except for the suspended list (Line 25). The last statement (Line 24) contains an assumption which restates the concatenation operation on Line 20 in which the postcondition should be satisfied. Unfortunately, the assumption is needed to verify the concatenation postcondition in Line 24 because Dafny forgets the operation performed (see Line 20) [3].

[3]Sequence theories are not complete in Dafny. See `https://dafny.codeplex.com/discussions/529249`

Figure 87 shows `resumeAllTasks()`'s specification and implementation. A few common annotations are shown such as the class invariant and framing. The `Valid` predicate is preserved (Lines 1,6). This method can modify `this` and `suspendList` (Lines 4-5). The precondition `runningStatus` == 2 requires the scheduler is suspended (Line 3). The postcondition is shown in Figure 86. The ensures clauses (see Figure 87) include universal quantifier, clearing the suspended list, and changing the scheduler state. The quantifier states that all tasks are moved into the ready list (Lines 7-10). Line 7 states the bounds for `g_allTasks` in which each of its tasks are initialized. Each task's priority ranges between zero and `g_readyList`'s length (Line 8). Line 9 states each task is initialized. The task at index `i` is in the ready list (Line 10). The suspended list is cleared (Line 11), but `g_suspendList` maintains an initialized value. Its sequence is cleared and `g_suspendList` ! = `null` would violate `Valid`. As the final postcondition, the scheduler status changes to running (Line 12).

Lines 14-16 contains the implementation. A ghost annotation assigns an empty sequence, `[]`, to `g_suspendList.unorderedList`. This clears the ghost suspended list. `runningStatus` is assigned a one signifying the scheduler is running (Line 15). `sendToReadyList()` sends all suspended tasks into the ready list (Line 16). Lines 20-29 contains the private method's specification. `sendToReadyList()`'s postconditions satisfies `resumeAllTasks()`'s postconditions in which this verifies.

This subsection described two mutation operations used on task lists. The scheduler changes states in these operations and moves task to different lists (i.e. suspend or ready lists). The next section covers changing the running task.

$$Valid() \wedge (g\_suspendList.Valid) \wedge (g\_suspendList.u \neq null) \wedge$$
$$(\neg g\_suspendList.isFull()) \wedge (g\_waitingList.Valid) \wedge$$
$$(g\_waitingList.q \neq null) \wedge$$
$$(g\_delayList.Valid) \wedge (g\_delayList.q \neq null)$$

Figure 83: `suspendAllTasks()` precondition

$$Valid() \wedge (runningTask \neq null) \wedge$$
$$(g\_suspendList.unorderedList = old(g\_suspendList.unorderedList) + old(g\_allTasks)) \wedge$$
$$(runningStatus = 2) \wedge (g\_waitingList.items = 0) \wedge$$
$$(g\_delayList.items = 0) \wedge$$
$$(\forall\, k : int \mid 0 \leq k <| \; g\_readyList \mid \wedge g\_readyList[k] \neq null \Rightarrow$$
$$\mid g\_readyList[k].fifo \mid = 0)$$

Figure 84: `suspendAllTasks()` postcondition

5.4.10 Context Switch

FreeRTOS supports multitasking in which it must schedule tasks to receive fair-share of processor time. Multiple tasks alternate between the running and ready states for a given time period. Tasks run for a specific time period then are swapped with the next ready task. Context switching is a core component in a real-time system because it guarantees a task's running duration until a switch occurs. Figure 90 shows this operation. Additionally, Figures 91 and 92 displays lemmas called in `switchContext()` and `getTopReadyPriority()`.

`switchContext()` begins with specifications seen in previously described methods. In addition to Figures 88 and 89, Figure 90 lists the method specification. The class invariant is preserved (Lines 2, 17) while `this` and `runningTask` support write access (Lines 3-4). The next requires clause (Line 5) states that the

```
method suspendAllTasks()
        requires Valid;
        requires g_suspendList.Valid && g_suspendList.u != null;
        requires !g_suspendList.isFull();
        requires g_waitingList.Valid && g_waitingList.q != null;
        requires g_delayList.Valid && g_delayList.q != null;
        modifies this;
        modifies suspendList;
        modifies waitingList, waitingList.q, waitingList.pq;
        modifies delayList, delayList.q, delayList.pq;
        ensures Valid;
        ensures runningTask != null;
        ensures g_suspendList.unorderedList ==
                old(g_suspendList.unorderedList) + old(g_allTasks);
        ensures runningStatus == 2;
        ensures g_waitingList.items == 0;
        ensures g_delayList.items == 0;
        ensures forall k :: 0 <= k < |g_readyList| && g_readyList[k] != null ==>
                |g_readyList[k].fifo| == 0;
{
        g_suspendList.unorderedList := g_suspendList.unorderedList + g_allTasks;
        runningStatus := 2;
        validPriorityQueueLemma();
        clearLists();
        removeFromAllListsLemma();
        assume g_suspendList.unorderedList ==
                old(g_suspendList.unorderedList) + old(g_allTasks);
}

ghost method {:axiom} validPriorityQueueLemma()
        ensures Valid;
        ensures waitingList.Valid;
        ensures delayList.Valid;

/*private*/
method clearLists()
        requires Valid;
        requires waitingList != null;
        requires waitingList.q != null;
        requires waitingList.Valid;
        requires suspendList != null;
        requires suspendList.u != null;
        requires delayList != null;
        requires delayList.q != null;
        requires delayList.Valid;
        requires suspendList != null;
        requires suspendList.u != null;
        modifies this, waitingList, waitingList.q, delayList, delayList.q, suspendList;
        ensures Valid;
        ensures g_waitingList.items == 0;
        ensures g_delayList.items == 0;
        ensures forall k :: 0 <= k < |g_readyList| && g_readyList[k] != null ==>
                |g_readyList[k].fifo| == 0;

ghost method removeFromAllListsLemma()
        requires Valid;
        requires g_suspendList != null;
        modifies this;
        ensures Valid;
        ensures runningStatus == 2;             // 2 == suspended
        ensures g_delayList.pq == [];
        ensures forall k :: 0 <= k < |g_readyList| && g_readyList[k] != null ==>
                g_readyList[k].fifo == [];
```

Figure 85: `suspendAllTasks()` specification and implementation

$$
\begin{aligned}
&\mathit{Valid}() \wedge \\
&(\forall\, i : \mathit{int} \mid 0 \leq i < \mathit{old}(|\ \mathit{g\_allTasks}\ |) \Rightarrow \mathit{old}(\mathit{g\_allTasks}[i]) \neq \mathit{null} \wedge \\
&\quad 0 \leq \mathit{old}(\mathit{g\_allTasks}[i].\mathit{key}) <| \ \mathit{g\_readyList}\ | \ \wedge \\
&\quad \mathit{g\_readylist}[\mathit{old}(\mathit{g\_allTasks}[i].\mathit{key})] \neq \mathit{null} \wedge \\
&\quad \mathit{old}(\mathit{g\_allTasks}[i]) \in \mathit{g\_readyList}[\mathit{old}(\mathit{g\_allTasks}[i].\mathit{key})].\mathit{fifo}) \wedge \\
&(\mathit{g\_suspendList}.\mathit{unorderedList} = []) \wedge (\mathit{runningTask} = 1)
\end{aligned}
$$

Figure 86: `resumeAllTasks()` postcondition

```
1  method resumeAllTasks()
2          requires Valid;
3          requires runningStatus == 2;
4          modifies this;
5          modifies suspendList;
6          ensures Valid;
7          ensures forall i :: 0 <= i < old(|g_allTasks|) ==> old(g_allTasks[i]) != null &&
8            0 <= old(g_allTasks[i].key) < |g_readyList| &&
9              g_readyList[old(g_allTasks[i].key)] != null &&
10               old(g_allTasks[i]) in g_readyList[old(g_allTasks[i].key)].fifo;
11         ensures g_suspendList.unorderedList == [];
12         ensures runningStatus == 1;
13 {
14         g_suspendList.unorderedList := [];
15         runningStatus := 1;
16         sendToReadyList();
17 }
18
19 /*private*/
20 method sendToReadyList()
21         requires Valid;
22         modifies this;
23         ensures Valid;
24         ensures forall i :: 0 <= i < old(|g_allTasks|) ==> old(g_allTasks[i]) != null &&
25           0 <= old(g_allTasks[i].key) < |g_readyList| &&
26             g_readyList[old(g_allTasks[i].key)] != null &&
27               old(g_allTasks[i]) in g_readyList[old(g_allTasks[i].key)].fifo;
28         ensures g_suspendList != null && g_suspendList.unorderedList == [];
29         ensures runningStatus == 1;
```

Figure 87: `resumeAllTasks()` specification and implementation

scheduler is not in the suspended state (i.e. `runningStatus ! = 2 `). Lines 6-8 requires `topReadyPriority` ranges between zero and ready list's length along with `g_readyList[topReadyPriority]` is initialized. Lines 9-12 fulfill the preconditions for calling `remove()` and `enqueue()`. The ready list is not full at index `topReadyPriority` (Line 9). `readyList[topReadyPriority]` is readable and writable (Lines 10-11) along with the ready list is not empty (Line 12). The postconditions are shown between Lines 14-17. `topReadyPriority` ranges between zero and ready list's length (Line 14) and `g_readyList[topReadyPriority]` contains a value (Line 15). A context switch is specified concisely as `runningTask ! = null` (Line 16). By stating the running task is some initialized value, the task is either a different task or the same if there is a single ready task. As long as `runningTask` is not annotated as preserved, this specifies a context switch. Finally, `parameter` is preserved (Line 17).

The implementation is shown (Lines 19-23). In case the running task is in the ready list, `remove()` ensures the list does not contain `runningTask` (Line 19). `roundRobinLemma()` allows the running task to change (Line 20). This reminds Dafny `switchContext()`'s preconditions still hold. In addition, `enqueue()` and `dequeue()` are valid calls. `runningTask` is appended ready list's tail-end and the next running task is assigned (Lines 21-22). This switch inserts the running task into `readyList[topReadyPriority]` and assigns the next task to run with `dequeue()`. `changeTaskLemma()` specifies the context switch and allows Dafny to verify it (Line 23).

The FreeRTOS API does not specify if `switchContext()` will update `topReadyPriority`. `switchContext()` performs a round-robin switch if `topReadyPriority` (i.e. next running task's index) has not changed. In case the `topReadyPriority` needs to change, the `getTopReadyPriority()` method updates this value. Its specification is shown in Figure 92. This method finds the next ready priority and assigns it to

$$
\begin{aligned}
&Valid() \wedge (runningStatus \neq 2) \wedge \\
&(0 \leq topReadyPriority < |\ g\_readyList\ | \wedge \\
&\quad g\_readyList[topReadyPriority] \neq null \wedge \\
&\quad g\_readyList[topReadyPriority].q \neq null) \wedge \\
&(|\ readyList[topReadyPriority].fifo\ | < readyList[topReadyPriority].maxSize) \wedge \\
&(readyList[topReadyPriority].Valid) \wedge (g\_readyList[topReadyPriority].fifo \neq [])
\end{aligned}
$$

Figure 88: `switchContext()` precondition

$$
\begin{aligned}
&Valid() \wedge (|\ g\_readyList\ | > topReadyPriority \geq 0) \wedge \\
&(g\_readyList[topReadyPriority] \neq null) \wedge \\
&(runningTask \neq null) \wedge \\
&(parameter = old(parameter))
\end{aligned}
$$

Figure 89: `switchContext()` postcondition

`topReadyPriority` (Lines 5). Line 6 defines `nextReadyPriority` ranges between zero and ready list's length. This method is specified if needed.

This subsection described changing the running task. Context switching allows a scheduler to allocate processor time to other tasks. The next section describes two more "getter" methods.

5.4.11 Get Current Task and Scheduler Status

There are two more accessor methods. `getCurrentTask()` retrieves the current running task and `getSchedulerStatus()` returns whether the scheduler status is initialized, running, or suspended. Both methods are shown in Figure 93. Lines 1-10 displays `getCurrentTask()` which returns the current running task. The class invariant is preserved (Lines 2-3). The postcondition `currentTask` == `runningTask`

```
method switchContext()
        requires Valid;
        modifies this;
        modifies runningTask;
        requires runningStatus != 2;
        requires 0 <= topReadyPriority < |g_readyList| &&
                g_readyList[topReadyPriority] != null &&
                        g_readyList[topReadyPriority].q != null;
        requires |readyList[topReadyPriority].fifo| < readyList[topReadyPriority].maxSize;
        requires readyList[topReadyPriority].Valid;
        modifies readyList[topReadyPriority];
        requires g_readyList[topReadyPriority].fifo != [];
        ensures Valid;
        ensures |g_readyList| > topReadyPriority >= 0;
        ensures g_readyList[topReadyPriority] != null;
        ensures runningTask != null;
        ensures parameter == old(parameter);
{
        readyList[topReadyPriority].remove(runningTask);
        roundRobinLemma();
        readyList[topReadyPriority].enqueue(runningTask);
        runningTask := readyList[topReadyPriority].dequeue();
        changeTaskLemma();
}
```

Figure 90: `switchContext()` specification and implementation

```
method getTopReadyPriority() returns (nextReadyPriority : int)
        requires Valid;
        modifies this, readyList[topReadyPriority];
        ensures Valid;
        ensures topReadyPriority == nextReadyPriority &&
                0 <= nextReadyPriority < |g_readyList|;
        ensures parameter == old(parameter);
        ensures readyList[topReadyPriority] != null;
```

Figure 91: `getTopReadyPriority()` method specification

```
ghost method roundRobinLemma()
        requires Valid;
        modifies this, readyList[topReadyPriority];
        ensures Valid;
        ensures readyList[topReadyPriority] != null &&
                readyList[topReadyPriority].Valid;
        ensures |readyList[topReadyPriority].fifo| < readyList[topReadyPriority].maxSize;
        ensures runningTask != null;
        ensures |g_readyList[topReadyPriority].fifo| > 1 ==>
                g_allTasks == old(g_allTasks);

/*
* Dafny forgets runningTask != null && Valid == true,
* even though this is true throughout the rest of the method.
*/
ghost method changeTaskLemma()
        modifies this;
        ensures Valid;
        ensures runningTask != null;
```

Figure 92: Lemmas for `switchContext()`

```
1  method getCurrentTask() returns (currentTask : T.Task)
2          requires Valid;
3          ensures Valid;
4          ensures currentTask == runningTask;
5          ensures runningTask == old(runningTask);
6          ensures currentTask != null;
7
8  {
9          currentTask := runningTask;
10 }
11
12 method getSchedulerStatus() returns (status : nat)
13         requires Valid;
14         ensures Valid;
15         ensures runningStatus == status;
16         ensures runningStatus == old(runningStatus);
17 {
18         status := runningStatus;
19 }
```

Figure 93: `getCurrentTask()` and `getSchedulerStatus()` specifications and implementations

ensures the return value is the running task (Line 4). `runningTask` is preserved (Line 5) and `currentTask` is initialized (Line 6). The implementation assigns `runningTask` to `currentTask` which returns the running task (Line 9).

Lines 12-19 contains the `getSchedulerStatus()` method which returns the scheduler status. As mentioned earlier, `status` is either zero, one, or two. Zero represents the scheduler is initialized, one is the scheduler is running, and two the scheduler is suspended. The specification starts with preserving the class invariant, `Valid` (Lines 13-14). This predicate also defines status is a natural number ranging for zero to two. The return value equals the running status (Line 15). `runningStatus` is preserved (Line 16). In conclusion of the additional "getter" methods, the implementation assigns `runningStatus` to `status` in which the value is returned (Line 18).

5.5 Summary

This chapter covered FreeRTOS's scheduler API which consisted of a specification and implementation. The first section described eliciting the FreeRTOS API

specifications from IISc's `Z` model. The schema are referenced and utilized throughout the Dafny specification, but FreeRTOS's API documentation provided additional guidance. While the `Z` model minimizes ambiguous language because it is formal language, the schemas are open to interpretation when translating to Dafny. In comparison to the `Z` model, the API documentation is written in natural language, but it is well-written and detailed. Section two specified the scheduler class's data members. These included imported modules, `xList` components, tick counting (i.e. `clock`), `runningTask`, `idleTask`, non-port macros (i.e. `parameter`), `runningAStatus`, and various ghost variables. Section three covered the class invariant, `Valid`. This predicate specifies which values are initialized. `Convention()` and `Correspondence()` predicates define bounds and mappings of concrete values to specification variables (i.e. ghost annotations). `Valid` is declared as a precondition and postcondition in methods to preserved the object's invariant states. The fourth section contains specification annotations and code for `Scheduler`'s methods. Except for the constructor, all methods preserve the class invariant. Any operations mutated the object contain the `modifies` clause to allow changing the data members. `constructor` contains `Valid` only as a postcondition because the object is initialized. Scheduler operations include initialization, start scheduling, manipulating tasks, incrementing the clock tick, context switching, and various accessor methods.

CHAPTER VI

Lessons Learned

There are notable lessons learned from using Dafny including several advantages and disadvantages. The program verifier is a tool: it is worth stating Dafny's limitations, but also its wonderful features which other formal method's have not obtained. Whether Dafny is a hammer or wrench, this tool should be applied when necessary: a wrench is used on bolts, but it is not a replacement for a hammer. This chapter covers how well Dafny functions as a tool.

6.1 Dafny's Pros and Cons

This section explains Dafny's advantages and disadvantages in a pros-and-cons format. The first part contains advantages while the second part describes the disadvantages. Recommendations are given regarding features or improvements in the disadvantages section.

6.1.1 Pros

There are several advantages to using Dafny including that it is a simple language and object-oriented. It is both a verifier and compiler. Ghost annotations such as sequences and sets can model data structures while refinement is supported through modules. Verification time is relatively quick. This section describes the advantages.

Simplified Language Dafny avoids language complexity by containing a minimal amount of programming and specification constructs. Users are not overwhelmed by different constructs to solve solutions. In addition, Dafny's proof rules are simpler. For example, there is a single looping construct in which searching and assigning values are common. Since loop invariant annotations are often lengthy, Dafny also provides a `forall` "loop" that allows multiple and simultaneous assignments for arrays and sequences [20]. The language does not contain a `for`, `do-while`, or `foreach` loop. This may increase the work of Dafny users, but the language does not have an overwhelming amount of programming constructs.

Constructs common in various programming languages are also included. These were used throughout Chapters 3-5. Typical constructs include arithmetic, branching, assignment, primitive data types, arrays, and user-defined data types.

A Verifier and Complier Verifying programs is the main activity in proving program correctness. While this starts with a specification, an implementation is usually a refinement step after creating a formal specification. Not all formal methods tools contain a compiler (i.e. model checkers) [6]. Dafny contains a compiler in which .NET Framework bytecode is generated [21]. Therefore, the refined program code can execute in a .NET Framework program. This allows verified software components to be used in real programs instead of containing only a formal model.

Object-oriented Object-oriented programming (OOP) is supported by allowing users to create classes and instantiate objects. Methods provide different operations on an object. Inheritance and sub-typing is not supported [22], but delegation is a utilized in which objects may another through methods. Modern programs written in object-oriented languages can verify in Dafny. This programming paradigm was used throughout this thesis in previous chapters. Classes and objects were declared and instantiated, respectively. This allows for reusable implementations as seen in OOP.

Sets and Sequences Dafny can verify unordered and ordered data structures with sets and sequences, respectively. Sequences were used to verify task lists in which specifications contained slicing and concatenation operations (see Chapter 4). Sets may represent unordered data structures in which operations include intersection, union, and difference. These are also utilized in dynamic framing [19].

Termination Heuristics While loop `invariants` are often difficult, a related annotation is automated. A neat feature includes "guessing" the termination metric for a `while` loop [22]. This automates part of the loop annotations. Dafny attempts this automatically when a loop is declared. If a termination metric is not guessed, Dafny will state the user needs to define the `decreases` clause.

Refinement with Modules Refinement is described as developing a program in iterations starting from the specification and resulting in an implementation [6]. Programs are developed in iterations starting from a specification and resulting with an implementation. Modules allow for reusing specifications and refining into program code. This was used in the `xList` and scheduler where `refines` transforms specification module into code.

Performance Dafny's SMT solver is very efficient. The Schorr-Waite algorithm example contains thirty-two lines of loop invariant annotations and it is verified in under five seconds [19]. Another example is the verification for the scheduler and its underlying data structures. Dafny takes approximately fifty seconds for proving seventy-three verifications conditions. Considering the scheduler is large and Dafny must apply many different theories to the proof, performance is very good.

6.1.2 Cons

There are disadvantages to using Dafny as a verification tool. Lemmas and framing issues are troublesome. Incompleteness of sequences requires lemmas to help verification. The lack of documentation may lead to frustration because there are many undocumented features of Dafny. These are all discussed.

Creating Lemmas Whether a ghost method or assertion, lemma creation is a common activity in Dafny. Assertions are often needed to remind the verifier of certain facts while ghost methods help Dafny in less obvious proofs. In addition, method implementations may contain ghost annotations. The ideal Dafny program contains modular specifications where methods bodies contain only code. Method level specifications only contain specification constructs. However, some situations require writing code which does not separate the specification from the implementation (see `suspendAllTasks()` method in Chapter 5).

One recommendation is further modularizing specifications from the implementation. Other verifiers such as Resolve, separates the specification from the implementation using concepts and realizations [11]. This allows calling modular specifications from reusable components. A concept contains the specification for a component such as a list data structure. Also, theories are imported to model data structures

or components (e.g. a string theory is imported to model a list). The concept contains specifications and method signatures only. A realization reuses a concept and its specification by defining the convention and correspondence in which concrete variables are mapped to abstract values. Except for annotations for loop invariants, the implementation (e.g. realization) contains no additional specification constructs because everything the theorem prover knows regarding the realized data structure is available in the concept.

A Resolve-like concept and realizations may reduce or eliminate the need for lemmas. If Dafny's development team included concepts and realizations, the most likely solution utilizes "pure mathematical modeling" to further modularize specifications as recommended by Bronish and Weide [4]. The development team may need to decide if further modularizing specifications is a project goal.

Framing Issues Framing constructs include `reads` and `modifies` which allow accessing and changing values in functions, methods or classes. Objects and arrays are statically defined in framing annotations throughout the scheduler and task lists (see previous chapters 4-5). When using classes, dynamic framing is available to declare several valid states for the instantiated object as it mutates [20] [19]. This is automatically applied when declaring a class with `autocontracts`.

Dynamic framing is not used in the task lists and scheduler. It was used in earlier revisions of this Dafny specification. Unfortunately, dynamic framing seems to contain limitations when declaring several layers of classes. The delegation/composition pattern allows `xList` components to become data members in the scheduler class. After several layers of classes are declared, it is unclear how to define dynamic framing in the `Valid` predicate. In addition, there is no documentation defining how

```
1  var s : seq<int>;
2  s := [];
3  s := s + [0];
4  s := s + [1];
5
6  assert 0 in s;  // verifies
7  assert 1 in s;  // verifies
8
9  s := s + [2];
10 assert 2 in s;  // needed or will not verify
11
12 s := s + [3];
13 assert 3 in s;  // this is needed
14
15 var i := 1;
16 var s0 := insertAt(s, i, 5);
17
18 assert 0 in s0;
19 assert 1 in s0;
20 assert 2 in s0;
21 assert 3 in s0;   // verifies only assert 3 in s, several lines above, is there
22 assert 5 in s0;
```

Figure 94: Sequence incompleteness example

`autocontracts` applies dynamic framing to multiple class levels. The recommended solution is providing documentation for dynamic framing with object-oriented code.

Incompleteness of Sequence Theories As footnoted throughout this thesis, the sequence theories are incomplete in Dafny[1]. This discussion thread example was posted and it is shown in Figure 94. After every concatenation operation (Lines 3,4,9,12), assertions are needed to prove an item is contained in sequence. The comments mark which lines are needed for a successful verification. Lines 18-22 test the integers are part of the sequence `s0`.

In addition to incompleteness, a minor bug does not allow referencing sequences with natural numbers[2]. Sequences are referenced extensively in the task lists and scheduler which this shows this construct is quite usable. A very common pattern in Dafny is assigning an array to a sequence (see Chapters 4-5). In this case, sequences should use natural numbers because all sequence indexes are zero or positive integers.

[1]Again, see `https://dafny.codeplex.com/discussions/529249`

[2]This is an extended Dafny tutorial which mentions integers are used, but not natural numbers with sequences: `http://rise4fun.com/Dafny/tutorial/Sequences`

This is accomplished by referencing array and sequence indexes with integers. As a result, integer bounds are defined as a natural number, but the actual `nat` data type is not used because this bug. The short-term solution is recording and documenting limitations of sequences. This allows Dafny users less guessing when working with sequences. The long term goal should include improving sequence theories as Dafny matures as a verification tool.

Lack of Documentation While the discussion threads [3], examples [4] and papers [22] [20] [19] [18] [12] are very helpful; official documentation such as a user manual is needed. Any examples on dynamic framing, `autocontracts`, classes, sequences, and modules are valuable. There is some literature on these topics, but it is not compiled into one document. All mentioned Dafny papers took over a year to find. When answers were not found in papers, the discussion threads are helpful. However, a user manual would provide a faster reference in which common questions may be answered. When all else fails, users are left to trial-and-error and guessing to verify Dafny programs. As a result, the user manual would solve this problem along and remedy many other issues stated in previous paragraphs.

6.2 Summary

This chapter discussed Dafny's pros and cons. Simplified language keeps specification and constructs to a minimal in which the basics are easy to learn. Simplicity is not the only advantage. An implementation is not only verified, but it can compile. Whether compiling or verifying, the language is object-oriented in which it allows verifying modern programming languages. The Dafny language also includes ghost annotations such as sets and sequences. These can describe unordered and ordered

[3]Dafny discussion threads: urlhttps://dafny.codeplex.com/discussions
[4]Dafny code examples: urlhttps://dafny.codeplex.com/SourceControl/latest

data structures. Other annotations include loop invariants and termination metrics. While loop invariants are required annotations, Dafny automates the termination metric by "guessing" when the loop ends. Besides annotation constructs, refinement is supported. Another feature is verifying specifications in a reasonable duration. Dafny solves proofs efficiently, but it requires lemmas when certain constructs such as sequences are used. Sequence theories are incomplete. Other issues are related to classes. Framing issues exist when multiple levels of classes are declared through composition. Most problems may be solved with additional papers or documentation. Literature on Dafny is available, but official user documentation is lacking. User documentation may solve many shortcomings by providing a compilation of common issues and solutions.

CHAPTER VII

Related Work

This chapter describes a parallel FreeRTOS project and other approaches taken to verifying this real-time operating system. The first section describes the Indian Institute of Science's (IISc) FreeRTOS verification research in which this thesis referenced numerous times. The second section discusses another project using the B-method to specify and implement a verified FreeRTOS. The third section discusses another formal methods project possibility.

7.1 IISc's xList and Scheduler

FreeRTOS is documented as API comments containing parameters, return values, calling example, and a description. The parameters and return values state the function signature while the calling example shows how the it is called. A description summarizes the function's behavior and possible constraints. This is an informal and imprecise document. Converting the documentation into a formalized specification allows for a precise and unambiguous specification. Translating all of these components

into a Z model provides a formal, precise, and unambiguous documentation.

The API comments are declared in C source files. In addition, the code-base may provide insight on a function's behavior. The Z notations allows creating a formal, unambiguous, and precise document in set theory and formal logic [8]. The model shown in Z Model of Tasks Lists in FreeRTOS documents the xList data structures' operations formally and tools are used to validate and simulate the schema [8]. After this was accomplished, the scheduler documented into Z operation schema [7].

After completing a Z model of the xList and scheduler, translating the Z model to an annotated C implementation is the next research goal. Microsoft's C program verifier, VCC, would allow creating a specification and implementation; and also proving both correct. The challenging part is converting Z to VCC specifications. There is no official process of converting Z to VCC specifications, but the annotations are also based on set theory and formal logic. A separate implementation is also coded which conforms to the specification.

Other challenges include verifying concurrency and numerous annotations. VCC contains annotations for proving parallel programing. Spin-locks and monitors are presented in the documentation because it is common method for handling concurrency. Besides concurrency, a heavy amount of annotations required for verification. Specifications are less modular than Dafny and may have more annotations per lines of code.

The completion of this research will contain a formal specification and verified implementation of FreeRTOS's task lists and scheduler. If any bugs are found, the FreeRTOS code-base may be modified and corrected. Note that this does not eliminate the need for traditional software quality assurance; testing, code reviews, walk-

throughs, and inspections are still needed [3]. A formal specification and verified implementation further minimizes defects in software, but may not eliminate them completely.

7.2 The B Method

Formalizing FreeRTOS: First Steps converts this operating system into a formal language similar to Z. The B method contains set theory and logic-based specifications and focuses on refining the specification into executable code [6]. Similar to IISc's research, verification starts with translating the FreeRTOS documentation to a formal specification. A basic model is created from the requirements and its operations are defined. Each refinement step expands the model to include tasks, queues, mutexes, and valid scheduler states [6]. Every step adds another requirement to the model and consistency is checked. The requirements are reviewed if any inconsistency in the model. The final refinement artifact contains the implementation in a programming language [6].

The main advantage to the B method is the final artifact consists of program code. This is a very useful feature in which not all formal methods tools accomplish. For example, stand-alone theorem provers do not provide tools for refinement. Z3 is an example of this, but Dafny does support refinement through a module system (see Chapter 5). The B method supports refinement whether through a process or tool set. In addition, tools are available for all development stages [6].

While the B method has many positive aspects, Formalizing FreeRTOS: First steps states that the tool support needs improvement [6]. The tool support includes specification, interactive proofs, and code generation **??**. While there are a variety of

B method tools available, some may be cumbersome as stated by `Deharbe et al`. Their research described the beginning steps of formalizing FreeRTOS, but it did not contain refined code. The tool support may have improved since this research occurred several years ago. Despite the flawed tools, this is very promising formal methods research.

7.3 Model-checking and FreeRTOS

Creating a FreeRTOS formal specification can be done in a model-checker. The FreeRTOS API documentation would translate into set theory and formal logic in which the language is similar to specification annotations used in program verifiers (i.e. Dafny). However, model-checkers often support temporal logic in which modeling timing events is possible. Model-checking tools such as NuSMV contain LTL or CTL constructs in the language [14]. This would allow NuSMV to model timing events such as a tick in FreeRTOS.

A model-checker is perfect for modeling ticks and checking if the timing requirements hold. NuSMV supports modules in which `xList` and scheduler can be separated into different components. NuSMV is open-source and it is a more mature tool than Dafny. There is plenty of support and examples available for this model-checking tool.

While the tool maturity and ability to model temporal events are advantages, refinement is very difficult. Refining a formal model into executable code is difficult because there is no implementation or programming language associated with model-checking tools. Program verifiers such as Dafny refining a specification into code, but this is not common in model-checking.

7.4 Summary

This chapter covered related work to verifying FreeRTOS in which several formal methods projects were described. The first is a program verification project. IISc translated the FreeRTOS documentation into a `Z` model that captured the `xList` and scheduler requirements. In addition, there are plans to convert the `Z` model into an annotated C code-base using `VCC`. This research used the `Z` formal language and plans to refine the specification into an implementation. There is another similar project. The Formalizing FreeRTOS: First Step paper presents research on translating the documentation into another formal language using the B method. The tools and techniques will refine the specification into executable code. However, the tool support exists, but it needs improvement. Another tool which may formalize FreeRTOS is a model-checker. While tools such as NuSMV are mature and stable, refinement into program code is difficult. The advantages include modeling temporal events with logic not supported in other formal methods tools.

CHAPTER VIII

Conclusion

Software engineering formal methods provide reduction in program defects in which producing high-integrity systems is possible [3]. Real-time software such as FreeRTOS requires integrity, robustness, reliability, availability, and temporal constraints. In addition, the operating system's website claims quality [2]. Formal methods may lower defects in FreeRTOS. As a result, program verification is another SQA activity to further minimize software defects [3] [9]. We proposed proving the correctness of FreeRTOS using the Dafny verifier. As a result, a specification and implementation of the scheduler API and task list data structures were created and verified.

Verifying FreeRTOS requires a formal specification and correct implementation. Dafny provides annotations for specifying FreeRTOS's behavior into contracts in methods, classes, and modules. The API documentation and IISc's `Z` model [7] [8] was referenced and a Dafny specification was created in classes and methods. The `xList` was verified in which it contains priority queue, FIFO queue, and unordered list. Each data structure was specified and implemented as a class. The scheduler

API uses the `xList` data structures in ready, delay, waiting, and suspended lists. These task lists, `Parameter`, and `Task` classes were declared in modules and imported by the scheduler. The scheduler operations were specified and implemented in a class with its respective task-managing operations.

There were challenges to verifying FreeRTOS along with successful attributes. Some specifications such as FIFO queue and unordered list were separated into methods, classes, and modules in which these artifacts refine into a implementation-only module. The priority queue and scheduler classes did not contain a separate specification and implementation due to problems with modules (see Chapter 6). In addition, framing and sequences had some issues, but it was remedied by a workaround. Regardless of Dafny issues, the priority queue, FIFO queue, scheduler, task, and parameter classes were successfully verified. The data structures and scheduler API contain specifications and implementations. The API contained eighteen operations for managing tasks while the `xList` data structures have eight operations. The scheduler class required approximately seventy-seven verification conditions to prove correctness. As a result, Dafny performed adequately as a verification tool.

8.1 Future Work

The Dafny specification and implementation could refine into another artifact which may include executable code. The scheduler's operations were specified and implemented in which it may convert into another programming language. However, this is a difficult conversion because C is low-level and not an object-oriented language, but Dafny uses objects, classes, and methods. Another approach includes converting Dafny specifications to `VCC`. Both program verifiers contain similar annotations from sharing the same verification engine, Boogie. There is no process or tool to convert

Dafny to `VCC` and it requires knowledge and skill of both verifiers. The converting Dafny to `VCC` would accomplish having a C implementation of the scheduler API and `xList` data structures.

A long-term goal in formalizing FreeRTOS research is creating executable C code from a specification [6] [7] [8]. This allows several possibilities. One includes comparing a verified implementation to the original in which the formalized code-base may catch defects. In addition, traditional SQA activities can be applied to the verified code-base. This can assure further if defects were removed. Another possibility is comparing defect rates through fault injection to test if program verification or traditional SQA is more effective. Finally, a verified FreeRTOS port can built from the annotated code, but port-specific code would need developing to run on a test platform.

Research is needed to continue developing and improving verification tools. Program verifiers such as Dafny and `VCC` are considered "auto-active" verifiers in which the developer is responsible for creating the specifications and implementations [17]. A fully automatic program verifier which generates a formal specification from code and proves correctness does not exist. Some researchers question if an automatic verifier is feasible and possible [17]. Dafny contains some automation and heuristics for specifying a termination metric [22], but it needs improvement in decoupling the specification from the implementation [4].

BIBLIOGRAPHY

[1] R. Barry. About freertos, November 2013.

[2] R. Barry. Freertos - market leading rtos for embedded systems supporting 34 microcontroller architectures, September 2013.

[3] J. P. Bowen and M. G. Hinchey. Ten commandments of formal methods. *Computer*, 28(4):56–63, 1995.

[4] D. Bronish and B. W. Weide. A review of verification benchmark solutions using dafny. In *Proceedings of the 2010 Workshop on Verified Software: Theories, Tools, and Experiments*, 2010.

[5] F. P. Brooks Jr. No silver bullet-essence and accidents of software engineering. *IEEE computer*, 20(4):10–19, 1987.

[6] D. Déharbe, S. Galvão, and A. M. Moreira. Formalizing freertos: First steps. In *Formal Methods: Foundations and Applications*, pages 101–117. Springer, 2009.

[7] S. Divakaran and D. D'Souza. Z model of freertos.

[8] S. Divakaran and D. D'Souza. Z model of tasks lists in freertos.

[9] D. Galin. *Software quality assurance: from theory to implementation*. Pearson education, 2004.

[10] R. Goyette. An analysis and description of the inner workings of the freertos kernel. *Carleton University Department of Systems and Computer Engineering, SYSC5701: Operating System Methods for Real-Time Applications, 1st*, 2007.

[11] H. K. Harton. *Mechanical and Modular Verification Condition Generation for Object-Based Software*. PhD thesis, Clemson University, 2011.

[12] L. Herbert, K. R. M. Leino, and J. Quaresma. Using dafny, an automatic program verifier. In *Tools for Practical Software Verification*, pages 156–181. Springer, 2012.

[13] F. Huch. Verification of erlang programs using abstract interpretation and model checking. In *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, ICFP '99, pages 261–272, New York, NY, USA, 1999. ACM.

[14] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2004.

[15] R. Inam, J. Maki-Turja, M. Sjodin, S. M. Ashjaei, and S. Afshar. Support for hierarchical scheduling in freertos. In *Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference on*, pages 1–10. IEEE, 2011.

[16] C. Jones, P. O'Hearn, and J. Woodcock. Verified software: A grand challenge. *IEEE Computer*, 39(4):93–95, 2006.

[17] C. Le Goues, K. R. M. Leino, and M. Moskal. The boogie verification debugger (tool paper). In *Software Engineering and Formal Methods*, pages 407–414. Springer, 2011.

[18] K. R. M. Leino. Automating theorem proving with smt.

[19] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370. Springer, 2010.

[20] K. R. M. Leino. Developing verified programs with dafny. *Ada Lett.*, 32(3):9–10, Dec. 2012.

[21] K. R. M. Leino. Dafny: a language and program verifier for functional correctness, 2013.

[22] K. R. M. Leino and R. Monahan. Dafny meets the verification benchmarks challenge. In *Proceedings of the Third international conference on Verified software: theories, tools, experiments*, VSTTE'10, pages 112–126, Berlin, Heidelberg, 2010. Springer-Verlag.

[23] T. Nipkow et al. Getting started with dafny: A guide. *Software Safety and Security: Tools for Analysis and Verification*, 33:152, 2012.

[24] D. Plagge and M. Leuschel. Validating Z specifications using the ProB animator and model checker. In J. Davies and J. Gibbons, editors, *Integrated Formal Methods*, volume 4591 of *Lecture Notes in Computer Science*, pages 480–500. Springer-Verlag, 2007.

[25] Real Time Engineers Ltd. Freertos readme file, September 2013.

[26] Real Time Engineers Ltd. Official freertos ports. microcontrollers and compiler tool chains supported by freertos, September 2013.

[27] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 14th indian reprint edition, October 2002.

[28] A. G. Stephenson, D. R. Mulville, F. H. Bauer, G. A. Dukeman, P. Norvig, L. LaPiana, P. Rutledge, D. Folta, and R. Sackheim. Mars climate orbiter mishap investigation board phase i report, 44 pp. *NASA, Washington, DC*, 1999.

[29] C. Svec. Freertos architecture. the architecture of open source applications (volume 2): Freertos. In *The Architecture of Open Source Applications*. September 2013.

[30] A. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 3rd edition edition, December 2007.

[31] B. W. Weide, M. Sitaraman, H. K. Harton, B. Adcock, P. Bucci, D. Bronish, W. D. Heym, J. Kirschenbaum, and D. Frazier. Incremental benchmarks for software verification tools and techniques. In *Verified Software: Theories, Tools, Experiments*, pages 84–98. Springer, 2008.