Automated Verification of the FreeRTOS Scheduler in HIP/SLEEK

João F. Ferreira*†, Guanhua He* and Shengchao Qin*‡
*School of Computing, Teesside University, UK
†HASLab / INESC TEC, Universidade do Minho, Portugal
‡State Key Lab. for Novel Software Technology, Nanjing University, China
Emails: {jff, g.he, s.qin}@tees.ac.uk

Abstract

Automated verification of operating system kernels is a challenging problem, partly due to the use of shared mutable data structures. In this paper, we show how we can automatically verify memory safety and functional correctness of the task scheduler component of the FreeRTOS kernel using the verification system HIP/SLEEK. *We show how some of* HIP/SLEEK *features like user-defined predicates and lemmas make the specifications highly expressive and the verification process viable. To the best of our knowledge, this is the first code-level verification of memory safety and functional correctness properties of the FreeRTOS scheduler. The outcome of our experiment confirms that* HIP/SLEEK *can indeed be used to verify code that is used in production. Moreover, since the properties that we verify are quite general, we envisage that the same approach can be adopted to verify the scheduler of other operating systems.*

1. Introduction

In recent years, the number of embedded devices in the marketplace has increased substantially due to significant reductions in size and costs of microprocessors. As a result, the safety of the real-time operating systems (RTOSs) that are traditionally used by embedded devices is becoming increasingly important. The industry has already recognised the importance of providing safe and reliable RTOSs [1] and the academic community is actively working on tools and methods that can improve the current standards of software quality. In particular, the advances in theory and tool support have inspired industrial and academic researchers to join up in an international Grand Challenge (GC) in Verified Software [2], [3]. In the context of this international challenge, Jim Woodcock proposed the verification of FreeRTOS [4], a real-time, multitasking, preemptive operating system for embedded devices [5]. However, as Woodcock points out, FreeRTOS involves lots of pointers and the automatic verification of heap-manipulating programs is challenging. For that reason, Woodcock suggests the use of separation logic [6], which supports reasoning about shared mutable data structures.

In this paper, we take the FreeRTOS kernel as a case study and show how we can automatically verify the memory safety and functional correctness of its main component: the task scheduler. We use the verification system HIP/SLEEK, developed by Chin et al. [7], [8], which allows the combination of both separation (i.e. shape) and numerical (e.g. size) information. HIP/SLEEK also allows user-specified inductive predicates to appear in program specifications, making the specifications highly expressive. We only consider partial correctness: we prove, for example, that the next task chosen by the scheduler is the task that should be executed, but we do not guarantee that the task will eventually be chosen. To the best of our knowledge, we provide the first code-level verification of memory safety and functional correctness properties of the FreeRTOS scheduler.

In Section 2, we explain FreeRTOS and the main data structures used in its scheduler. The specification and verification process is presented in Section 3, followed by related work and concluding remarks.

2. FreeRTOS

FreeRTOS [4] is a real-time, multitasking, preemptive operating system for embedded devices. The most important concept in FreeRTOS is the concept of *task*. Each executing program is a task under the control of the operating system (some operating systems use the term *process*). In the presence of multiple tasks, the operating system has to decide which task to execute at any particular time. The part of the kernel responsible for switching tasks is the *scheduler*. FreeRTOS uses a fixed-priority scheduling policy, ensuring that at any given time, the processor executes the highest priority task of all those tasks that are currently ready to execute[1]. Among other properties, the scheduler also has to guarantee that only tasks that are ready to execute are actually executed.

FreeRTOS[2] is written mostly in C, with a few assembler functions that take care of architecture-specific details. There are four main C files that represent the kernel of FreeRTOS. The file *tasks.c* implements most of the scheduler functionalities, making use of the structures and functions defined in the file *list.c*. The file *queue.c* implements thread-safe queues that are used for inter-task communication and synchronisation. The file *croutine.c* implements coroutines, which are very simple and lightweight tasks that make a very limited use of stack. In this paper, we focus on the methods defined in the files *tasks.c* and *list.c*.

1. Note that FreeRTOS does not guarantee any deadlines for the execution of tasks. The only guarantee is that the highest priority task that is ready to execute will run as soon as possible.

2. The work described in this paper is based on version 6.1.1 of FreeRTOS. Have in mind that the data structures and the algorithms involved may have changed since that version.

2.1. Data structures

Lists. FreeRTOS provides a list API that is designed for the scheduler needs, but that can also be used by application code. Lists are used to organise tasks; for example, the scheduler maintains a list of tasks ready to execute and a list of tasks that are blocked. The data structure representing lists is called *xList* and is defined as follows:

```
typedef struct xLIST {
 volatile unsigned portBASE_TYPE
                        uxNumberOfItems;
 volatile xListItem * pxIndex;
 volatile xMiniListItem xListEnd;
} xList;
```

A list consists of a structure with three fields: the number of items in the list (*uxNumberOfItems*), a pointer to a list item (*pxIndex*), and a (mini) list item that contains the maximum possible item value, which is used as a marker (*xListEnd*). The type `portBASE_TYPE` is architecture dependent; in the context of this paper, it can be viewed as an unsigned integer. Note that lists only store pointers to structures of the type *xListItem*, whose definition is:

```
struct xLIST_ITEM {
 portTickType xItemValue;
 volatile struct xLIST_ITEM * pxNext;
 volatile struct xLIST_ITEM * pxPrevious;
 void * pvOwner;
 void * pvContainer;
};
typedef struct xLIST_ITEM xListItem;
```

Each list item holds a value (*xItemValue*), a pointer to the object (normally a task) that contains the list item (*pvOwner*), a pointer to the list in which the list item is placed (*pvContainer*), a pointer to the previous list item (*pxPrevious*), and a pointer to the next list item (*pxNext*). The existence of the pointers *pxPrevious* and *pxNext* suggests that lists are doubly-linked. As we will see later (section 3), lists are indeed *cyclic* doubly-linked lists. Note that the end marker, *xListEnd*, is a structure of the type *xMiniListItem*; the only difference between this structure and the structure *xListItem* is the omission of the fields *pvOwner* and *pvContainer*.

The list API provides several functions to manipulate lists. For example, the function used to initialise all the members of an *xList* structure is:

```
void vListInitialise( xList pxList );
```

After initialisation, the pointer *pxIndex* points to the field *xListEnd*, which is the only element of the list. Regarding the field *xListEnd*, its field *xItemValue* is set to the maximum possible value (*portMAX_DELAY*) and its pointers *pxNext* and *pxPrevious* are set to point to itself. As a result, the list can be seen as a doubly-linked list of size 1 (note, however, that the first field, *uxNumberOfItems*, contains the value 0, which is the number of elements different from the end marker). Figure 1 illustrates the state of a list immediately after initialisation. The three *xList* fields are laid out horizontally; the first holds the value of the variable *uxNumberOfItems*, which is zero; the second holds the pointer *pxIndex*; the third holds a structure of type *xMiniListItem*.

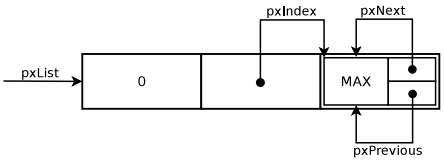

Fig. 1. xList data structure after initialisation

Another function, *vListInsertEnd*, is used to insert items into a list at the position following the item pointed by *pxIndex*. Its definition is:

```
void vListInsertEnd( xList pxList,
                     xListItem pxNewListItem )
{
 xListItem pxIndex;

 /* Insert a new list item into pxList, but
 rather than sort the list, makes the new list
 item the last item to be removed by a call to
 pvListGetOwnerOfNextEntry. This means it has
 to be the item pointed to by the pxIndex
 member.
 */
 pxIndex = pxList.pxIndex;

 pxNewListItem.pxNext = pxIndex.pxNext;
 pxNewListItem.pxPrevious = pxList.pxIndex;
 (pxIndex.pxNext).pxPrevious = pxNewListItem;
 pxIndex.pxNext = pxNewListItem;
 pxList.pxIndex = pxNewListItem;

 /* Remember which list the item is in. */
 pxNewListItem.pvContainer = pxList;

 (pxList.uxNumberOfItems)++;
}
```

Note how *pxIndex* is changed to point to the item that was just inserted. The relevance of *vListInsertEnd* will become apparent later, when we explain how the scheduler determines which task to run next. The API also offers a function, *vListRemove*, that is used to remove items from a list:

```
void vListRemove( xListItem pxItemToRemove );
```

Tasks. In FreeRTOS, a task is represented by a *task control block (TCB)*. TCBs are defined as follows[3]:

3. To simplify the presentation, we do not include fields specific to architectures that have a Memory Protection Unit (MPU), nor fields related with debugging. Also, the order of the fields has been rearranged.

```
typedef struct tskTaskControlBlock
{
 volatile portSTACK_TYPE *pxTopOfStack;
 portSTACK_TYPE          *pxStack;

 xListItem               xGenericListItem;
 xListItem               xEventListItem;

 unsigned portBASE_TYPE  uxPriority;
} tskTCB;

typedef void * xTaskHandle;
```

The first two fields of a TCB are related with the task's stack: *pxTopOfStack* points to the location of the last item placed on the task's stack, and *pxStack* points to the start of the stack.

Each TCB maintains two fields of the type *xListItem*: *xGenericListItem* is used to place the TCB in ready and blocked lists, and *xEventListItem* is used to place the TCB in event lists.

Finally, the field *uxPriority* represents the priority of the task, where 0 is the lowest priority.

FreeRTOS creates a special task —the *idle task*— when the scheduler starts (i.e., when the function *vTaskStartScheduler* is called). The idle task only executes when there are no other tasks able to do so, and it is responsible for freeing memory for tasks that have been deleted.

2.2. Scheduling

The scheduler starts when the function *vTaskStartScheduler* is called. The kernel can suspend and later resume a task many times during the task's lifetime. Because tasks are unaware of when they are suspended or resumed by the kernel, the scheduler has to guarantee that upon resumption a task has a context identical to that immediately prior to its suspension. The process of saving the context of a task being suspended and restoring the context of a task being resumed is called *context switching*. The context of a task is saved in its own stack.

The scheduler maintains several global variables that assist in the scheduling process. For example, the scheduler keeps track of the highest priority of which there are tasks ready to execute (*uxTopReadyPriority*). It also uses a pointer to the TCB that is currently running (*pxCurrentTCB*) and it maintains an array of lists, called *pxReadyTasksLists*, that contains lists of tasks ready to execute. Each list is associated to a different priority and the array is sorted in ascending order of priority; in other words, *pxReadyTasksLists[k]* is the list of tasks with priority k that are ready to run. To determine what is the next task to execute, the scheduler selects the highest k such that *pxReadyTasksLists[k]* is non-empty[4], and then uses a round-robin strategy. The next code listing shows how these *pxCurrentTCB* and *pxReadyTasksLists* are defined. The variable *configMAX_PRIORITIES* represents the maximum number of priorities that can be used.

```
static volatile unsigned portBASE_TYPE
                                uxTopReadyPriority;
tskTCB * volatile pxCurrentTCB = NULL;
static xList pxReadyTasksLists[
                            configMAX_PRIORITIES];
```

We now explain the dynamics of the FreeRTOS scheduler using a simple example. To simplify the presentation, we assume that we only have one list of tasks that are ready to execute (of priority *tskIDLE_PRIORITY*); also, we assume that the list has been initialized and is in the state shown in figure 1.

Adding new tasks. Suppose that two tasks, A and B, are created. The function that creates the tasks also adds them to the list of tasks ready to execute, using a function called *prvAddTaskToReadyQueue*. This function uses *vListInsertEnd* to add the tasks and, if necessary, it updates the variable *uxTopReadyPriority*. Hence, the state shown in figure 1 is changed to the state shown in figure 2. Because task B is the last task to be inserted, *pxIndex* points to task B's TCB. Note how the two TCBs are part of a doubly-linked list through the field *xGenericListItem*. Tasks are added to the list of tasks ready to execute when they are newly created or when they become unblocked.

Picking the next task. Each time a tick is generated, FreeRTOS saves the context of the task that is currently running and executes the function *vTaskSwitchContext*. This function selects the highest priority list that contains at least one task ready to execute. Once the list is identified, the task that follows the pointer *pxIndex* is chosen to run. The function responsible for that is *listGET_OWNER_OF_NEXT_ENTRY*[5], which is shown in figure 3.

Before, we mentioned that the function *vListInsertEnd* was relevant to the way in which the scheduler determines which task to run at a particular time. Indeed, given that the scheduler uses the macro *listGET_OWNER_OF_NEXT_ENTRY* to determine which task to execute next, an invariant of the scheduling process is that, for each list of ready tasks with the same priority, the TCB pointed by *pxIndex* will be the last one to execute (this is a consequence of using a cyclic-doubly linked list). Since the function *vListInsertEnd* sets *pxIndex* to point to the newly inserted TCB, this will be the last one to execute.

Going back to the example illustrated in figure 2, we can see that the scheduler would choose task A to run, since it follows the task pointed by *pxIndex*. Moreover, *listGET_OWNER_OF_NEXT_ENTRY* would change *pxIndex* to point to task A. So, if no tasks are added nor removed from the list before the next execution of *list-*

4. Here we use the term *non-empty* to qualify a list that has *at least one* TCB; that is, we do not consider *xListEnd* as a list item.

5. In fact, *listGET_OWNER_OF_NEXT_ENTRY* is defined as a C macro, but we define it here as a function. Also, we use HIP's notation so that the reader can see an example of a HIP program. Note that we use a dot for accessing fields, rather than C's arrow notation ->. We also use the keyword *ref* to express that the value of *pxTCB* is returned by reference.

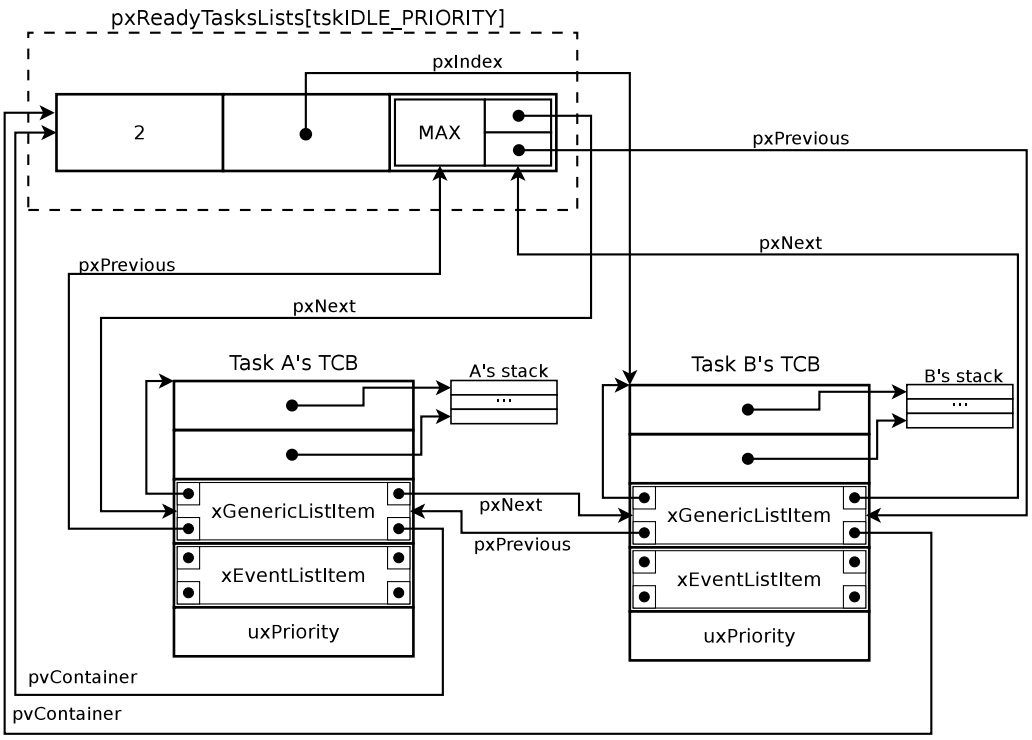

Fig. 2. *pxReadyTasksLists[tskIDLE_PRIORI-TY] after the creation of tasks A and B*

```
void listGET_OWNER_OF_NEXT_ENTRY
          (ref tskTCB pxTCB, xList pxList ) {
 xList pxConstList = pxList;

 /* Increment the index to the next item and
    return the item, ensuring we don't return
    the marker used at the end of the list. */

 pxConstList.pxIndex =
              (pxConstList.pxIndex).pxNext;
 if(pxConstList.pxIndex==pxConstList.xListEnd)
 {
  pxConstList.pxIndex =
              (pxConstList.pxIndex).pxNext;
 }
 pxTCB = (pxConstList.pxIndex).pvOwner;
}
```

Fig. 3. *listGET_OWNER_OF_NEXT_ENTRY* is used to switch executing tasks

GET_OWNER_OF_NEXT_ENTRY, the next task to run will be task B.

Removing tasks. In case a task blocks or is destroyed, function *vListRemove* is used to remove the task from the list of tasks that are ready to execute.

3. Specification and Verification

The verification of a scheduler involves many different types of properties. In this paper, we focus on memory safety and functional correctness properties. More specifically, some properties that we verify are:

- When tasks become ready to execute (newly created tasks, or recently unblocked tasks), the scheduler adds the tasks into the correct position of the "ready-tasks list";
- When switching context, the scheduler picks the *right* task to execute, i.e., the highest priority task that is ready to execute;
- When tasks are blocked or removed, the scheduler does not pick them to run;
- Memory safety: when the scheduler manipulates the data structures involved in scheduling, their shapes are maintained and there are no dereferencing of null pointers[6].

One of the main goals of this work is to investigate how we can automatically verify memory safety and functional correctness properties of a scheduler in a combined separation and numerical domain. We also want to test the suitability of the prototype HIP/SLEEK to verify code that is used in production. As a result, our main challenge is to model the data structures involved in the scheduling process and annotate FreeRTOS code with expressive specifications.

HIP/SLEEK is a state-of-the-art verification tool developed by Chin et al. [7], [8]. The front-end of the system is the Hoare-style forward verifier HIP, which takes user-defined predicates and program code with method specifications as input, and invokes a set of forward reasoning rules. The entailment checker SLEEK is used to systematically check that the precondition is satisfied at each call site, and the postcondition

6. It is important to note that in this work we do not verify if TCBs' stack and code pointers are valid. Invalid TCBs can affect context switching, but here we focus on ensuring that the scheduler makes *the right choices*.

is successfully verified for each method definition against the given precondition. Proof obligations related with the numeric domain are discharged to external automatic provers (e.g. MONA [9]).

3.1. On user-defined predicates

One advantage of HIP/SLEEK is the ability to define the shapes of data structures by separation logic combined with numerical (e.g. size) and bag (e.g. multi-set of values) information. Therefore, the specification language is expressive and powerful to capture not only memory safety properties, but also functional correctness properties.

For example, the shape of the lists used by the FreeRTOS scheduler can be captured by the shape predicate XLIST shown below.

$$\begin{array}{lcl} \texttt{XLIST(p)} & \equiv & \texttt{p}{\mapsto}\texttt{xList(\_,i,i)} \ * \ \texttt{DLS(i,q,i,q)} \\ & \vee & \texttt{p}{\mapsto}\texttt{xList(\_,i,e)} \ * \ \texttt{DLS(e,e1,i,f1)} \\ & & * \ \texttt{DLS(i,f1,e,e1)} \end{array}$$

Capitalised words refer to *shape predicates* and xList is a *data node*. So, XLIST(p) means that p is a pointer to a structure of the shape XLIST and a data node of the shape xList(_, i, e) represents an element of the datatype *xList* shown in section 2. The first field corresponds to the variable *uxNumberOfItems* and is left anonymously defined (_), since its value is not needed to define the shape of the structure. The other two fields, i and e, correspond to the fields *pxIndex* and *xListEnd*, respectively. It is important to note that we are treating the end marker as a normal *xListItem*, so that we can avoid explicit casts (these are not supported by HIP/SLEEK)[7]. The predicate XLIST is divided in two cases and depends on the predicate DLS, which captures the shape of doubly-linked list segments. The first disjunct captures the case where *pxIndex* and *xListEnd* point to the same entry (which is a doubly-linked list segment); the second disjunct captures the case where they point to different segments. The star ($*$) operator represents *separating conjunction* and ensures that its arguments reside in disjoint heaps (for more information about separation logic, see [6]). In both cases, the arguments that are passed to the DLS predicate ensure that the items are indeed in *cyclic doubly-linked lists*. The definition of the shape predicate DLS is:

$$\begin{array}{l} \texttt{DLS(p,pv,ob,ib)} \equiv \\ \quad \texttt{p}{\mapsto}\texttt{xListItem(\_,pv,ob,\_,\_)} \wedge \texttt{ib=p} \\ \vee \ \texttt{p}{\mapsto}\texttt{xListItem(\_,pv,t,\_,\_)} * \texttt{DLS(t,p,ob,ib)} \end{array}$$

In the definition of DLS, p is a pointer to the first element of the segment, pv is a pointer to the element preceding the segment, ob is a pointer to the element following the segment, and ib is a pointer to the last element of the segment. So, to express a cyclic doubly-linked list, we have to set p=ob and pv=ib.

These predicates can be directly used in HIP/SLEEK specifications to express, for example, that the result of a given function is a list of the shape XLIST, thus guaranteeing that the items are arranged as a cyclic doubly-linked list. However, since HIP/SLEEK supports the combination of shape information with numerical information, we can be more expressive. We can, for example, extend the shape predicates shown above with a natural number n representing the length of the list and with a bag B containing all the references to items in the list that are different from the end marker.

$$\begin{array}{l} \texttt{XLIST(p,n,B)} \equiv \\ \quad \texttt{p}{\mapsto}\texttt{xList(n,i,i)} * \texttt{DLS(i,q,i,q,n+1,B1)} \\ \quad \wedge \ \texttt{B} = \texttt{diff(B1,\{i\})} \\ \vee \ \texttt{p}{\mapsto}\texttt{xList(n,i,e)} * \texttt{DLS(e,e1,i,f1,m1,B1)} \\ \quad * \texttt{DLS(i,f1,e,e1,m2,B2)} \wedge \ \texttt{n=m1+m2-1} \\ \quad \wedge \ \texttt{B} = \texttt{diff(union(B1,B2),\{e\})} \end{array}$$

$$\begin{array}{l} \texttt{DLS(p,pv,ob,ib,n,B)} \equiv \\ \quad \texttt{p}{\mapsto}\texttt{xListItem(\_,pv,ob,\_,\_)} \wedge \texttt{ib=p} \wedge \ \texttt{n=1}\wedge\texttt{B=\{p\}} \\ \vee \texttt{p}{\mapsto}\texttt{xListItem(\_,pv,t,\_,\_)} * \texttt{DLS(t,p,ob,ib,n-1,B1)} \\ \quad \wedge \ \texttt{B} = \texttt{union(B1,\{p\})} \end{array}$$

The highlighted parts show the new numerical information. Note how in the definition of XLIST, the bag B is defined to exclude the end marker.

When FreeRTOS is compiled, the user has to define statically how many priorities the scheduler will support (by defining the variable *configMAX_PRIORITIES*). To simplify the verification process, we assume that we have exactly two different priorities. Also, because the version of HIP/SLEEK that we have used only has experimental support for arrays, we encapsulate the lists of tasks that are ready to execute in a user-defined data node:

```
data readyTskLists { xList l0; xList l1;}
```

The data node readyTskLists can be seen as an array with two lists, l0 and l1. We also provide a function *pxReadyTasksLists* that given a priority, returns the corresponding list of tasks ready to execute. In summary, we model the array *pxReadyTasksLists* as a partial function and we assume the existence of only two priorities.

3.2. On lemmas

User-defined predicates allow expressive descriptions of data structures with complex invariants. However, we may want to use properties of the data structures that are not directly obtained from the user-defined predicates. For example,

7. By treating the field *xListEnd* as a normal *xListItem*, our model adds two extra fields to the end marker: *pvContainer* and *pvOwner*. However, since these fields are never accessed for the end marker, this simplification is safe.

from the definition of DLS, the verification system cannot conclude immediately that two consecutive doubly-linked list segments can be merged into one doubly-linked list segment. To overcome this limitation, HIP/SLEEK allows the definition of lemmas that can be used to relate predicates beyond their original definitions. We can express lemmas using the reserved word coercion:

```
coercion appenddls
 DLS(self, pre1, ob2, ib2, n1 + n2, B3)∧B3=union(B1, B2)
 ←
 DLS(self, pre1, ob1, ib1, n1, B1) ∗
 DLS(ob1, ib1, ob2, ib2, n2, B2);
```

This lemma, called **appenddls**, states that two consecutive segments (note how ob1 and ib1 match) can be merged together. This lemma is necessary to verify the function *vListInsertEnd*. Another important lemma states that a doubly-linked list segment of size n can be decomposed into a doubly-linked list segment of size $n-1$ followed by a list item (for $n \geq 2$). We call this lemma **taildls** and define it as:

```
coercion taildls
        DLS(self, prev, ob, ib, n, B) ∧ n≥2
         →
        DLS(self, prev, ib, nib, n−1, B1) ∗
        ib↦xListItem(_, nib, ob, c, o) ∧
        B = union(B1, {ib});
```

This lemma is necessary to verify the function *vListRemove*, because the function links the element preceding the item to be removed with the element following it. Since the item to remove can be preceded by a DLS (as in the second case of *vListRemove*'s precondition shown below), we need a lemma that exposes the last element of the DLS.

3.3. Some examples of verified properties

We now show how some of the desired properties are verified, by discussing specifications that were successfully verified by HIP/SLEEK.

Manipulating lists. The scheduler relies on the list API, so, in order to verify properties of the scheduler, it is required that we verify first the methods used for manipulating lists. We have verified all the functions provided by the list API, but in this section, we only show some relevant functions together with their specifications. The first of these functions is *vListInitialise*, which is used to initialise lists. Using the predicates defined above, its formal specification can be written as follows:

```
void vListInitialise(xList pxList)
 requires pxList↦xList(_, _, _)
 ensures XLIST(pxList, 0, {})
{ ··· }
```

The keyword requires refers to the precondition and the keyword ensures refers to the postcondition. The specification expresses that, provided that the argument pxList is a pointer to an *xList* structure, the function guarantees that on termination pxList is of the shape XLIST(pxList, 0, {}). This simple example is included to illustrate how one aspect of memory safety is guaranteed: the function guarantees that the list is properly initialised with no items other than the end marker (as illustrated in figure 1).

As explained in section 2, function *vListInsertEnd* is relevant to the way in which the scheduler determines which task to run next. We can specify it as follows:

```
void vListInsertEnd(xList pxList,
                    xListItem pxNewListItem)
 requires XLIST(pxList, n, B) ∗
          pxNewListItem↦xListItem(_, _, _, _, o) ∗
          o↦tskTCB(_, _, pxNewListItem, _, _)
 ensures XLIST(pxList, n + 1, B1) ∧
         B1 = union({pxNewListItem}, B) ∧
         pxList.pxIndex = pxNewListItem;
{ ··· }
```

The precondition states that the argument *pxList* has to be an *XLIST* of size n with elements given by bag B. The postcondition assures that, on termination, *pxList* is an *XLIST* of size $n+1$ and the argument *pxNewListItem* is the new element. Moreover, it states that pxList.pxIndex is updated as expected. Note that by using separating conjunction, the precondition also states that the new element cannot already be an element of the list. If we look at the definition of the function shown in section 2, we see that there are no restrictions on the item being added to the list. As a result, if the TCB pointed by the field *pxIndex* is used as an argument, the shape of the list is destroyed! Since the list API can be used by application code, this can be seen as a potential serious problem. However, if our annotation is included and checked against all the calls, we can be sure that the problem will never arise. HIP/SLEEK is quite flexible, so if we want to be more precise about the shape of the data structures involved, we can specify *vListInsertEnd* alternatively as:

```
void vListInsertEnd(xList pxList,
                    xListItem pxNewListItem)
 requires XLIST(pxList, n, B) ∗
          pxNewListItem↦xListItem(_, _, _, _, o) ∗
          o↦tskTCB(_, _, pxNewListItem, _, _)
 ensures pxList↦xList(n + 1, pxNewListItem, e) ∗
         DLS(e, prev, pxNewListItem, ib, n1, B1) ∗
         DLS(pxNewListItem, ib, e, prev, n2, B2) ∧
         n = n1 + n2 − 2 ∧
         union(B, {pxNewListItem}) = union(B1, B2))
{ ··· }
```

This postcondition states explicitly that *pxNewListItem* can be seen as doubly-linked list segment adjacent to the doubly-linked list segment *pxList*. We could be even more specific and state in the postcondition that the TCB pointed by o is unchanged.

Note that the preconditions above include a reference to a *tskTCB* named *o* that is never used in the postconditions. We have to include it, because in the last line of the function, the field *pvOwner* is dereferenced (see section 2). This can be seen as another example of memory safety: HIP/SLEEK cannot verify functions that try to insert a list item with a null *pvOwner* field.

Finally, the function *vListRemove* can be specified as follows:

```
void vListRemove(xListItem pxItemToRemove)
 requires c↦xList(n, pxItemToRemove, e) ∗
          DLS(e, prev, pxItemToRemove, ib1, n1, B1) ∗
          DLS(pxItemToRemove, ib1, e, prev, n2, B2) ∧
          n = n1 + n2 − 1
  or c↦xList(n, p, e) ∗ DLS(e, prev, p, ib1, n1, B1) ∗
          DLS(p, ib1, pxItemToRemove, ib2, n2, B2) ∗
          DLS(pxItemToRemove, ib2, e, prev, n3, B3) ∧
          n = n1 + n2 + n3 − 1
 ensures XLIST(c, n − 1, _) ∗
          pxItemToRemove↦xListItem(_, _, _, _, _);
{ ··· }
```

The cases in the precondition arise because the item to be removed can be the one pointed by the field *pxIndex*. The postcondition guarantees that the size of the list is decreased and that the list and the item to remove reside in separate parts of memory.

Picking the next task. The function shown in figure 3 is where the task that runs next is selected. It can be specified as follows:

```
void list_GET_OWNER_OF_NEXT_ENTRY(ref tskTCB pxTCB,
                                  xList pxList)
  requires XLIST(pxList, _, B)
  ensures XLIST(pxList, _, B) ∗
          pxTCB′↦tskTCB(_, _, gli, _, _) ∧ gli ∈ B
{ ··· }
```

The primed variable pxTCB′ denotes the value of the pointer pxTCB after execution of the function. The specification expresses that given an argument list pxList with the tasks contained in the bag B, the return value pxTCB′ is guaranteed to be a task that is in the bag B. The field gli in the specification refers to the *xListItem* field that is used to place a TCB in a list (as mentioned before in section 2). This is another example of memory safety certification: the pointer to the task chosen by the scheduler to execute next will certainly point to a task in the list of tasks ready to execute and will never point to the end marker.

Although *list_GET_OWNER_OF_NEXT_ENTRY* is the function responsible for the change of the running TCB, it is called by the function *vTaskSwitchContext*, which is executed after each clock tick. Assuming that l0 and l1 are lists of tasks ready to execute with priorities 0 and 1, respectively, we can specify *vTaskSwitchContext* as:
The specification states that if the highest priority of the tasks ready to execute is 0 (i.e., *uxTopReadyPriority* is 0), then the list that is chosen is l0. Otherwise, l1 is chosen. It has to be said that to simplify the verification, we have changed the function to return the list that is chosen; in the original code, the type of the function is void.

```
xList vTaskSwitchContext()
 requires rtl↦readyTskLists(l0, l1) ∗
          XLIST(l0, _, _) ∗ XLIST(l1, _, _) ∧
          uxTopReadyPriority = 0
 ensures res = l0
 requires rtl↦readyTskLists(l0, l1) ∗
          XLIST(l0, _, _) ∗ XLIST(l1, _, _) ∧
          uxTopReadyPriority = 1
 ensures res = l1
{ ··· }
```

Adding new tasks. The function used to add new tasks to the list of tasks ready to execute is *prvAddTaskToReadyQueue*, which can be specified as follows:

```
void prvAddTaskToReadyQueue(ref tskTCB pxTCB)
 requires pxTCB↦tskTCB(_, _, gli, _, 0) ∗
          gli↦xListItem(_, _, _, _, pxTCB) ∗
          rtl↦readyTskLists(l0, l1) ∗
          XLIST(l0, _, _) ∗ XLIST(l1, _, _)
 ensures DLS(gli, ib, e, prev, _, _) ∗
          l0↦xList(_, gli, e) ∗
          DLS(e, prev, gli, ib, _, _) ∧
          uxTopReadyPriority′≥0
 requires pxTCB↦tskTCB(_, _, gli, _, 1) ∗
          gli↦xListItem(_, _, _, _, pxTCB) ∗
          rtl↦readyTskLists(l0, l1) ∗
          XLIST(l0, _, _) ∗ XLIST(l1, _, _)
 ensures DLS(gli, ib, e, prev, _, _) ∗
          l1↦xList(_, gli, e) ∗
          DLS(e, prev, gli, ib, _, _) ∧
          uxTopReadyPriority′≥1
{ ··· }
```

The specification states that the TCB is added to the list associated with its priority and the global variable *uxTopReadyPriority* is updated accordingly.

Removing tasks. As explained before, the scheduler uses *vListRemove* to remove a task from the list of tasks ready to execute. This function is described above.

4. Related Work

Much work has been done on the verification of operating systems; see [10] for an overview on the topic. Here, we focus on RTOSs, on separation logic, and on FreeRTOS. Verification of RTOSs has been identified as one of the grand challenges in software verification [5]. A number of tools have been developed to verify real system tools. A notable project on verification of RTOSs is ASTRÉE [11], which proves no runtime error in the electric flight-control code for the A380. ASTRÉE detects numeric error and overflows, but with the restriction that no dynamic memory allocation is in the program code. Other tools, like SLAM [12] and BLAST [13], have been used to ensure that device drivers satisfy the requirement of system APIs. However, these tools do not handle memory safety properties. Various other RTOSs claim to be formally

verified, such as OpenComRTOS has been verified by Verhulst et al. [14]. Baumann et al. [15] uses deductive techniques to verify the correctness of the PikeOS system. However, these works only focus on the functional correctness of the systems, but not the memory safety properties.

Separation logic has been adopted by a number of tools to verify the memory safety of system code, such as SMALLFOOT [16], SPACEINVADER [17] and THOR [18]. However, most of these tools support only a limited set of predicates, which limits the capability to verify the full functional correctness of system code. Finally, a closely related work in progress is reported in [19], where the authors discuss different approaches that can be used to verify FreeRTOS. Particularly relevant is their use of Verifast [20], a verification system based on separation logic. Although they do not present any details or annotations, it would be interesting, as future work, to compare their annotations with ours.

5. Concluding Remarks

This paper shows how the combination of shape and numerical information can be used to specify and verify the scheduler of FreeRTOS. The results confirm that HIP/SLEEK can indeed be used to automatically verify important properties of production code. To the best of our knowledge, this is the first code-level verification of memory safety and functional correctness properties of the FreeRTOS scheduler. Since the properties that we verify are quite general, we envisage that the same approach can be adopted to verify the scheduler of other operating systems.

5.1. Future Work

A direction that we want to pursuit is related with inference and scalability. Although the specifications written so far have been supplied by us, recent developments [21], [22] in HIP/SLEEK will allow the automatic inference of properties, making our approach more scalable.

A challenge that we foresee is extending HIP/SLEEK to support overlaid structures [23], which are structures that contain nodes for multiple data structures and these links are intended to be used at the same time. For example, in FreeRTOS, a task can be simultaneously in two lists and when it is removed from one of them, it also has to be removed from the other (an example is the function *xTaskRemoveFromEventList*).

Finally, as we verify other components of FreeRTOS, we will certainly gain more in-depth knowledge about the system. This means that specifications will possibly be refined and improved. By tackling some of these challenges, we hope to develop new theory results and to extend HIP/SLEEK.

Acknowledgements

We would like to thank the anonymous reviewers, who provided valuable feedback. This work was supported by the EPSRC Project EP/G042322.

References

[1] "The SafeRTOS™ project website," URL: http://www.freertos.org/safertos.html.
[2] T. Hoare, "The verifying compiler: A grand challenge for computing research," *J. ACM*, vol. 50, 2003.
[3] C. Jones, P. O'Hearn, and J. Woodcock, "Verified software: A grand challenge," *Computer*, vol. 39, 2006.
[4] "The FreeRTOS™ project website," URL: http://www.freertos.org.
[5] J. Woodcock, "Grand challenge in software verification," in *SBMF*, 2008.
[6] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *LICS*, 2002.
[7] H. H. Nguyen, C. David, S. Qin, and W.-N. Chin, "Automated verification of shape and size properties via separation logic," in *VMCAI*, 2007.
[8] W.-N. Chin, C. David, H. H. Nguyen, and S. Qin, "Automated verification of shape, size and bag properties via user-defined predicates in separation logic," *Science of Computer Programming*, vol. In Press, 2010.
[9] N. Klarlund and A. Møller, *MONA Version 1.4 User Manual*, 2001.
[10] G. Klein, "Operating system verificationan overview," *Sadhana*, vol. 34, pp. 27–69, 2009, 10.1007/s12046-009-0002-4. [Online]. Available: http://dx.doi.org/10.1007/s12046-009-0002-4
[11] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "A static analyzer for large safety-critical software," in *PLDI*, 2003.
[12] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers," in *EuroSys*, 2006.
[13] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, "Abstractions from proofs," in *POPL*, 2004.
[14] B. H. C. Sputh, O. Faust, E. Verhulst, and V. Mezhuyev, "Opencomrtos: A runtime environment for interacting entities," in *CPA*, 2009.
[15] C. Baumann, B. Beckert, H. Blasum, and T. Bormer, "Formal verification of a microkernel used in dependable software systems," in *SAFECOMP*, 2009.
[16] J. Berdine, C. Calcagno, and P. W. O'Hearn, "Smallfoot: Modular automatic assertion checking with separation logic," in *FMCO*, 2005.
[17] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn, "Scalable shape analysis for systems code," in *CAV*, 2008.
[18] S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay, "Thor: A tool for reasoning about shape and arithmetic," in *CAV*, 2008.
[19] J. T. Mühlberg and F. Leo, "Verifying freertos: from requirements to binary code," in *11th International Workshop on Automated Verification of Critical Systems (AVoCS)*, vol. CS-TR-1272. Newcastle University, September 2011. [Online]. Available: https://lirias.kuleuven.be/handle/123456789/333258
[20] B. Jacobs, J. Smans, and F. Piessens, "A quick tour of the verifast program verifier," in *APLAS*, ser. Lecture Notes in Computer Science, K. Ueda, Ed., vol. 6461. Springer, 2010, pp. 304–311.
[21] S. Qin, G. He, C. Luo, and W.-N. Chin, "Loop invariant synthesis in a combined domain," in *ICFEM*, 2010.
[22] S. Qin, C. Luo, W.-N. Chin, and G. He, "Automatically refining partial specifications for program verification," in *FM*, 2011.
[23] O. Lee, H. Yang, and R. Petersen, "Program analysis for overlaid data structures," in *CAV*, 2011.