**Noname manuscript No.**
(will be inserted by the editor)

# Automated Verification of the FreeRTOS Scheduler in Hip/Sleek

**João F. Ferreira, Cristian Gherghina, Guanhua He, Shengchao Qin and Wei-Ngan Chin**

Received: date / Accepted: date

**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 properties 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 components of other operating systems.

**Keywords** FreeRTOS · separation logic · automated verification · operating systems · embedded systems · task scheduler · HIP/SLEEK

This work was supported in part by the EPSRC project EP/G042322 and NNSFC project 61373033.

João F. Ferreira
School of Computing, Teesside University
HASLab / INESC TEC, Universidade do Minho

Christian Gherghina
Singapore University of Technology and Design

Guanhua He
School of Computing, Teesside University

Shengchao Qin (Corresponding author)
Shenzhen University and Teesside University
E-mail: shengchao.qin@gmail.com

Wei-Ngan Chin
National University of Singapore

## 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 [2] 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 [13,15]. In the context of this international challenge, Jim Woodcock proposed the verification of FreeRTOS [1], a real-time, multitasking, preemptive operating system for embedded devices [33]. However, as Woodcock points out, FreeRTOS involves lots of pointers and the automatic verification of heap-manipulating programs is challenging [30]. For that reason, Woodcock suggests the use of separation logic [29], 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, 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 (i.e., temporal properties and other properties like thread safety on shared mutable data structures are not considered). To the best of our knowledge, we provide the first code-level verification of memory safety and functional correctness properties of the FreeRTOS scheduler.

We start in Section 2 by describing how FreeRTOS is structured and by explaining what are the main data structures involved in scheduling. In Section 3 we give an overview of the Hip/Sleek verification system, and in Section 4, we give an overview of the specification and verification process, focusing in particular on, how the main data structures used in the FreeRTOS scheduler are modelled and how relevant properties of functional correctness and memory safety are specified. In Section 5, we discuss related work. We finish in Section 6 with a discussion on what was achieved, on what we have learnt from this experience, and on what we have planned for the next steps. We also discuss briefly the main challenges that we foresee.

## 2 FreeRTOS

FreeRTOS [1] 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 the C programming language, with a few assembler functions that take care of architecture-specific details. There are four main C files that represent the kernel of FreeRTOS. FreeRTOS source code is distributed under a free software license and is structured as shown in Figure 1. Most of the architecture-independent code is in the *Source* directory. 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*. FreeRTOS supports many different architectures; the architecture-dependent code is in the directory *portable*. Although memory allocation and deallocation are specifically defined for some of the architectures, the directory *MemMang* contains several C implementations that are portable for most of the architectures.

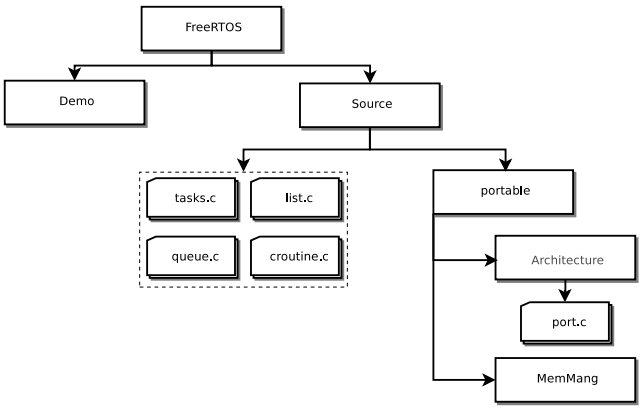



**Fig. 1** Structure of FreeRTOS Source Code

### 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 a key part of the scheduler, because they 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

[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.

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 4), 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 the following five public functions:

```
void vListInitialise(xList *pxList);
void vListInitialiseItem(xListItem *pxItem);
void vListInsert(xList *pxList,
                 xListItem *pxNewListItem);
void vListInsertEnd(xList *pxList,
                    xListItem *pxNewListItem);
void vListRemove(xListItem *pxItemToRemove);
```

The function *vListInitialise* initialises all the members of an *xList* structure. This function must be called before a list is used. 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 2 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*.

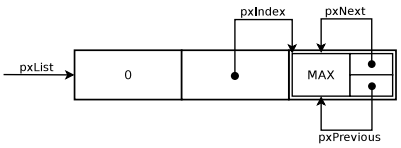



**Fig. 2** xList structure after initialisation

The function *vListInitialiseItem* sets the container of the given item to *NULL*, guaranteeing that the item is not recorded as being on a list. The function *vListInsert* inserts an item into a list in ascending item value order. Figure 3 illustrates how the list shown in Figure 2 would like if an *xListItem* A was inserted into it (we omit the pointer *pxList* for simplicity).

If an *xListItem* B with an item value greater than A's item value was to be inserted into the list shown in Figure 3, then it would be placed after A, as illustrated in Figure 4. Note how insertion guarantees that the doubly-linked list is cyclic.

The function *vListInsertEnd* inserts an item into a list at the position following the item pointed by *pxIndex*. Its definition is shown in Figure 5.
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. Suppose that we have the list shown in Figure 3 and we insert an *xListItem* B with an item value greater than A's item value using the function *vListInsertEnd*. Figure 6 illustrates the resulting list. Note how the pointer *pxIndex* is updated. The relevance of this function will become apparent later, when we explain how the scheduler determines which task to run.

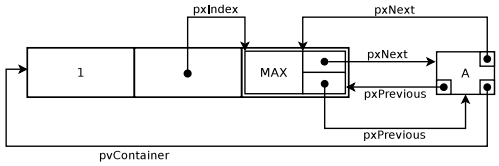



**Fig. 3** xList structure after insertion of the *xListItem* A

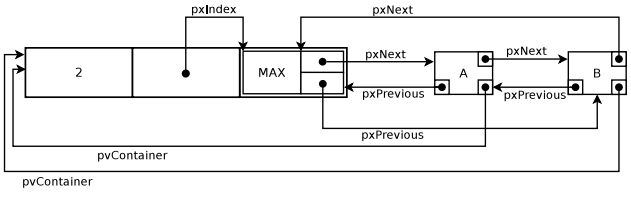



**Fig. 4** xList structure after insertion of the *xListItem* B

```
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)++;
}
```

**Fig. 5** Definition of the function *vListInsertEnd*

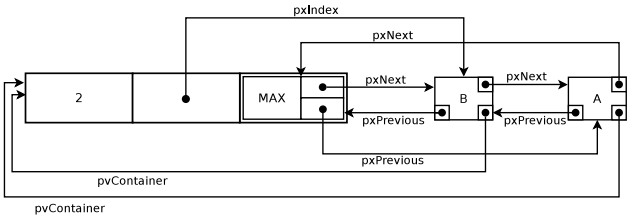



**Fig. 6** xList structure after insertion of the *xListItem* B (using *vListInsertEnd*)

Finally, the function *vListRemove* removes an item from a list, updating the pointer *pxIndex* if necessary. For example, if we remove the item B from the list shown in Figure 6, we get the list shown in Figure 3 (*pxIndex* is set to point to the previous item in the list).

***Tasks*** In FreeRTOS, a task is represented by a *task control block (TCB)*. TCBs are defined as shown in Figure 7[3].
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

[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;
```

**Fig. 7** Definition of task control blocks

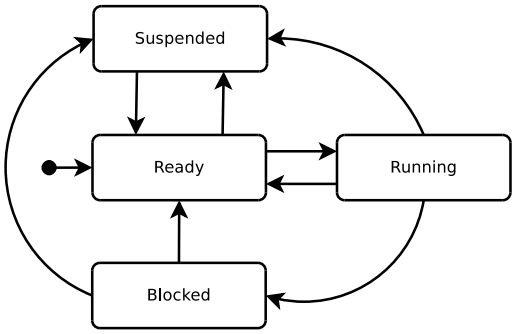



**Fig. 8** Valid task state transitions (for more details, see [1])

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 named *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.

Tasks can be in one of four states:

**Running** the task is currently using the processor;
**Ready** the task is ready to execute, but not currently running because a different task of equal or higher priority is running;
**Blocked** the task is waiting for an event. Blocked tasks are not available for scheduling;
**Suspended** the task is not available for scheduling. Tasks will only enter or exit the suspended state when explicitly commanded to do so.

For completeness, we show in Figure 8 the valid task state transitions.

Rather than associating with each task a flag expressing representing its state (e.g., a "Running" flag), the scheduler maintains several global lists that agglomerate tasks that are in the same state.

## 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 2.

***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 2 is changed to the state shown in Figure 9. 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.

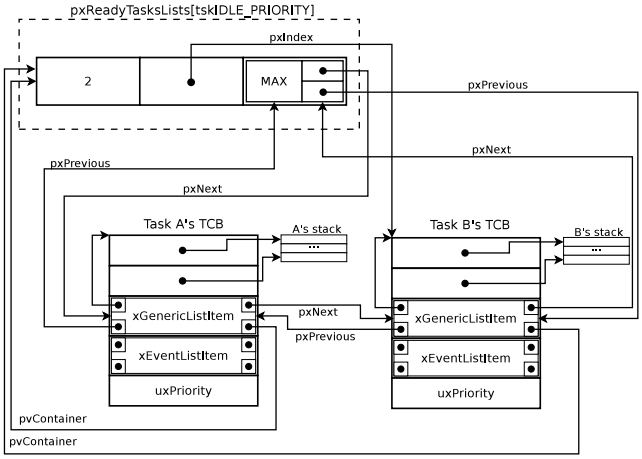



**Fig. 9** *pxReadyTasksLists[tskIDLE_PRIORITY] after the creation of tasks A and B*

***Picking the next task*** Each time a clock 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 10.

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 named *listGET_OWNER_OF_NEXT_ENTRY* to determine which task to execute next, an invariant of the scheduling process is:

*For each list of ready tasks with the same priority, the TCB pointed by* pxIndex *will be the last one to execute.*

[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.

```
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. 10** *listGET_OWNER_OF_NEXT_ENTRY* is used to switch executing tasks

(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 9, we can see that the scheduler would choose task A to run, since it follows the task pointed by *pxIndex*. Moreover, the macro *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 *listGET_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 The Hip/Sleek Verification System

The Hip/Sleek verification system developed by Chin et al. [8,23] is aimed at automatic verification of memory safety and functional correctness of heap-manipulating programs. The front-end of the system is the Hoare-style forward verifier Hip, which takes user-defined predicates, program code, and program specifications (loop invariants, method pre/post) as input, and invokes a set of forward reasoning rules to symbolically execute the program (starting from the initial abstract state specified by the precondition). The backend entailment prover Sleek is used to automatically prove formulae entailment (proof obligations generated by Hip during its forward verification). Key scenarios where Sleek is invoked include (1) systematically check that the precondition is satisfied at each call site, and (2) the post-condition 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 [16]).

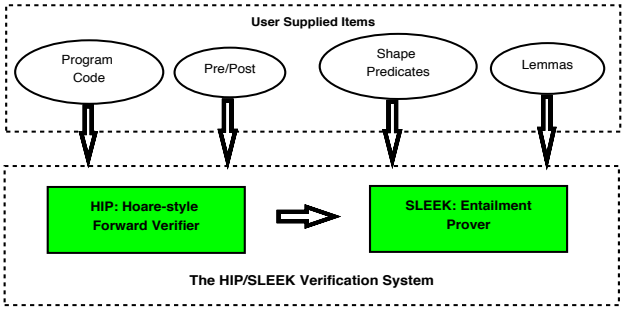



**Fig. 11** Overview of Hip/Sleek

The overall structure of Hip/Sleek is shown in Figure 11. In what follows, we shall illustrate user-defined shape predicates via examples, and briefly introduce the Hip verifier and the Sleek prover. The advanced feature about lemmas is delayed to a later section.

### 3.1 User-defined predicates

For better flexibility and expressivity, Hip/Sleek allows users to define inductive shape predicates to leverage both shape and pure properties and to capture their desired level of program correctness (to be verified). For example, with a singly-linked list structure defined as

$$\texttt{data node \{ int val; node next; \}}$$

a user interested in pointer-safety may define a list shape predicate (as in [7,11]):

$$\texttt{list(root)} \equiv (\texttt{root=null}) \vee (\exists \texttt{i,q} \cdot \texttt{root} \mapsto \texttt{node(i,q)} * \texttt{list(q)})$$

Note that in the inductive case, the separation conjunction $*$ (for more information about separation logic, see [29]) ensures that two heap portions (the head node and the tail list) are domain-disjoint. The parameter `root` for the predicate is the root pointer referring to the data structure.

Hip/Sleek allows the use of numerical information in shape predicates. This means that if the user is interested in tracking also the length of a list to analyse quantitative measures, the following shape predicate can be defined

$$\texttt{ll(root,n)} \equiv (\texttt{root=null} \wedge \texttt{n=0}) \vee (\texttt{root} \mapsto \texttt{node(\_,q)} * \texttt{ll(q,m)} \wedge \texttt{n=m+1})$$

Note that unbound variables, such as q and m, are implicitly existentially quantified, and _ is used to denote an existentially quantified anonymous variable. This predicate may be changed to capture information about the content of lists, to support a higher level of correctness with a multi-set (bag) property:

$$\texttt{llB}(\texttt{root},\texttt{S}) \equiv (\texttt{root}{=}\texttt{null} \wedge \texttt{S}{=}\emptyset) \vee (\texttt{root} \mapsto \texttt{node}(\texttt{v},\texttt{q}) * \texttt{llB}(\texttt{q},\texttt{S}_1) \wedge \texttt{S}{=}\{\texttt{v}\} \sqcup \texttt{S}_1)$$

The length of the list is implicitly captured by the cardinality |S|. The operator ⊔ denotes bag union. A further strengthening can capture also the sortedness property:

$$\texttt{sllB}(\texttt{root},\texttt{S}) \equiv (\texttt{root}{=}\texttt{null} \wedge \texttt{S}{=}\emptyset) \vee (\texttt{root} \mapsto \texttt{node}(\texttt{v},\texttt{q}) * \texttt{sllB}(\texttt{q},\texttt{S}_1) \wedge \texttt{S}{=}\{\texttt{v}\} \sqcup \texttt{S}_1 \wedge (\forall \texttt{x} \in \texttt{S}_1 \cdot \texttt{v} \leq \texttt{x}))$$

Therefore, users can provide predicate definitions with respect to various correctness levels and program properties, which can be as simple as normal lists or as complicated as AVL trees, depending on their requirements. These predicates are non-trivial to be defined but can be reused multiple times for specifications of different methods. Hence, efforts involved in such predicate design are often significantly amortised.

User-defined shape predicates can be used to specify program specifications such as loop invariants and method specifications. For instance, in Figure 12, the predicates llB and sllB are used to specify the methods insert_sort and insert (Line 3, 4, 12, 13):

```
 1 data node { int val; node next; }
 2 node insert_sort(node x)
 3  requires llB(x,S) ∧ |S|≥1
 4  ensures  sllB(res,T) ∧ T=S
 5 { if (x.next == null) return x;
 6   else { node s = x.next;
 7     node r = insert_sort(s);
 8     return insert(r, x);
 9   }
10 }
11 node insert(node r, node x) {
12  requires sllB(r,S)*x↦node(v,_)
13  ensures  sllB(res,T) ∧ T=S⊔{v}
14   if (r == null) {
15     x.next = null; return x;
16   } else if (x.val <= r.val) {
17     x.next = r; return x;
18   } else {
19     r.next = insert(r.next, x);
20     return r;
21   }
22 }
```

**Fig. 12** The insertion sort program for lists.

| | | |
|---|---|---|
| *spred* | ::= | $c\langle v^* \rangle \equiv \Phi$ **inv** $\pi$ |
| *mspec* | ::= | **requires** $\Phi_{pr}$; **ensures** $\Phi_{po}$ |
| $\Phi$ | ::= | $\bigvee(\exists v^* \cdot \kappa \wedge \pi)^*$ |
| $\pi$ | ::= | $\gamma \wedge \phi$ |
| $\gamma$ | ::= | $v_1{=}v_2 \mid v{=}\texttt{null} \mid v_1{\neq}v_2 \mid v{\neq}\texttt{null} \mid \gamma_1 \wedge \gamma_2$ |
| $\kappa$ | ::= | $\texttt{emp} \mid v{::}c\langle v^* \rangle \mid \kappa_1 * \kappa_2$ |
| $\Delta$ | ::= | $\Phi \mid \Delta_1 \vee \Delta_2 \mid \Delta \wedge \pi \mid \Delta_1 * \Delta_2 \mid \exists v \cdot \Delta$ |
| $\phi$ | ::= | $\varphi \mid b \mid a \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \exists v \cdot \phi \mid \forall v \cdot \phi$ |
| $a$ | ::= | $s_1{=}s_2 \mid s_1{\leq}s_2$ |
| $b$ | ::= | $\texttt{true} \mid \texttt{false} \mid v \mid b_1{=}b_2$ |
| $s$ | ::= | $k^{\texttt{int}} \mid v \mid k^{\texttt{int}} \times s \mid s_1{+}s_2 \mid -s \mid max(s_1,s_2) \mid min(s_1,s_2) \mid \lvert B \rvert$ |
| $\varphi$ | ::= | $v{\in}\texttt{B} \mid \texttt{B}_1{=}\texttt{B}_2 \mid \texttt{B}_1 \sqsubset \texttt{B}_2 \mid \forall v{\in}\texttt{B} \cdot \phi \mid \exists v{\in}\texttt{B} \cdot \phi$ |
| B | ::= | $\texttt{B}_1 \sqcup \texttt{B}_2 \mid \texttt{B}_1 \sqcap \texttt{B}_2 \mid \texttt{B}_1 - \texttt{B}_2 \mid \{\} \mid \{v\}$ |

**Fig. 13** The specification language

For completeness, we include the grammar that defines the syntax of inductive shape predicates (*spred*) in Figure 13.

### 3.2 The Hip System

Hip is a separation logic based automated verification system for a C-like imperative language, able to modularly verify heap-manipulating programs. The system can handle programs with complex data structures. It accepts abstract descriptions for such structures in the form of inductive predicates shown in the previous subsection.

The grammar of the C-like imperative language supported by Hip is shown in Figure 14. A program comprises a list of type declarations ($tdecl^*$) and a list of method declarations ($meth^*$). We use the superscript $^*$ to denote a list of items; for example $v^*$ denotes a list of variables, $v_1, .., v_n$. With regard to the terminal symbols, $c$ denotes the name of a user-defined data type, $v, v_1, v_2$ stand for variable names, $mn$ represents a method name, $k$ is a numeric constant, and $f$ denotes a field name. The language supports data type declaration via *datat*, and shape predicate definition via *spred*.

Given annotations for each method/loop with one or more pre/post conditions, the Hip verifier constructs a set of obligations in the form of implication checks (entailments) between pairs of formulae which are then sent to the backend Sleek prover to be discharged. Note that method specifications are denoted as *mspec* in Figure 13. The formulae $\Phi_{pr}$ and $\Phi_{po}$ in the method specification "**requires** $\Phi_{pr}$; **ensures** $\Phi_{po}$" denote the precondition and postcondition of the method, respectively. The specification language allows rich specifications that contain both heap constraints expressed as separation logic formulae and several different logic

$$
\begin{array}{lll}
P & ::= & tdecl^*\ meth^* \\
tdecl & ::= & datat \mid spred \\
datat & ::= & \texttt{data}\ c\ \{\ field^*\ \} \\
field & ::= & type\ v \\
type & ::= & c \mid \tau \\
\tau & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{float} \mid \texttt{void} \\
meth & ::= & type\ mn\ ((\texttt{ref}\ type\ v)^*, (type\ v)^*)\ \texttt{where}\ mspec\ \{e\} \\
e & ::= & \texttt{null} \mid k^{\tau} \mid v \mid v.f \mid v{:=}e \mid v_1.f{:=}v_2 \mid \texttt{new}\ c(v^*) \\
 & & \mid e_1; e_2 \mid type\ v;\ e \mid mn(v^*) \mid \texttt{if}\ (v)\ e_1\ \texttt{else}\ e_2
\end{array}
$$

**Fig. 14** The core imperative language supported by Hip

fragments like Presburger arithmetic, bags, and lists for the pure constraints. By making use of set/bag solvers, the user can also encode reachability conditions as a set/bag of values that can be collected from some given data structure. Such conditions are then automatically discharged by Hip.

The forward verification process conducted by Hip is essentially a symbolic execution process. Given a Hoare-style triple $\{\Delta_1\}\texttt{e}\{\Delta_2\}$, the Hip verifier starts from the abstract pre-state $\Delta_1$, symbolically executes the program code e to generate an abstract post-state $\Delta'$ and invokes Sleek to automatically prove that $\Delta'$ entails $\Delta_2$. During the symbolic execution, the Sleek prover may be invoked many times to discharge generated proof obligations. The Hip verifier conducts verification in a modular way, i.e. it verifies each method once against its specifications. Methods in a program are verified in a (bottom-up) order according to the program's call-dependency graph, starting from the methods in the leaves of the graph. For each method, the verification starts from its precondition, computes a post-state by symbolically executing the method body, and then proves the generated state entails the expected postcondition. Once a method is verified against its specifications, when the method is invoked, the verifier only needs to check that the abstract state at the call site establishes the precondition. If it does, it can assume the postcondition at the end of the method call. A more detailed introduction to the verification process via Hip can be found at Chin et. al [8].

### 3.3 The Sleek Prover

The Hip verifier relies on the Sleek prover in order to discharge verification conditions. Sleek is a fully automatic prover for separation logic with frame inferring capability. It takes two heap states as input (say $\Delta_A$ and $\Delta_C$) represented by separation formulae, and checks if one formula $\Delta_A$ (the antecedent) entails the other $\Delta_C$ (the consequent): $\Delta_A \vdash \Delta_C * \Delta_R$ . The antecedent may cover more heap states than the consequent, so a residual heap state ($\Delta_R$) which represents the frame condition can be returned by the prover. This residual heap state will include the pure state of the antecedent. Sleek also supports instantiation of logical variables that appear during the entailment as existential variables in the consequent. As part of the implication check, Sleek discharges the heap obligations (the obligations pertaining to the shape of data structures) and translates the remaining pure obligations to pure constraints that can be discharged by other off-the-shelf theorem provers. The list of possible pure provers includes Omega , MONA , CVC Lite, Z3, and Isabelle.

Apart from handling disjunctions and existential quantifiers and dealing with the case when the consequent formula has an empty heap part, there are three key steps that may take place in a Sleek proof, namely, (1) matching up heap nodes/predicates from the antecedent and the consequent, (2) unfolding a shape predicate in the antecedent, and (3) folding against a shape predicate in the consequent.

As an example, the matching step takes place in the following entailment proof:

$$\texttt{ll(x,n)} \wedge \texttt{n>1} \ \vdash\ \exists \texttt{m} \cdot \texttt{ll(x,m)} \wedge \texttt{m>0}$$

leading to the pure entailment

$$\texttt{n>1} \ \vdash\ \texttt{n>0}$$

The matching step also takes place in the following slightly different entailment proof:

$$\texttt{ll(x,n)} \wedge \texttt{n>1} \ \vdash\ \texttt{ll(x,m)} \wedge \texttt{m>0}$$

leading to the pure entailment

$$\underline{\texttt{n=m}} \wedge \texttt{n>1} \ \vdash\ \texttt{n>0}$$

Note that the underlined part denotes an implicit instantiation of the free variable m.

As an example of the unfolding step, let us look at the following entailment check:

$$\texttt{ll(x,n)} \wedge \texttt{n>1} \ \vdash\ \exists \texttt{r,m} \cdot \texttt{x}{\mapsto}\texttt{node(\_,r)} * \texttt{ll(r,m)} \wedge \texttt{m>0}$$

An unfolding to the ll predicate in the antecedent leads to

$$\exists \texttt{q} \cdot \texttt{x}{\mapsto}\texttt{node(\_,q)} * \texttt{ll(q,n-1)} \wedge \texttt{n>1} \ \vdash\ \exists \texttt{r,m} \cdot \texttt{x}{\mapsto}\texttt{node(\_,r)} * \texttt{ll(r,m)} \wedge \texttt{m>0}$$

This can then be handled by two matching steps.

The folding process is more involved and can take place in two different scenarios: (1) the base case and (2) the recursive case. An example for the base case is as follows:

$$\texttt{y=null} \ \vdash\ \texttt{ll(y,m)} \wedge \texttt{m=0}$$

The folding process will try to prove a (different) recursive entailment:

$$\texttt{y=null} \vdash \texttt{y=null} \wedge \texttt{m=0} \vee \texttt{y}\mapsto\texttt{node}(\_,\texttt{r})*\texttt{ll}(\texttt{r},\texttt{m}-1)$$

where in this new entailment, the consequent is the definition of $\texttt{ll}(\texttt{y},\texttt{m})$. Once the new entailment is discharged, the result is returned and the original entailment becomes

$$\texttt{y=null} \wedge \underline{\texttt{m=0}} \vdash \texttt{m=0}$$

Note that the underlined part denotes the result returned from the folding process.

An example for the recursive folding process is as follows:

$$\texttt{x}\mapsto\texttt{node}(\_,\texttt{r}) \wedge \texttt{r=null} \vdash \texttt{ll}(\texttt{x},\texttt{n}) \wedge \texttt{n>0}$$

The folding process invokes the following recursive entailment:

$$\texttt{x}\mapsto\texttt{node}(\_,\texttt{r}) \wedge \texttt{r=null} \vdash \texttt{x=null} \wedge \texttt{n=0} \vee \exists \texttt{q},\texttt{k} \cdot \texttt{x}\mapsto\texttt{node}(\_,\texttt{q})*\texttt{ll}(\texttt{q},\texttt{k}) \wedge \texttt{k=n}-1$$

where the consequent of this new entailment is the definition of $\texttt{ll}(\texttt{x},\texttt{n})$. This recursive entailment is discharged by a matching process followed by a base case folding. When it returns, the original entailment becomes

$$\texttt{r=null} \wedge \underline{\texttt{n}-1\texttt{=0}} \vdash \texttt{n>0}$$

Same as the above, the underlined part denotes the result obtained from the folding process.

Formal details about the Sleek entailment proving process can be found at [8]. In the next section, we present our experience of applying Hip/Sleek to specify and verify the FreeRTOS scheduler.

## 4 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 important 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.

### 4.1 On user-defined predicates

As mentioned in the previous section, 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.

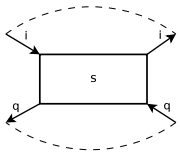
$$\texttt{XLIST(p)} \equiv \texttt{p}\mapsto\texttt{xList}(\_,\texttt{i},\texttt{i}) * \texttt{DLS}(\texttt{i},\texttt{q},\texttt{i},\texttt{q}) \vee \texttt{p}\mapsto\texttt{xList}(\_,\texttt{i},\texttt{e}) * \texttt{DLS}(\texttt{e},\texttt{e1},\texttt{i},\texttt{f1}) * \texttt{DLS}(\texttt{i},\texttt{f1},\texttt{e},\texttt{e1})$$

Capitalised words refer to *shape predicates* and `xList` is a *data node*. So, $\texttt{XLIST}(\texttt{p})$ means that `p` is a pointer to a structure of the shape `XLIST` and a data node of the shape $\texttt{xList}(\_,\texttt{i},\texttt{e})$ 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

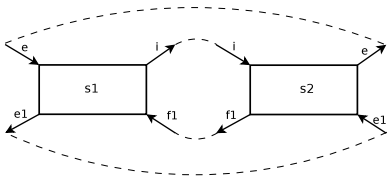
[6] It is important to note that in this work we do not verify if TCBs' stack and code pointers are valid. Invalid TCBs can affect context switching, but here we focus on ensuring that the scheduler makes *the right choices*.

[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.

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 identified by $i$); the second disjunct captures the case where they point to different segments. Recall that the star ($*$) operator represents *separating conjunction* and ensures that its arguments reside in disjoint heaps (for more information about separation logic, see [29]). In both cases, the arguments that are passed to the DLS predicate ensure that the items are indeed in *cyclic doubly-linked lists*. To help understand how the cyclic structure is formed, we refer the reader to Figure 15, where a graphical representation of these cyclic doubly-linked list segments is shown. Figure 15(a) refers to the first conjunct in the definition of the predicate XLIST, where only one segment is used (we name it s in the figure). Figure 15(b) refers to the second conjunct in the definition of the predicate XLIST, where two segments, named s1 and s2, are used: one referenced by *pxIndex* (variable i) and the other one referenced by *xListEnd* (variable e). The dashed lines indicate how the circular structures are achieved.




(a) First disjunct: $s$ is a doubly-linked circular segment




(b) Second disjunct: $s1$ and $s2$ are two connected doubly-linked circular segments

**Fig. 15** Graphical representation of doubly-linked list segments used in the definition of XLIST

The definition of the shape predicate DLS is:

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

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. Using the same graphic notation as in Figure 15, we would represent a circular doubly-linked list segment as shown in Figure 16.

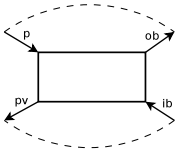



**Fig. 16** Graphical representation of $\texttt{DLS}(\texttt{p},\texttt{pv},\texttt{ob},\texttt{ib})$.

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.

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

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

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.

## 4.2 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, 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*. A graphical representation of the lemma is shown in Figure 17.

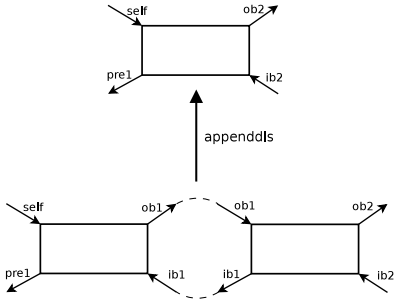



**Fig. 17** Graphical representation of the lemma **appenddls**

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 *vListRemove* function, 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`. A graphical representation of the lemma is shown in Figure 18.

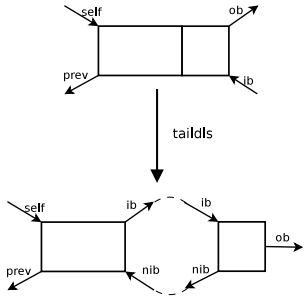



**Fig. 18** Graphical representation of the lemma **taildls**

## 4.3 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. In this section, we only show the functions relevant for the scheduler, 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 $\texttt{XLIST}(\texttt{pxList}, 0, \{\})$. 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 2).

Not all functions need to use shape predicates. For example, the function *vListInitialiseItem*, which sets the container of the given list item to *NULL*, can be specified as:

```
void vListInitialiseItem(xListItem pxItem)
 requires pxItem↦xListItem(v, p, q, _, o)
 ensures pxItem↦xListItem(v, p, q, null, o)
{ ··· }
```

Note how the postcondition guarantees that all the fields remain the same, apart from the container.

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 the field `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 shown in Figure 19, where the 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.

```
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 ∧
         ∃ B_i · B_i = B1∪B2 ∧ B_i = B∪{pxNewListItem}
{ ··· }
```

**Fig. 19** Alternative specification of the function *vListInsertEnd*

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 *pvContainer* 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.

In our experiments, we have tweaked this specification to minimize the search space and leverage on proof splitting. The specification that we used is semantically the same, but it is divided into six different cases (each case depends on whether the item to be removed is the one pointed by *pxIndex* and whether *pxIndex* and the end marker of the list are empty). We include more details about performance tuning in Section 6.

***Picking the next task*** The function shown in Figure 10 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:

```
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
{ ··· }
```

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.

***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.

***Functions that manipulate global variables*** Some important functions related with scheduling are controlled by manipulating global variables. Although these functions do not pose any challenge to the verification process, we include here two examples to show how they can be verified. The two examples shown are the functions *vTaskSuspendAll* and *vTaskEndScheduler*.

The function *vTaskSuspendAll* suspends all the tasks and the only command it performs is to increment by 1 a global variable named *uxSchedulerSuspended*. Hence, we can specify it as:

```
void vTaskSuspendAll()
  requires uxSchedulerSuspended≥0
  ensures uxSchedulerSuspended′ =
                          uxSchedulerSuspended + 1
{ ··· }
```

The precondition states that the value of *uxSchedulerSuspended* has to be a natural number; the postcondition guarantees that its value is incremented by 1. (The initial value of the variable is zero, so the precondition is satisfied initially.)

The function *vTaskEndScheduler* terminates the scheduler by setting the global variable *xSchedulerRunning* to false. We can specify it as:

```
void vTaskEndScheduler()
  requires true
  ensures !xSchedulerRunning′
{ ··· }
```

The precondition is true, meaning that the function can be called at any time without any restrictions. The postcondition states that the new value of *xSchedulerRunning'* is false.

### 4.4 Verification statistics

We finish by presenting some verification statistics associated with the functions discussed in this section. In Table 1 we summarise the annotation overhead of the functions discussed. The first column identifies the function, the second column refers to the size of the function (measures in lines of code), and the third column refers to the number of lines of annotations (we also include the annotation overhead percentage). To improve readability, we abbreviate the function name *list_GET_OWNER_OF_NEXT_ENTRY* to *list_GOONE*.

| **Function** | **Size** | **Annotations** |
|---|---|---|
| *vListInitialise* | 5 | 2 (+40%) |
| *vListInitialiseItem* | 1 | 2 (+200%) |
| *vListInsertEnd* | 9 | 13 (+144%) |
| *vListRemove* | 14 | 15 (+107%) |
| *list_GOONE* | 5 | 2 (+40%) |
| *vTaskSwitchContext* | 32 | 4 (+13%) |
| *prvAddTaskToReadyQueue* | 8 | 4 (+50%) |
| *vTaskSuspendAll* | 1 | 2 (+200%) |
| *vTaskEndScheduler* | 1 | 2 (+200%) |
| Totals | 76 | 46 (+61%) |

**Table 1** Annotation overhead of the functions discussed. The size of each function and the annotations column are measured in lines of code.

Table 1 shows that as the size of functions grow, the annotation overhead decreases. The overhead associated with one-liners will always be at least 200%, because we need to provide a pre- and a post-condition (and we write pre- and post-conditions in two separate lines). Note that if we remove the three one-liners, the overall overhead decreases from 61% to 55%.

The overhead associated with shape predicates and data structures is smaller than the overhead associated with functions. The data structures discussed in this paper use 35 lines of code and the shape predicates are written in 6 lines; this represents an overhead of 17%. However, if we include the 4 lines used for lemmas, the overhead is very similar to the one obtained for the functions: 29%.

Although in general the overhead decreases as the size of the functions grow, these results suggest that for large code bases, manually writing the annotations is not a scalable approach and can become difficult to manage. In Section 6, we briefly discuss how this could improve.

Table 2 presents the verification time for each of the functions discussed. Our test platform is a GNU/Linux server (Debian 3.2.46-1) with 8 cores Intel Core i7 CPU (8MB Cache, 2.93GHz) and with 16GB of RAM. The times are measured in seconds and were achieved using the main branch of the prototype Hip/Sleek compiled natively. We used MONA for proving properties involving bags and Omega for all other numerical properties (we used Hip/Sleek's option `-tp om`). The table displays the median value of five measurements. The times associated with functions *vListInsertEnd* and *vListRemove* are considerably higher than all the other times, due to the big search space that arises from the use of the shape predicates `XLIST` and `DLS`.

| Function | Verification time |
|---|---|
| *vListInitialise* | 0.27s |
| *vListInitialiseItem* | 0.10s |
| *vListInsertEnd* | 130.20s |
| *vListRemove* | 814.75s |
| *list_GOONE* | 0.60s |
| *vTaskSwitchContext* | 0.19s |
| *prvAddTaskToReadyQueue* | 0.34s |
| *vTaskSuspendAll* | 0.10s |
| *vTaskEndScheduler* | 0.11s |

**Table 2** Verification times of the functions discussed (in seconds).

Overall, given that Hip/Sleek is a research tool, we consider that the verification times are satisfactory. However, when disjunctions are included in the specification, the verification time is less than satisfactory (as can be observed with the function *vListRemove*). These results suggest that for large code bases, where disjunctive specifications are likely to be used, the verification times can easily become less than satisfactory. Techniques like the one described in [9], where disjunctive formulae are pruned when unfolding shape predicates, can be used to achieve better results. Another approach that can achieve better verification times is to divide the search space in disjoint parts and run the verification in parallel for each of them.

## 5 Related Work

Much work has been done on the verification of operating systems; see [17] 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 [33]. A number of tools have been developed to verify real system tools. A notable project on verification of RTOSs is Astrée [6], which proves no run-time 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 [3] and Blast [12], 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. [32]. Baumann et al. [4] 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. Finally, a complete verification of the seL4 microkernel is described in [18]. The verification uses the interactive prover Isabelle/HOL [24] and establishes the correspondence between an abstract specification and a low-level concrete C representation of the system (that was modelled in Isabelle/HOL and manually written by the authors). The work on seL4 is claimed to be the first-ever general-purpose OS kernel that has been verified and is indeed much more general than the work presented here. Another major difference is that our goal was never to write the C code for a verified OS kernel, but rather to verify an existing one.

Separation logic has been adopted by a number of tools to verify the memory safety of system code, such as SmallFoot [5], SpaceInvader [34] and THOR [21]. 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 [22], where the authors discuss different approaches that can be used to verify FreeRTOS. Particularly relevant is their use of Verifast [14], 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.

## 6 Discussion and Conclusion

This paper shows how the combination of shape and numerical information can be used to specify and verify key properties of 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.

### 6.1 Lessons learned

Verification of real-world code used in production is a very time-consuming process. The verification was initiated by the first author as an exercise in automated verification of real-world code. Although he has a background in formal methods, he was not familiar with FreeRTOS and he had no previous experience with Hip/Sleek. Therefore, he experienced a steep learning curve and got stuck a few times dealing with some of Hip/Sleek's idiosyncrasies; the other authors provided guidance whenever needed. The project took approximately six man-months. The verification progressed quite slowly for several reasons. First, understanding how FreeRTOS works in enough detail to be able to identify and verify relevant properties took approximately one man-month. Most of the time was spent modelling the data structures involved (i.e., identifying and refining shape predicates) and annotating the code. A second reason is that we needed to convert the original C code to Hip notation, and there are aspects we need to take into account other than the specifications. Some of the conversions were purely syntactic: for example, all the field accesses of the form $var{\rightarrow}field$ had to be rewritten as $var.field$ and C macros had to be rewritten as functions. However, some conversions were not so trivial: for example, Hip does not support type conversions and casts, so we had to carefully rewrite the code to avoid them (e.g., to avoid casts from *xMiniListItem* to *xListItem*, we decided to model the field *xListEnd* as a *xListItem*; this required a careful check to make sure that in this context, the fields *pvOwner* and *pvContainer* were not accessed). Third, because Hip/Sleek is under active development and has different branches with disjoint features, we were unable to use simultaneously some of these features (for example, for the work described in this paper, we initially used a stable version which did not include support for lemmas).

A considerable amount of time was also spent on performance tuning. In particular, i) tweaking the specifications to minimize the search space and leverage on proof splitting; ii) constraining lemma applications to precise points and forcing formula transformations through no-op method calls; and iii) improving the efficiency of the pure provers by adding a customized pointer constraint solving procedure.

First, during the specification process we observed that for seemingly small variations in specification structure the verification timings vary wildly. As Sleek was designed as a fully automatic entailment checker, it is built with a comprehensive search strategy. As discussed earlier, in some cases, by explicitly forcing an unfold or a fold operation within a method specifica-

tion we were able to reduce the timings considerably by reducing the inherent search space. As a further performance adjustment, we introduced annotations on the pure constraints that identify the distinct properties captured by the predicates, in this case, pointer (dis-)equalities, list sizes and set of elements in order to leverage on the tool's ability to split the proofs in distinct, independent sub-proofs which were inherently faster.

Secondly, we leveraged on the method call mechanism to force the verifier to transform the current state. To this end, we inserted at key points, method calls to functions with an empty body whose pre and post conditions were syntactically different but semantically equivalent. While operationally such a call is a no-op statement, in the verification process it induced a state transformation by forcing an entailment of specifications given for the stub method. The gain was two fold: first, such transformations lead to a simpler expression of the current state after the method call; second, by constructing stub functions that encapsulate the lemma effect we were able to remove the presence of lemmas during the main verification effort which led to further pruning of the search space. In order to maintain the soundness of the verification we applied the verification system to prove the correctness of the no-op functions, in the presence of the required lemmas. We note that such transformations are not equivalent to the mechanisms they mimic as the general mechanisms are built in the entailment checker and can be automatically applied in the proving process while the no-op functions have a much more restricted scope: they are applied only when explicitly called by the programmer in the method body. Nevertheless, they can help with tuning the verification performance.

Finally, in the process of verifying these specifications, we observed that in many cases the generated proof obligations would accumulate a large number of pointer equality and disequality constraints which would be passed down to the pure provers leading to large proving times. To alleviate the problem, we introduced in the Sleek checker a specialized decision procedure for these two types of constraints. Thus, by calling the new checker whenever possible, instead of more powerful provers like Omega or Mona, we achieve a large drop in the verification times, up to seven fold for particular examples.

### 6.2 Future Work

We plan to verify other components of FreeRTOS, but it is still unclear how certain fairness and timing properties can be verified. For example, at the moment, we cannot use Hip/Sleek to prove that a task scheduled to run at moment $t$ will run at moment $t + \Delta t$ (we plan to use the work described in [10] as a starting point to extend our prototype). Also, a challenging problem is to verify that the queue accesses are indeed thread-safe (this implies reasoning about interrupts, which is known to be a difficult problem).

Another 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 [25,26,28,27] in Hip/Sleek will allow the automatic inference of properties, making our approach more scalable.

We plan to use recent work done on Hip/Sleek [31] to verify code based on overlaid structures [19,20], 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.

## References

1. The FreeRTOS™ project website. URL: `http://www.freertos.org`
2. The SafeRTOS™ project website. URL: `http://www.freertos.org/safertos.html`
3. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. In: EuroSys (2006)
4. Baumann, C., Beckert, B., Blasum, H., Bormer, T.: Formal verification of a microkernel used in dependable software systems. In: SAFECOMP (2009)
5. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: FMCO (2005)
6. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: PLDI (2003)
7. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. Journal of the ACM **58**(6), 26 (2011)
8. Chin, W.N., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Science of Computer Programming **77**, 1006–1036 (2012)
9. Chin, W.N., Gherghina, C., Voicu, R., Le, Q.L., Craciun, F., Qin, S.: A specialization calculus for pruning disjunctive predicates to support verification. In: CAV (2011)

10. Cook, B., Koskinen, E., Vardi, M.Y.: Temporal property verification as a program analysis task. In: CAV (2011)
11. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: TACAS, pp. 287–302 (2006)
12. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL (2004)
13. Hoare, T.: The verifying compiler: A grand challenge for computing research. J. ACM **50** (2003)
14. Jacobs, B., Smans, J., Piessens, F.: A quick tour of the verifast program verifier. In: APLAS (2010)
15. Jones, C., O'Hearn, P., Woodcock, J.: Verified software: A grand challenge. Computer **39** (2006)
16. Klarlund, N., Møller, A.: MONA Version 1.4 User Manual (2001)
17. Klein, G.: Operating system verification — An overview. Sadhana **34**, 27–69 (2009)
18. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: sel4: formal verification of an os kernel. In: SOSP (2009)
19. Lee, O., Yang, H., Petersen, R.: Program analysis for overlaid data structures. In: CAV (2011)
20. Lee, O., Yang, H., Petersen, R.: A divide-and-conquer approach for analysing overlaid data structures. Formal Methods in System Design **41**(1), 4–24 (2012)
21. Magill, S., Tsai, M.H., Lee, P., Tsay, Y.K.: Thor: A tool for reasoning about shape and arithmetic. In: CAV (2008)
22. Mühlberg, J.T., Leo, F.: Verifying FreeRTOS: from requirements to binary code. In: AVoCS (2011)
23. Nguyen, H.H., David, C., Qin, S., Chin, W.N.: Automated verification of shape and size properties via separation logic. In: VMCAI (2007)
24. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, *LNCS*, vol. 2283. Springer (2002)
25. Qin, S., He, G., Luo, C., Chin, W.N.: Loop invariant synthesis in a combined domain. In: ICFEM (2010)
26. Qin, S., He, G., Luo, C., Chin, W.N., Chen, X.: Loop invariant synthesis in a combined abstract domain. Journal of Symbolic Computation **50**(0), 386 – 408 (2013)
27. Qin, S., He, G., Luo, C., Chin, W.N., Yang, H.: Automatically refining partial specifications for heap-manipulating programs. To appear in Science of Computer Programming (2013)
28. Qin, S., Luo, C., Chin, W.N., He, G.: Automatically refining partial specifications for program verification. In: FM (2011)
29. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS (2002)
30. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: POPL (1999)
31. Sharma, A., Hobor, A., Chin, W.N.: Specifying compatible sharing in data structures (2013). In preparation.
32. Sputh, B.H.C., Faust, O., Verhulst, E., Mezhuyev, V.: Opencomrtos: A runtime environment for interacting entities. In: CPA (2009)
33. Woodcock, J.: Grand challenge in software verification. In: SBMF (2008)
34. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable shape analysis for systems code. In: CAV (2008)