Detecting All High-Level Dataraces in an RTOS Kernel

Suvam Mukherjee(✉), Arun Kumar, and Deepak D'Souza

Indian Institute of Science, Bangalore, India
{suvam,deepakd}@csa.iisc.ernet.in, aruns.siva@gmail.com

Abstract. A high-level race occurs when an execution interleaves instructions corresponding to user-annotated critical accesses to shared memory structures. Such races are good indicators of atomicity violations. We propose a technique for detecting *all* high-level dataraces in a system library like the kernel API of a real-time operating system (RTOS) that relies on flag-based scheduling and synchronization. Our methodology is based on model-checking, but relies on a meta-argument to bound the number of task processes needed to orchestrate a race. We describe our approach in the context of FreeRTOS, a popular RTOS in the embedded domain.

1 Introduction

Atomicity violations [13] precisely characterize the bugs in a method library that arise due to *concurrent* use of the methods in the library. An execution of an application program that uses the library is said to exhibit an atomicity violation if its behaviour cannot be matched by any "serialized" version of the execution, where none of the method calls interleave with each other. As one may expect, such bugs can be pernicious and difficult to detect.

A necessary condition for an atomicity violation to occur in a library L is that two method invocations should be able to "race" (or interleave) in an execution of an application that uses L. In fact it is often necessary for two "critical" access paths in the source code of the methods (more precisely the instructions corresponding to them) to interleave in an execution, to produce an atomicity violation. With this in mind, we could imagine that a user (or the developer herself) annotates blocks of code in each method as critical accesses to a particular unit of memory structures. We can now say that an execution exhibits a "high-level" race (with respect to this annotation) if it interleaves two critical accesses to the same memory structure.

Suppose we now had a way of finding the precise set R of pairs of critical accesses that could race with each other, across *all* executions in *all* applications programs that use L. We call this the problem of finding all high-level races in L. The user can now focus on the set R, which is hopefully a small fraction of the set of all possible pairs, and investigate each of them to see whether they could lead to atomicity violations. We note that the user can *soundly* disregard the

© Springer International Publishing AG 2017
A. Bouajjani and D. Monniaux (Eds.): VMCAI 2017, LNCS 10145, pp. 405–423, 2017.
DOI: 10.1007/978-3-319-52234-0_22

pairs outside R as they can never race to begin with, and hence can never be the cause of any atomicity violation.

In this paper we are interested in the problem of finding all high-level races in a library like the Application Programmer Interface (API) of a real-time kernel. The particular system we are interested in is a real-time operating system (RTOS) called FreeRTOS [23]. FreeRTOS is one of the most popular operating systems in the embedded industry, and is widely used in real-time embedded applications that run on microcontrollers with small memory. FreeRTOS is essentially a library of API functions written in C and Assembly, that an application programmer invokes to create and manage tasks. Despite running on a *single* processor or core, the execution of tasks (and hence the kernel API functions) can interleave due to interrupts and context-switches, leading to potential races on the kernel data-structures.

The kind of control-flow and synchronization mechanisms that kernels like FreeRTOS use are non-standard from a traditional programming point of view. To begin with, the control-flow *between* threads is very non-standard. In a typical concurrent program, control could potentially switch between threads at *any* time. However in FreeRTOS, control switching is restricted and depends on whether interrupts have been disabled, the value of certain flag variables like `SchedulerSuspended`, and whether the task is running as part of an interrupt service routine (ISR). Secondly, FreeRTOS does not use standard synchronization mechanisms like locks, but relies instead on mechanisms like disabling interrupts and flag-based synchronization. This makes it difficult to use or adapt some of the existing approaches to high-level race detection like [3,28] or classical datarace detection like [9,29], which are based on standard control-flow and lock-based synchronization.

An approach based on model-checking could potentially address some of the hurdles above: one could model the control-flow and synchronization mechanism in each API function faithfully, create a "generic" task process that non-deterministically calls each API function, create a model (say M_n) that runs n of these processes in parallel, and finally model-check it for data-races. But this approach has some basic roadblocks: certain races need a minimum number of processes running to orchestrate it—how does one determine a sufficient number of processes n that is guaranteed to generate *all* races? Secondly, even with a small number of processes, the size of the state-space to be explored by the model-checker could be prohibitively large.

The approach we propose and carry out in this paper is based on the model-checking approach above, but finds a way around the hurdles mentioned. The key idea is to create a set of *reduced* models, say $\mathcal{M}_{red}$, in which each model essentially runs only three API functions at a time. We then argue that a race that shows up in M_n, for *any* n, must also be a race in one of the reduced models in $\mathcal{M}_{red}$. Model-checking each of these reduced models is easy, and gives us a way of finding *all* data-races that may ever arise due to use of the FreeRTOS API. We note that the number of API functions to run in each reduced model (three in this case), and the argument of sufficiency, is specific to FreeRTOS. In general, this will depend on the library under consideration.

On applying this technique to FreeRTOS (with our own annotation of critical accesses) we found a total of 48 pairs of critical accesses that could race. Of these 10 were found to be false positives (i.e. they could not happen in an actual execution of a FreeRTOS application). Of the remaining, 16 were classified as harmful, in that they could be seen to lead to atomicity violations. The bottom-line is that the user was able to disregard 99.8% of an estimated 41,000 potential high-level races.

In the next couple of sections we describe how FreeRTOS works and our notion of high-level races in its context. In Sect. 4 we describe how we model the API functions and control-flow in Spin, and give our reduction argument in Sect. 5. We describe our experimental results in Sect. 6 and related work in Sect. 7.

2 Overview of FreeRTOS

FreeRTOS [23] is a real-time kernel meant for use in embedded applications that run on microcontrollers with small to mid-sized memory.

It allows an application to organise itself into multiple independent tasks (or threads) that will be executed according to a priority-based preemptive scheduling policy. It is implemented as a library of functions (or an API) written mostly in C, that an application programmer can include with their code and invoke as functions. The API provides the programmer ways to create and schedule tasks, communicate between tasks (via message queues, semaphores, etc.), and carry out time-constrained blocking of tasks.

Figure 1 shows a simple FreeRTOS application. In `main` the application first creates a queue with the capacity to hold a single message of type `int`. It then creates two tasks called "`Prod`" and "`Cons`" of priority 2 and 1 respectively using the `TaskCreate` API function which adds these two tasks to the "Ready" list. The FreeRTOS scheduler is then started by the call to `StartScheduler`. The scheduler schedules the `Prod` task first, being the highest priority ready task. `Prod` sends a message to the queue, and then asks to be delayed for two time units. This results in `Prod` being put into the "Delayed" list. The next available task,

```
int main(void) {
  QueueHandle q;
  q = QueueCreate(1, sizeof(int));
  TaskCreate(prod, "Prod", 2, ...);
  TaskCreate(cons, "Cons", 1, ...);
  StartScheduler();
}

void prod(void* params) {
  for(;;) {
    QueueSend(q,...);
    TaskDelay(2);
  }
}

void cons(void* params) {
  for(;;) {
    QueueReceive(q,...);
  }
}
```

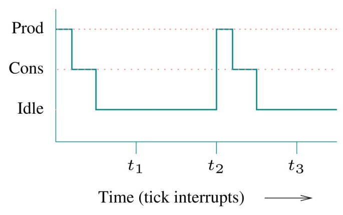

Fig. 1. An example FreeRTOS application and its execution

`Cons`, is run next. It dequeues the message from the queue, but is blocked when it tries to dequeue again. The scheduler now makes the `Idle` task run. A timer interrupt now occurs, causing an ISR called `IncrementTick` to be run. This routine increments the current tick count, and checks the delayed list to see if any tasks need to be woken up. There are none, so the `Idle` task resumes execution. However when the second tick interrupt occurs, the ISR finds that the `Prod` task needs to be woken up, and moves it to the ready list. As `Prod` is now the highest priority ready task, it executes next. This cycle repeats, ad infinitum.

The FreeRTOS kernel maintains a bunch of data-structures, variables and flags, some of which are depicted in Fig. 2. Tasks that are ready to run are kept in the `ReadyTasksList`, an array which maintains—for each priority—a pointer to a linked list of tasks of that priority that are ready to run. When a running task delays itself, it is moved from the `ReadyTasksList` to the `DelayedTaskList`, with an appropriate time-to-awake value. User-defined queues, like `q` in the example application, are maintained by the kernel as a chunk of memory to store the data (shown as `QueueData` in the figure), along with an integer variable `MessagesWaiting` that records the number of messages in the queue, and two associated lists `WaitingToSend` and `WaitingToReceive` that respectively contain the tasks that are blocked on sending to and receiving from the queue.

Even though FreeRTOS applications typically run on a *single* processor (or a *single core* of a multi-core processor), the kernel API functions can interact with each other in an interleaved manner. While a function invoked by the current task is running, there could be an interrupt due to which an ISR runs, which in turn may either invoke another API function, or unblock a higher priority task which goes on to execute another API function. The FreeRTOS API functions thus need to use some kind of synchronization mechanism to ensure "exclusive" access to the kernel data-structures. They do so in a variety of ways, to balance the trade-off between securing fully exclusive access and not losing interrupts. The strongest exclusion is achieved in a "critical section," where an API function disables interrupts to the processor, completes its critical accesses, and then re-enables interrupts. During such a critical section no preemption (and hence no interleaving) is possible. The second kind of exclusion is achieved by "suspending" the scheduler. This is done by setting the kernel flag `SchedulerSuspended` to 1. While the scheduler is suspended (i.e. this flag is set), no other task will be scheduled to run; however, unlike in a critical section, *interrupts* can still occur and an ISR can execute some designated API functions (called "fromISR" functions which are distinguished from the other "task" functions). The implicit protocol is that these functions will check whether the `SchedulerSuspended` flag is set, and if so they will not access certain data-structures like the `ReadyTasksList`, but move tasks when required to the `PendingReadyList` instead. Figure 2 shows some of the structures protected by the `SchedulerSuspended` flag.

The final synchronization mechanism used in FreeRTOS is a pair of per-user-queue "locks" (actually *flags* which also serve as *counters*) called `RxLock` and

`TxLock`, that protect the `WaitingToReceive` and `WaitingToSend` lists associated with the queue. When a task executes an API function that accesses a user-queue, the function sets these locks (increments them from their initial value of -1 to 0). Any fromISR function that now runs will now avoid accessing the waiting lists associated with this queue, and instead increment the corresponding lock associated with the queue to record the fact that data has been added or removed from the queue. When the interrupted function resumes, it will move a task from the waiting list back to the ready list, for each increment of a lock done by an ISR. These locks and the lists they protect are also depicted in Fig. 2.

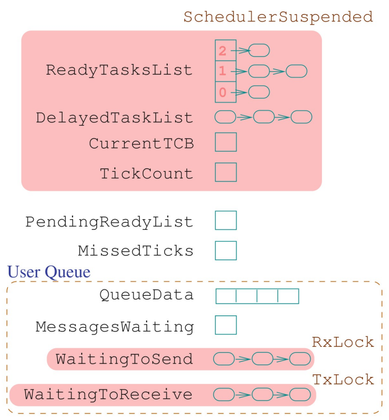

Fig. 2. Kernel data-structures and protecting flags in FreeRTOS

Figure 3 shows parts of the implementation of two FreeRTOS APIs. The `QueueSend` function is used by a task to enqueue an item in a user-defined queue `pxQ`. Lines 3–9 are done with interrupts disabled, and corresponds to the case when there is place in the queue: the item is enqueued, a task at the head of `WaitingToReceive` is moved to the `ReadyTasksList`, and the function returns successfully. In lines 14–26, which corresponds to the case when the queue is full, the function enables interrupts, checks again that the queue is still full (since after enabling interrupts, an ISR could have removed something from the queue), and goes on to move itself from the Ready queue to the `WaitingToSend` list of `pxQ`. This whole part is done by first suspending the scheduler and locking `pxQ`, and finally unlocking the queue and resuming the scheduler. The call to `LockQueue` in line 16

```
int QueueSend(QHandle pxQ, void *ItemToQueue) {
 1  // Repeat till successful send
 2  DISABLE_INTERRUPTS();
 3  if(!QueueFull(pxQ)) { // Queue is not full
 4    // Copy data to queue
 5    CopyDataToQueue(pxQ, ItemToQueue);
 6    if(!empty(pxQ->WaitingToReceive)) {
 7      ... // Move task from WaitingToReceive
 8      ... // to ReadyTasksList
 9    }
10    ENABLE_INTERRUPTS();
11    return PASS;
12  }
13  // Reach here when queue is full
14  ENABLE_INTERRUPTS();
15  ++SchedulerSuspended; // Suspend scheduler
16  LockQueue(pxQ);//Inc Tx(Rx)Lock with ints disabled
17  if(QueueFull(pxQ)) { // Check if queue still full
18    ... // Move current task from ReadyTasksList
19    ... // to WaitingToSend
20  }
21  UnlockQueue(pxQ);//Move tasks from waiting lists
22    // and unlock, with ints disabled
23  --SchedulerSuspended; // Resume scheduler
24  if (...) { // higher priority task woken
25    YIELD();
26  }
}

void IncrementTick() {
 1  if(SchedulerSuspended == 0) {
 2    ++TickCount;
 3    if(TickCount == 0) {
 4      ... // swap delayed lists
 5      DelayedTaskList = OverflowDelayedTaskList;
 6    }
 7    ... // Move tasks whose time-to-awake is now,
 8    ... // from DelayedTaskList to ReadyTasksList.
 9  }
10  else {
11   ++MissedTicks;
12  }
}
```

Fig. 3. Excerpts from FreeRTOS functions

increments both `RxLock` and `TxLock`. The call to `UnlockQueue` in line 21 decrements `RxLock` as many times as its value exceeds 0, each time moving a task (if present) from `WaitingToSend` to Ready. It does a similar sequence of steps with `TxLock`. Both these functions first disable interrupts and re-enable them once their job is done. Finally, in lines 24–26, the function checks to see if it has unblocked a higher priority task, and if so "yields" control to the scheduler.

The second API function in Fig. 3 is the `IncrementTick` function that is called by the timer interrupt, and which we consider to be in the fromISR category of API functions. If the scheduler is *not* suspended, it increments the `TickCount` counter, and moves tasks in the `DelayedTaskList` whose time-to-awake equals the current tick count, to the Ready list. If the scheduler *is* suspended, it simply increments the `MissedTicks` counter.

3 High-Level Races in FreeRTOS

In this section we describe our notion of a high-level race in a system library like FreeRTOS. Essentially a race occurs when two "critical" access paths in two API functions interleave. We make this notion more precise below.

Consider a system library L. Our notion of a race in L is parameterized by a set $\mathcal{S}$ of shared memory structures maintained by the library, and a set $\mathcal{C}$ of "critical accesses" of structures in $\mathcal{S}$. The set of structures in $\mathcal{S}$ is largely determined by the developer's design for thread-safe access. We can imagine that the developer has in mind a partition of the shared memory structures into "units" which can be independently accessed: thus, it is safe for two threads to simultaneously access two *distinct* units, while it is potentially unsafe for two threads to access the *same* unit simultaneously. For instance, in FreeRTOS, the set $\mathcal{S}$ could contain shared variables like `SchedulerSuspended`, or shared data-structures like `ReadyTasksList`, or an entire user-queue. The set of critical accesses $\mathcal{C}$ would comprise contiguous blocks of code in the API functions of L, each of which corresponds to an access of one of these units in $\mathcal{S}$. The accesses are "critical" in that they are *not* meant to interleave with other accesses to the same unit of structures. Each critical access comes with a classification of being a *write* or *read* access to a particular shared structure v in $\mathcal{S}$. For example, we could have the block of code in lines 17–20 of the `QueueSend` function in Fig. 3, as a critical write to the user-queue structure, in $\mathcal{C}$. Finally, we say that a pair of accesses in $\mathcal{C}$ are *conflicting* if they both access the same structure v in $\mathcal{S}$ and at least one is a write access.

An execution of an application program A that uses L—an L*-execution* for short—is an interleaving of the execution of the tasks (or threads) it creates. An execution of a task in turn is a (feasible) sequence of instructions that follows the control-flow graph of its compiled version. Since these tasks may periodically invoke the functions in L, portions of their execution will correspond to the critical paths in these functions. We say that an L-execution exhibits an $(\mathcal{S}, \mathcal{C})$ *high-level race* (or just $(\mathcal{S}, \mathcal{C})$*-race* for short) on a structure v in $\mathcal{S}$, if it *interleaves* the execution paths corresponding to two conflicting critical accesses to v (i.e. the second critical access begins before the first ends).

When do we say an $(\mathcal{S}, \mathcal{C})$-race is "harmful"? We can use the notion of atomicity violation from [13] (see also [10]) to capture this notion. Consider an L-execution ρ. Each task in the application may invoke functions in L along ρ, and some of these invocations may overlap (or interleave) with invocations of functions of L in other tasks. A *linearized* version of ρ follows the same sequence of invocations of the functions in L along ρ, except that *overlapping* invocations are re-ordered so that they no longer overlap. We refer the reader to [17] for a more formal definition of linearizability. We can now say that an L-execution ρ exhibits an *atomicity violation* if there is *no* linearized version of the execution that leaves the shared memory structures in the same state as ρ. This definition differs slightly from [13] in that we prefer to use the notion of linearizability rather than serializability.

For a given $(\mathcal{S}, \mathcal{C})$, we say that an $(\mathcal{S}, \mathcal{C})$-race is *harmful* if there is an L-execution that contains this race, exhibits an atomicity violation, and this race plays a role (possibly along with other threads) in producing this atomicity violation. Otherwise we say the race is *benign*. Finally, we say that a given $(\mathcal{S}, \mathcal{C})$ pair is *safe* for L, if every L-execution that exhibits an atomicity violation also exhibits an $(\mathcal{S}, \mathcal{C})$-race. We note that we can always obtain a safe $(\mathcal{S}, \mathcal{C})$ by putting all memory structures into a single unit in $\mathcal{S}$ and entire method bodies into $\mathcal{C}$. However this would lead to lots of false positives, and it is thus preferable to have as finely-granular an $(\mathcal{S}, \mathcal{C})$ as possible.

We now proceed to describe our choice of what we believe to be safe choice of $\mathcal{S}$ and $\mathcal{C}$ for FreeRTOS. Some natural candidates for units in $\mathcal{S}$ are the various task lists like `ReadyTasksList` and `DelayedTaskList`. For a user-defined queue, one could treat the entire queue—comprising `QueueData`, `MessagesWaiting`, and the `WaitingToSend` and `WaitingToReceive` lists—as a single unit. However, this view would go against the fact that, by design, a task could be accessing the `WaitingToSend` component, while an ISR accesses the `QueueData` component. Hence, we keep each component of a user-defined queue as a separate unit in $\mathcal{S}$. Finally, we include all shared flags like `SchedulerSuspended`, pointer variables like `CurrentTCB`, and counters and locks like `TickCount` and `xRxLock`, in $\mathcal{S}$. Corresponding to this choice of units in $\mathcal{S}$, we classify, for example, the following blocks of code as critical accesses in $\mathcal{C}$: line 3 of the `QueueSend` function as a read access of `MessagesWaiting`, line 5 as a write to `QueueData`, line 6 as a read of `WaitingToSend`, and lines 7–8 as a write to both `WaitingToReceive` and `ReadyTasksList`.

We now give a couple of examples of races with respect to the set of structures $\mathcal{S}$ and accesses $\mathcal{C}$ described above. Assume for the sake of illustration, that the `QueueSend` function did *not* disable interrupts in line 2. Consider an execution of the example application in Fig. 1, in which the `Prod` task calls the `QueueSend` function, and begins the critical write to `ReadyTasksList`. At this point a timer interrupt comes and causes the `IncrementTick` ISR to run and execute the critical write to `ReadyTasksList` in lines 7–8. This execution would constitute a race on `ReadyTasksList`.

As a second example, consider the write access to SchedulerSuspended in the equivalent of line 15 of the QueueReceive function, and the read access of the same variable in line 1 of IncrementTick. Then an execution of the example application of Fig. 1 in which Cons calls the QueueReceive function when the queue is empty and executes the equivalent of line 15 to suspend the scheduler, during which it is interrupted by the IncrementTick ISR which goes on to execute line 1. This execution constitutes a race between the QueueReceive and IncrementTick API functions on the SchedulerSuspended variable.

The race on ReadyTasksList above is an example of a harmful race since it could lead to the linked list being in an inconsistent state that cannot be produced by any linearization of the execution. The race on SchedulerSuspended turns out to be benign, essentially due to the variable being declared to be *volatile* (so reads/writes to it are done directly from memory), and fact that an ISR runs to *completion* before we can switch back to QueueReceive.

4 Modelling FreeRTOS in Spin

In this section we describe how we model the FreeRTOS API and check for $(\mathcal{S}, \mathcal{C})$-races using the model-checking tool Spin. Spin's modelling language Promela can be used to model finite-state concurrent systems with standard communication and synchronization mechanisms (like message channels, semaphores, and locks). One can then model-check the system model to see if it satisfies a given state assertion or LTL property. For more details on Spin we refer the reader to [18].

Our first aim is to generate a Promela model M_n which captures the possible interleavings of critical accesses in any FreeRTOS application with at most n tasks. To make this more precise, consider a FreeRTOS application that—along any execution—creates at most n tasks. We denote such an application by A_n. We now define a Promela model M_n that has the following property (**P**):

> For every execution of A_n, which exercises the critical accesses within the FreeRTOS API functions in a certain interleaved manner, there is a corresponding execution in M_n with a similar manner of interleaving.

For a given n, the Promela model M_n is built as follows. We introduce four semaphores called task, sch, isr, and schsus to model the possible control switches between processes. Recall that a (binary) semaphore has two possible states 0 and 1, and blocking operations up and down which respectively change the state from 0 to 1 and 1 to 0. Initially all the semaphores are down (i.e. 0) except sch which is 1 to begin with. The semaphores are used to indicate when a particular API function is enabled. For example when the sch semaphore is up, the scheduler process—which first tries to down the sch semaphore—is enabled. Similarly, the task semaphore controls when a task function is enabled, and the isr semphore controls when a fromISR function is enabled. The schsus semaphore is used to ensure that whenever a task function is interrupted while the scheduler is *suspended*, control returns to the interrupted task function only.

```
inline QueueSend() {
  // do in a loop
  interrupt();
  // atomically, so no interrupts
  _MessagesWaiting++;
  _MessagesWaiting--;
  if
  :: skip ->
       // Copy data to queue
       _queueData += 2; _queueData -= 2;
       // Check if WaitingToReceive is non-empty
       _WaitingToReceive++; _WaitingToReceive--;
       // Move task from WaitingToReceive to Ready
       _WaitingToReceive += 2; _WaitingToReceive -= 2;
       _ReadyTasksList += 2; _ReadyTasksList -= 2;
  :: skip;
  fi
  // end of atomic, so interrupts enabled
  interrupt();
  if
  :: ++SchedulerSuspended; interrupt();
     LockQueue(); interrupt();
     // Move current task from Ready to WaitingToSend
     _ReadyTasksList += 2;interrupt();_ReadyTasksList -= 2;
     _WaitingToSend += 2;interrupt();_WaitingToSend -= 2;
     UnlockQueue(); interrupt();
     --SchedulerSuspended;  // Resume scheduler
     if
     :: up(sch); down(task); // Yield
     :: skip;
     fi
  :: skip;
  fi
}
```

```
inline interrupt() {
  if
  :: SchedulerSuspended==0 ->
       up(isr); down(task)
  :: SchedulerSuspended==1 ->
       up(isr); down(schsus)
  :: skip;
  fi}

inline LockQueue() {
//Inc Tx(Rx)Lock with ints disabled
   _TxLock +=2; TxLock++;
   _TxLock -= 2;
   _RxLock +=2; RxLock++;
   _RxLock -= 2;}

inline UnlockQueue() {
   // atomically
   do
   :: TxLock > 0 -> ... --TxLock;
      //Move tasks from
      //WaitingToReceive to Ready
   :: TxLock = 0 -> break;
   od
   TxLock = -1; // unlock queue
   //end atomic
   interrupt();
   ... // Similarly for RxLock}
```

Fig. 4. Promela model of the `QueueSend` API function

Each API function is modelled as a Promela function with the same name. We model variables of FreeRTOS that are critical to maintaining mutual exclusion, like `SchedulerSuspended`, `RxLock` and `TxLock`. We capture conditionals involving these variables and updates to these variables faithfully, and abstract the remaining conditionals conservatively to allow control-flow (non-deterministically) through both true and false branches of the conditional.

For each structure $\mathtt{v} \in \mathcal{S}$, we introduce a numeric variable called "_v", which is initialized to 0. For each critical write access to a structure `v` in an API function F, we add a statement `_v += 2` (short for `_v = _v + 2`) at the beginning of the block, and the statement `_v -= 2` at the end of the block, in the Promela version of F. Similarly, for a read access of `v` we add the statements `_v++` and `_v--` at appropriate points in the function. The possible context-switches due to an interrupt or yield to the scheduler are captured by uping the `isr` or `sch` semaphore. In particular, at any point in a function where an interrupt can occur (i.e. whenever interrupts are *not* disabled or an ISR itself is running), we add a call to `interrupt()` which essentially up's the `isr` semaphore and waits till a down is enabled on the `task` semaphore. The Promela function corresponding to the `QueueSend` function is shown in Fig. 4.

Each task in A_n is abstracted and conservatively modelled by a single process called `taskproc` in M_n, which repeatedly chooses a task API function

```
proctype scheduler() {
  do
  :: down(sch);
     if
     :: SchedulerSuspended==0 -> up(task)
     :: SchedulerSuspended==1 -> up(schsus)
     :: up(isr)
     fi
  od
}

proctype taskproc() {
  do
  :: down(task); QueueSend(); up(sch);
  :: ...
  :: down(task); TaskDelay(); up(sch);
  od
}

proctype isrproc() {
  do
  :: down(isr); IncrementTick(); up(sch);
  :: ...
  :: down(isr); QueueSendFromISR(); up(sch);
  od
}

init {
  run scheduler();
  // start n task and 1 ISR process
  run taskproc(); ...; run taskproc();
  run isrproc();
}
```

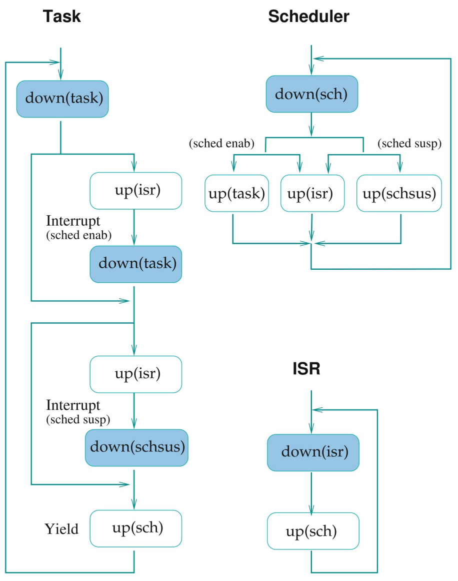

Fig. 5. (a) Promela model M_n and (b) Control flow and switching in M_1

non-deterministically and calls it. In a similar way, we model the fromISR API functions, and the `isrproc` process repeatedly invokes one of these non-deterministically. The Promela code of model M_n is depicted in Fig. 5(a). Thus M_n runs one `scheduler` process, one `isrproc` process, and n `taskproc` processes. Figure 5(b) illustrates the way the semaphores are used to model the control-switches.

Let us define what we consider to be a race in M_n. Let the statements in M_n be $s_1, \ldots, s_m$. If statement s_i is part of the definition of API F we write $\Gamma(s_i) = F$. An execution of M_n is a sequence of these statements that follows the control-flow of the model, and is *feasible* in that each statement is *enabled* in the state in which it is executed. We say an execution ρ of M_n exhibits a datarace on a structure v, involving statements s_i and s_j if (a) s_i and s_j are both increments of _v, (b) at least one increments _v by 2, and (c) ρ is of the form $\pi_1 \cdot s_i \cdot \pi_2 \cdot s_j$ with the segment π_2 not containing the decrement of _v corresponding to s_i. Note that the value of _v along ρ will exceed 2 after s_j.

It is not difficult to see that M_n satisfies the property (**P**) above. Consequently, any race on a structure $\mathtt{v} \in \mathcal{S}$ in application A_n will have a corresponding execution in M_n which exhibits a datarace on _v. Thus, it follows that by model-checking M_n for the invariant

```
((_ReadyTasksList < 3) && (_DelayedTaskList < 3) && ...)
```

we will find all races that may arise in an n-task application A_n. We note that there may be some false positives, due to conservative modelling of conditionals in the API functions, or because of 3 consecutive read accesses.

There are now two hurdles in our path. The first is that we need to model-check M_n for *each* n, as it is possible that some races manifest only for certain values of n. Secondly, model-checking even a single M_n may be prohibitively time-consuming due to the large state-space of these models. In fact, as we report in Sect. 6, Spin times out even on M_2, after running for several hours. We propose a way out of this problem, by first proving a meta-claim that any race between API functions F and G in M_n, will also manifest in a *reduced* model, $M_{F,G,I}$, in which we have a process that runs *only* F, one that runs *only* G, another that runs a fromISR function I, along with the `scheduler` process, and an ISR process that runs only the `IncrementTick` function. We denote this set of reduced models by $\mathcal{M}_{red}$. We then go on to model-check each of these reduced models for dataraces. Though there are now thousands of models to check, each one model-checks in a few seconds, leading to tractable overall running time.

In the next section we justify our meta-claim.

5 Reduction to $\mathcal{M}_{red}$

Before we proceed with our reduction claim, we note that this claim may not hold for a general library. Consider for example the library L with three API functions F, G, and H shown in Fig. 6. Suppose the variable `x` belongs to the set of structures $\mathcal{S}$ and the lines 2 and 6 constitute a critical read and write access, respectively, to `x`. Then the $(\mathcal{S}, \mathcal{C})$-race on `x` involving lines these accesses will never show up in any reduced model in $\mathcal{M}_{red}$, since we need all three functions to execute in order to produce this race. Thus, as we do for FreeRTOS below, any choice regarding the structure of models in $\mathcal{M}_{red}$ and the argument for its sufficiency, must be tailored for a given library and the way it has been modelled.

```
F() {           G() {            H() {
1  ...          4  ...           7  ...
2  read(x);     5  if (flag)     8  flag := true;
3  ...          6    write(x);   9  ...
}               }                }
```

Fig. 6. Example library where $\mathcal{M}_{red}$ does not suffice.

We now describe our reduction claim for our FreeRTOS model:

Theorem 1. *Let $n \geq 1$, and let ρ be an execution of M_n exhibiting a race involving statements s_i and s_j of M_n. Then there exists a model $M \in \mathcal{M}_{red}$, and an execution ρ_{red} of M, which also exhibits a race on s_i and s_j.*

We justify this claim in the rest of this section. By the construction of M_n, the execution ρ must be of the form

$$\pi_1 \cdot \texttt{down(task)} \cdot \pi_2 \cdot s_i \cdot \texttt{up(isr)} \cdot \pi_3 \cdot s_k$$

where s_i is a statement in API function F, and $\texttt{down(task)} \cdot \pi_2 \cdot s_i \cdot \texttt{up(isr)}$ is the portion of ρ corresponding to the racy invocation of F. We note that s_i must be part of a task function, while s_k could be part of either a task or fromISR function.

We consider two cases corresponding to whether the `SchedulerSuspended` flag is 1 or 0 after the statement s_i in ρ. Let us consider the first case where `SchedulerSuspended` is 1 after s_i. In this case, the statement s_k must belong to a fromISR function, say I. This is because the scheduler remains suspended after s_i in ρ (only F can resume it, and F never executes after $s_i \cdot$ `up(isr)` in ρ), and hence no task API function can run in this suffix of ρ. Further, since interrupts run to completion, the path π_3 must be of the form $\pi_3' \cdot \pi_4 \cdot s_k$, where π_3' comprises a sequence of fromISR functions, and $\pi_4 \cdot s_k$ is an initial path in I, beginning with a `down(isr)`.

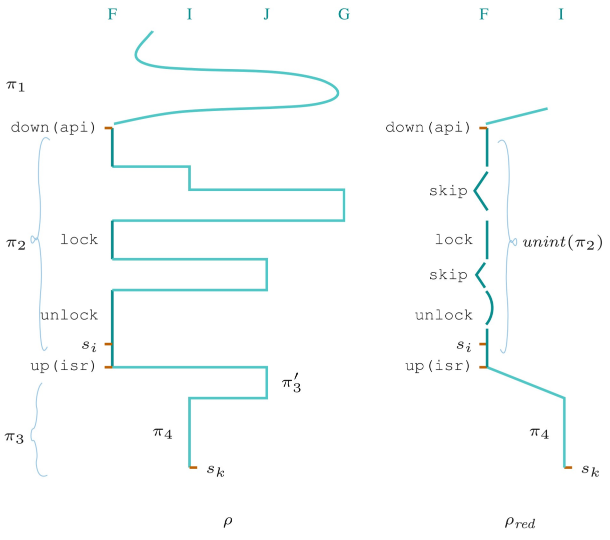

Fig. 7. The execution ρ and its reduction ρ_{red}

Consider the path π_2 that begins in F, contains some interrupting paths that visit other task or fromISR functions, and ends at s_i in F. We define an "uninterrupted" version of π_2, denoted $unint(\pi_2)$, to be the path that replaces each interrupt path by a `skip` statement (note that this non-deterministic branch exists in each interrupt call). In addition, the portion of π_2 that goes through an `UnlockQueue` call may have to change, since the path through an `UnlockQueue` depends on the values of `RxLock` and `TxLock` and these values may have changed by eliding the interrupt paths from π_2. Nevertheless, there is a path through `UnlockQueue` enabled for these new values, and we use these paths to obtain a feasible path $unint(\pi_2)$ through F.

We can now define the reduced path ρ_{red} we need as follows (see Fig. 7): $\rho_{red} = \mathtt{down(sch)} \cdot \mathtt{up(task)} \cdot \mathtt{down(task)} \cdot unint(\pi_2) \cdot s_i \cdot \mathtt{up(isr)} \cdot \pi_4 \cdot s_k$.

We need to argue that ρ_{red} is a valid execution of $M_{F,*,I}$ (here "$*$" stands for an arbitrary task function). We have already argued that

$$\mathtt{down(sch)} \cdot \mathtt{up(task)} \cdot \mathtt{down(task)} \cdot unint(\pi_2) \cdot s_i \cdot \mathtt{up(isr)}$$

is a valid execution of the model. Let the resulting state after this path be u'. It remains to be shown that the path $\pi_4 \cdot s_k$ is a feasible initial path in I, beginning in state u'.

Let us call two states v and w *equivalent* if they satisfy the following conditions: (a) the value of the control semaphores are the same (i.e. $v(\mathtt{isr}) = w(\mathtt{isr})$, etc.), (b) the value of SchedulerSuspended is the same, (c) $v(\texttt{RxLock}) = -1$ iff $w(\texttt{RxLock}) = -1$ and $v(\texttt{RxLock}) \geq 0$ iff $w(\texttt{RxLock}) \geq 0$, and (d) similarly for TxLock. By inspection of the conditionals in any fromISR function J in the model, we observe that the set of feasible initial paths through J, beginning from *equivalent* states is exactly the same. Let u be the resulting state after the prefix $\delta = \pi_1 \cdot \texttt{down(task)} \cdot \pi_2 \cdot s_i \cdot \texttt{up(isr)}$ of ρ, and let v be the resulting state after $\delta \cdot \pi_3' \cdot \texttt{up(isr)}$. To argue that $\pi_4 \cdot s_k$ is a feasible initial path in I beginning from state u', it is thus sufficient to argue that the states u' and v are equivalent.

To do this we argue that (a) u' and u are equivalent, and (b) that u and v are equivalent. To see (a), clearly the value of the control semaphores are identical in u and u'. Further, the value of SchedulerSuspended continues to be 1 in u' as well, as we are only excising paths from π_2 that are "balanced" in terms of setting and unsetting this flag. Finally, if the value of RxLock was -1 in u, it continues to be -1 in u' as well, since an UnlockQueue always resets the flag to -1. If the value of RxLock was 0 or more in u, then we must be between a LockQueue and its corresponding UnlockQueue. In this case the value of RxLock would have been set to 0 in u'. A similar argument holds for TxLock as well, and we are done. To see that the claim (b) holds, we note that only fromISR functions can execute between u and v, and they always either leave the value of RxLock and TxLock intact, or increment them if their value was already ≥ 0.

Thus, ρ_{red} is a valid execution of $M_{F,*,I}$, and it clearly contains a race on s_i and s_k. This completes the proof of the first case we were considering. The second case where SchedulerSuspended is 0 after δ is handled in a similar way. The detailed proof is available in [21].

6 Experimental Results

Of the 69 API functions in FreeRTOS v6.1.1, we model 17 task and 8 fromISR functions. These 25 library functions form the "core"of the FreeRTOS API. The remaining 44 functions are either defined in terms of these core functions, or they simply invoke the core functions with specific arguments, or are synchronization constructs. For example, the functions xQueuePeek and xSemaphoreTake are listed as library functions. However, they are defined in terms of the core function xQueueGenericReceive, which we do model. Thus, modelling these additional functions would be redundant: the races would still be in the core library functions which they invoke.

Our tool-chain is as follows: the user provides a Promela file which models each library function, as well as a template for the reduced models. Next, a Java program creates the "reduced" models (2023 of them in this case) from this Promela template. We then verify these reduced models using Spin. The output of the verification phase is a set of error trails, one corresponding to each interleaving which results in the violation of an assertion. The trails are not in a human readable format, so we need to perform a *simulation* run in Spin using these trails. The output of the simulation run is a set of human readable error traces. However, the number of such traces can be large (around 70,870 were generated during our experiments) and it is infeasible to manually parse them to find the list of races. Instead, we have yet another Java program which scans through these traces and reports the list of unique racing pairs. By a racing pair, we mean statements (s_i, s_k) constituting the race, along with the data-structure `v` involved (we also indicate a trace exhibiting the race).

While the model and the reduction argument need to be tailor-made for different kernel APIs, the software component of the tool-chain is fairly straightforward to reuse. Given a Promela model of some kernel API other than FreeRTOS, where the modelling follows the rules outlined in Sect. 4 (and the model is shown to be reducible), the tool-chain can be used to detect races with minimal changes.

An important point to consider here is the guarantees provided by the Spin tool itself. Spin does *not* exhaustively search for *all* possible violations of an assertion [18]. Instead, it is guaranteed to report *at least* one counter-example if the property is not satisfied. Hence, we make use of an iterative strategy. After each iteration, we change the assertion statement to suppress reporting the detected races again. We continue this process until no further assertion violations are detected by Spin. Thus, by the final iteration, we are *guaranteed* to have flagged every high-level datarace.

All our experiments were performed on a quad-core Intel Core i7 machine with 32 GB RAM, running Ubuntu 14.04. We use Spin version 6.4.5 for our experiments.

Evaluating M_2. The verification of M_2 on our machine took up memory in excess of 32 GB. As a result, we had to kill the verification run prematurely. Even on on a more powerful machine with 4 quad-Xeon processors (16 cores), 128 GB of RAM, running Ubuntu 14.04, the verification run took 39 GB of RAM, while executing for more than 3 hours, before timing out. The total number of tracked states was 4.43×10^8. Using rough calculations, we estimated that the total amount of memory needed to store the full state space of this model (assuming that the size of a single state is 100 bytes) is around 1 TB. On the contrary, while model-checking the 2023 reduced models, the RAM usage never exceeded 9 GB.

Evaluating $\mathcal{M}_{red}$. Recall that each $M_{F,G,I} \in \mathcal{M}_{red}$ comprises 5 processes: the first process runs the task function F, the second runs the task function G, and the third runs the ISR I (excluding the `tick` interrupt), the fourth runs the `tick` interrupt in a loop, while the fifth process runs the scheduler. Since there are 17 task functions and 7 fromISR functions (excluding the `tick`), we generate $17 \times 17 \times 7 = 2023$ models. We model check these reduced models in iterations, suppressing reported races to ensure they are not flagged again in subsequent iterations. In particular, we suppress reporting races on the `SchedulerSuspended` flag, which by design are aplenty. We have manually verified (along with discussion with the FreeRTOS developers) that these races are benign.

In the first iteration, the verification of $\mathcal{M}_{red}$ generated 38 assertion violations. Of these, 10 were false positives, owing to three consecutive read accesses or the conservative modeling of the conditionals. Among the rest, 16 can be definitely classified as harmful. In the second iteration, the tool reported 10 assertion violations, all of them being potentially benign races involving the variable `pxCurrentTCB`.

The cause was an unprotected read of the variable in the function `vTaskResume`. As there were several races involving this statement (it would race with every access, protected or otherwise, of `pxCurrentTCB` in almost all other functions), we supressed races involving this statement. With this change, we performed a third iteration of the verification process, which resulted in no further assertion violations.

Iteration	#Violations	F.P.	Harmful	Benign?	Time
1	38	10	16	12	1.5 hr
2	10	-	-	10	2.4 hr
3	-	-	-	-	1.84 hr

Fig. 8. Experimental evaluation of $\mathcal{M}_{red}$

The FreeRTOS API is quite carefully written. Despite the complexity of the possible task interactions, there are not many harmful races. Among the 16 harmful races detected after the first iteration, most involved the function `vQueueDelete`, which deletes the queue passed to it as argument. Several operations are involved as part of the deletion (removal of the queue from the registry, deallocating the memory assigned to the queue, etc.). Surprisingly, the set of operations, which forms a critical access path for the queue data-structure, is devoid of any synchronization. This causes critical access paths of the queue in other functions, for example `xQueueReceiveFromISR` (which reads the contents of the queue), to interleave with the path in `vQueueDelete`. The race is harmful because functions can potentially observe an inconsistent (partially deleted) state of the queue, which it would not otherwise observe along any linearized execution. We reported this bug to the FreeRTOS developers, and they argue that this is not serious since queue delete operations are rare and are usually performed at the end of the application's lifetime.

The other harmful races involve the `QueueRegistry` data-structure (which is essentially an array), where addition and deletion of items in the `QueueRegistry` can interleave, thereby causing a function to observe an inconsistent state of the registry. Some of the sample races are given in Table. 1.

Table 1. Some sample detected races. "H" indicates harmful races and "PB" indicates possibly benign.

Structure	Library functions involved	Scenario	**H/PB**
`xQueueRegistry`	P1: `vQueueAddToRegistry` P2: `vQueueUnregisterQueue`	While P1 reads the registry, P2 modifies it. Read-write race	H
`userQueue`	P1: `vQueueDelete` P2:`xQueueSend`	P2 sends data to a queue while it is being deleted by P1. Write-write race	H
`uxPriority`	P1: `xTaskCreate` P2: `vTaskPrioritySet`	P1 checks the value of uxPriority while it is being set by P2. The result is never an inconsistent state	PB

A summary of the various statistics of the experiments is given in Fig. 8. The running times are reported in hours. All artifacts of this work are available online at https://bitbucket.org/suvam/freertos.

7 Related Work

Along with work on detecting high-level races and atomicity violations, we also consider work on detecting classical (location-based) races as some of these techniques could be adapted for high-level races as well. We group the work into three categories below and discuss them in relation to our work. The table alongside summarizes the applicability of earlier approaches to our problem setting.

Dynamic Analysis Based Approaches. Artho et al. [3] coined the term "high-level datarace" and gave an informal definition of it in terms of accessing a set of shared variables (what we call a unit in $\mathcal{S}$) "atomically." They define a notion of a thread's "view" of the set of shared variables, and flag potential races whenever two threads have inconsistent views. They then provide a lock-set based algorithm, for detecting view inconsistencies dynamically along an execution. Among the techniques for detecting atomicity violations, Atomizer [11] uses the notion of left/right moving actions, SVD [31] uses atomic regions as subgraphs of the dynamic PDG, AVIO [20] uses interleaving accesses, and [30] uses a notion of trace-equivalence; to check if a given execution exhibits an atomicity violation. Techniques for dynamically detecting classical dataraces use locksets computed along an execution (for example [5,24]), or use the happens-before ordering (for example [7,12]), to detect races. None of these techniques apply directly to the kind of concurrency and synchronization model of FreeRTOS (there are no explicit locks, and no immediate analogue of the happens-before relation). Most importantly, by design these techniques explore only a part of the execution space and hence cannot detect *all* races.

Static Analysis Based Approaches. von Praun and Gross [28] and Pessanha et al. [6] extend the view-based approach of [3] to carry out a static analysis to detect high-level races. The notion of views could be used to obtain an annotation of critical accesses (an $\mathcal{S}$ and $\mathcal{C}$ in our setting) for methods in a library. However definition of views are lock-based and it is not clear what is the corresponding notion in our setting, and whether it would correspond intuitively to what we need.

Flannegan and Qadeer [13] give a type system based static analysis for proving atomicity of methods (i.e. the method's actions can be serialized with respect to interleavings with actions of another thread). The actions of a method are typed as left/right-movers and the analysis soundly infers methods to be atomic. Wang and Stoller [30] extend this type system for lock-free synchronization. In our setting, the notion of left/right-movers is not immediate, and such an approach will likely have a large number of false positives. Static approaches for classical race detection (e.g. [9,27,29]) are typically based on a lockset-based data-flow analysis, where the analysis keeps track of the set of locks that are "must" held at each access to a location and reports a race if two conflicting accesses hold disjoint locks. In [1] the locksets are built into a type system which associates a lock with each field declaration. All the approches above can handle libraries and can detect all races in principle, but in practice are too imprecise (lots of false positives) and often use unsound filters (for example [6, 29]) that improve precision at the expense of missing real races.

Schwarz et al. [25, 26] provide a precise data-flow analysis for checking races in FreeRTOS-like applications that handles flag-based synchronization and interrupt-driven scheduling. The technique is capable of detecting all races, but is applicable only to a given application rather than a library.

Model-Checking Approaches. In [2] Alur et al. study the problem of deciding whether a finite-state model satisfies properties like serializability and linearizability. This approach is attractive as in principle it could be used to verify freedom from atomicity violations. However, the number of threads need to be bounded (hence they cannot handle libraries) and the running time is prohibitive (exponential in number of transitions for serializability, and doubly-exponential in number of threads for linearizability). Farzan and Madhusudan [10] consider a stronger notion of serializability called "conflict serializability" and give a monitoring algorithm to detect whether a given execution is conflict-serializable or not. This also leads to a model-checking algorithm for conflict-serializability based atomicity violations. Again, this is applicable only to applications rather than libraries.

Several researchers have used model-checking tools like Slam, Blast, and Spin to precisely model various kinds of control-flow and synchronization mechanisms and detect errors exhaustively [8, 14–16, 32]. All these approaches are for specific application programs rather than libraries. Chandrasekharan et al. [4] follow a similar approach to ours for verifying thread-safety of a multicore version of FreeRTOS. However, the library there uses explicit locks and a standard notion of control-flow between threads. Further, they use a model which is the equivalent of M_2, which does not scale in our setting Fig. 9.

Earlier Work	FreeRTOS-like Concurrency	Handles Libraries	Can Detect all races
[3], [11], [31], [20], [30], [22], [5], [24], [7], [12], [32]	No	No	No
[27], [9], [29], [28], [6], [13], [30], [1], [4]	No	Yes	Yes
[2], [10], [16], [19]	No	No	Yes
[26], [25]	Yes	No	Yes

Fig. 9. Applicability of earlier work to our setting.

8 Conclusion

We have considered the problem of detecting all high-level races in a concurrent library, as an aid to zeroing-in on atomicity-related bugs in the library. We propose a solution to this problem for the FreeRTOS kernel which is representative of small embedded real-time kernels. The approach is based on model-checking but crucially uses a meta-level argument to bound the size of the model.

References

1. Abadi, M., Flanagan, C., Freund, S.N.: Types for safe locking: static race detection for Java. ACM Trans. Program. Lang. Syst. (TOPLAS) **28**(2), 207–255 (2006)
2. Alur, R., McMillan, K.L., Peled, D.A.: Model-checking of correctness conditions for concurrent objects. Inf. Comput. **160**(1–2), 167–188 (2000)
3. Artho, C., Havelund, K., Biere, A.: High-level data races. Software Test., Verification & Reliab. **13**, 207–227 (2003)
4. Chandrasekaran, P., Kumar, K.B.S., Minz, R.L., D'Souza, D., Meshram, L.: A multi-core version of FreeRTOS verified for datarace and deadlock freedom. In: Proceedings of ACM/IEEE Formal Methods and Models for Codesign (MEMOCODE), pp. 62–71 (2014)
5. Choi, J.-D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., Sridharan, M.: Efficient and precise datarace detection for multithreaded object-oriented programs. In: Proceedings of ACM SIGPLAN Programming Languages Design and Implementation (PLDI), pp. 258–269. ACM, New York (2002)
6. Dias, R.J., Pessanha, V., Lourenço, J.M.: Precise detection of atomicity violations. In: Biere, A., Nahir, A., Vos, T. (eds.) HVC 2012. LNCS, vol. 7857, pp. 8–23. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39611-3_8
7. Dinning, A., Schonberg, E.: Detecting access anomalies in programs with critical sections. In: Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging (PADD), pp. 85–96. ACM, New York (1991)
8. Elmas, T., Qadeer, S., Tasiran, S.: Precise race detection and efficient model checking using locksets. Technical Report MSR-TR–118, Microsoft Research (2005)
9. Engler, D., Ashcraft, K.: Racerx: effective, static detection of race conditions and deadlocks. SIGOPS Oper. Syst. Rev. **37**(5), 237–252 (2003)
10. Farzan, A., Madhusudan, P.: Monitoring atomicity in concurrent programs. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 52–65. Springer, Heidelberg (2008). doi:10.1007/978-3-540-70545-1_8
11. Flanagan, C., Freund, S.N.: Atomizer: a dynamic atomicity checker for multithreaded programs. Sci. Comput. Program. **71**(2), 89–109 (2008)
12. Flanagan, C., Freund, S.N.: Fasttrack: Efficient and precise dynamic race detection. In: Proceedings of ACM SIGPLAN PLDI, pp. 121–133. ACM, New York (2009)
13. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: Proceedings of ACM SIGPLAN Programming Language Design and Implementation (PLDI), pp. 338–349 (2003)
14. Havelund, K., Lowry, M.R., Penix, J.: Formal analysis of a space-craft controller using SPIN. IEEE Trans. Softw. Eng. **27**(8), 749–765 (2001)
15. Havelund, K., Skakkebæk, J.U.: Applying model checking in Java verification. In: Dams, D., Gerth, R., Leue, S., Massink, M. (eds.) SPIN 1999. LNCS, vol. 1680, pp. 216–231. Springer, Heidelberg (1999). doi:10.1007/3-540-48234-2_17

16. Henzinger, T.A., Jhala, R., Majumdar, R.: Race checking by context inference. In: Proceedings of ACM SIGPLAN Programming Language Design and Implementation (PLDI), pp. 1–13 (2004)
17. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
18. Holzmann, G.J.: The model checker spin. IEEE Trans. Softw. Eng. **23**, 279–295 (1997)
19. Kahlon, V., Sinha, N., Kruus, E., Zhang, Y.: Static data race detection for concurrent programs with asynchronous calls. In: ACM SIGSOFT FSE, pp. 13–22. ACM, New York (2009)
20. Lu, S., Tucek, J., Qin, F., Zhou, Y.: AVIO: detecting atomicity violations via access interleaving invariants. In: Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 37–48 (2006)
21. Mukherjee, S., Arunkumar, S., D'Souza, D.: Proving an RTOS kernel free of dataraces. Technical Report CSA-TR-2016-1, Department of Computer Science and Automation, IISc (2016)
22. Qadeer, S., Wu, D.: KISS: keep it simple and sequential. In: Proceedings of ACM SIG-PLAN Programming Languages Design and Implementaion (PLDI), pp. 14–24 (2004)
23. Real Time Engineers Ltd., The FreeRTOS Real Time Operating System (2014)
24. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: a dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. (TOCS) **15**(4), 391–411 (1997)
25. Schwarz, M.D., Seidl, H., Vojdani, V., Apinis, K.: Precise analysis of value-dependent synchronization in priority scheduled programs. In: McMillan, K.L., Rival, X. (eds.) VMCAI 2014. LNCS, vol. 8318, pp. 21–38. Springer, Heidelberg (2014). doi:10.1007/978-3-642-54013-4_2
26. Schwarz, M.D., Seidl, H., Vojdani, V., Lammich, P., Müller-Olm, M.: Static analysis of interrupt-driven programs synchronized via the priority ceiling protocol. In: Proceedings ACM SIGPLAN-SIGACT Principles of Programming Languages (POPL), pp. 93–104 (2011)
27. Sterling, N.: WARLOCK - a static data race analysis tool. In: Proceedings of Usenix Winter Technical Conference, pp. 97–106 (1993)
28. von Praun, C., Gross, T.R.: Static detection of atomicity violations in object-oriented programs. J. Object Technol. **3**(6), 103–122 (2004)
29. Voung, J. W., Jhala, R., Lerner, S.: RELAY: static race detection on millions of lines of code. In: Proceedings of ESEC/SIGSOFT Foundation of Software Engineering (FSE), pp. 205–214 (2007)
30. Wang, L., Stoller, S.D.: Runtime analysis of atomicity for multithreaded programs. IEEE Trans. Softw. Eng. **32**(2), 93–110 (2006)
31. Xu, M., Bodík, R., Hill, M.D.: A serializability violation detector for shared-memory server programs. In: Proceedings of ACM SIGPLAN Programming Language Design and Implementation (PLDI), pp. 1–14 (2005)
32. Zeng, R., Sun, Z., Liu, S., He, X.: McPatom: a predictive analysis tool for atomicity violation using model checking. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 191–207. Springer, Heidelberg (2012). doi:10.1007/978-3-642-31759-0_14