A Runtime Verification Tool for Detecting Concurrency Bugs in FreeRTOS Embedded Software

Sara Abbaspour Asadollah*, Daniel Sundmark*, Sigrid Eldh†, Hans Hansson*
*Mälardalen University, Västerås, Sweden
{sara.abbaspour, daniel.sundmark, hans.hansson}@mdh.se
†Ericsson AB, Kista, Sweden
sigrid.eldh@ericsson.com

***Abstract*—This article presents a runtime verification tool for embedded software executing under the open source real-time operating system FreeRTOS. The tool detects and diagnoses concurrency bugs such as deadlock, starvation, and suspension-based-locking. The tool finds concurrency bugs at runtime without debugging and tracing the source code. The tool uses the Tracealyzer tool for logging relevant events. Analysing the logs, our tool can detect the concurrency bugs by applying algorithms for diagnosing each concurrency bug type individually.**

In this paper, we present the implementation of the tool, as well as its functional architecture, together with illustration of its use. The tool can be used during program testing to gain interesting information about embedded software executions. We present initial results of running the tool on some classical bug examples running on an AVR 32-bit board SAM4S.

I. Introduction

Runtime Verification is concerned with checking a trace of the program against properties described in some logic [1]. When a property is violated, the program is able to take action in order to deal with the situation. Moreover, since the check is done during the program execution only practically reachable states are considered. Runtime verification is useful in both testing and monitoring. For instance, if a user comes up with a test case [2] and he/she wants to exercise a possible bug, then a runtime verification tool can be considered to automate the creation of oracles for detecting errors. Further, if a user is interested to take action in response to the violation of a property then he/she could consider runtime verification as a monitor while it may be used to define how the program reacts to bugs, possibly steering it to the correct behavior [3].

Due to increasing software system complexity, there is renewed interest in implementing tools for detecting faults and managing recovery from them during runtime. Concurrent programming also increases the complexity of different types of software.

Meanwhile, concurrent programming introduces the possibility of new types of software bugs, known as concurrency bugs [4]. The bugs typically appear under very specific situations which can nondeterministically occur e.g., due to thread interleavings between shared memory accesses. The effects of the bugs spread through the software and potentially cause the software to crash, hang or produce incorrect output. Such nondeterministic bugs are typically considered to be problematic errors [5], [6], since they are difficult to reproduce. A goal of the proposed tool is monitoring of embedded software behavior in order to determine whether it complies with the properties of the different concurrency bugs. To the best of our knowledge, this research is the first effort implement a tool for detecting concurrency bugs in embedded software based on the FreeRTOS [1] platform.

A. Paper Contributions

In summary, the following are our main contributions:

1) We propose a functional architecture of a runtime verification tool for embedded software in order to detect concurrency bugs.
2) We implement a tool based on the proposed functional architecture, covering deadlock, starvation and suspension-based-locking bugs for software running under FreeRTOS.
3) We evaluate the implemented runtime verification tool for detecting the concurrency bugs on FreeRTOS running on a SAM4S Xplained platform.

B. Paper Layout

The rest of the article is organized as follows: We survey related work in Section II. FreeRTOS and Tracealyzers' backgrounds as well as the terminologies, are presented in Section III. The functional architecture of our proposed tool in addition to our proposed algorithms are described in Section IV. The tool user interface together with three different examples are illustrated in Section V. Finally, we conclude the study and highlight the direction of future work in Section VI.

II. Related Work

Java PathExplorer (JPAX) is a runtime verification tool proposed by Havelund and Rosu [8], [9] for monitoring the execution of sequential and concurrent Java programs. The

[1]FreeRTOS is an open source, portable, and real-time operating system kernel [7].

prototype of Java PathExplorer has been applied to the executive module of the NASA Ames planetary Rover K9 [8]. The general concept of the tool concerns extracting events while the program is executing and then analysing these events with a remote observer process. JPAX instruments Java byte code to send a set of relevant events to the observation module that performs two kinds of verifications: 1) logic based monitoring and 2) error pattern analysis. Logic-based monitoring is a kind of specification based monitoring which counts upon an underlying logic and the user can express any application dependent, logical requirements. Error pattern analysis implements more or less standard programming language dependent algorithms, e.g., exploring execution trace to detect potential concurrency errors, including Deadlocks and Data races, even they do not explicitly occur in the trace.

Falcon is another tool for on-line monitoring and steering of large-scale parallel programs [10]. Its monitoring subsystem consists of higher-level view specification and low-level sensor specification. Programmers define application-specific sensors for capturing the program behavior and attributes during runtime. Falcon has another subsystem to permit users for implementing on-line display system to graphically display data structures, runtime program behaviors, and performance information. Falcon is designed for distributed systems and its implementation relies on the C threads library on several hardware platforms.

Java with Assertions (Jass) is a monitoring approach developed for sequential and concurrent systems written in Java [11]. Jass translates annotations to programs written in Java into pure Java code. Compliance with the specified annotations is dynamically tested during runtime. It checks specification violations dynamically at runtime by adding assertions which provide the specification of the program. Assertions are boolean expressions of Java with certain keywords and quantifications over finite sets. They are in the form of class invariants, loop invariants, method post and pre-conditions and additional checks which can be inserted into every part of the code. Jass is able to detect possible interferences in a parallel program by having the thread in Jass classes which start in the main method. When an assertion in one thread becomes invalid through statements in another thread then Jass is able to detect it.

In summary, there are a few runtime verification tools available, but none is a runtime verification tool for embedded software to detect concurrency bugs. For instance, XPA is a runtime verification tool for monitoring and detecting potential concurrency errors in Java programs. From our understanding, it cannot detect these concurrency bugs for embedded software running of freeRTOS framework. In addition, the proposed tool cannot detect other types of concurrency bugs such as *Starvation* and *Suspension* bugs. Similarly, Jass is a monitoring approach considering Java applications and not able to detect if the interferences are protected by synchronization methods [11] while it is not the case for our proposed tool. Moreover, the other tool (Falcon) relies on C libraries, however our proposed tool is designed and implemented to monitor the embedded software in order to detect special type of bugs (concurrency bus).

III. Preliminaries

In this section, we present the *FreeRTOS*, *Tracealyzer* and the *terminology* used in this article.

A. FreeRTOS

FreeRTOS is an open source real-time operating system. It is licensed under a modified GPL and developed by Real Time Engineers Ltd. FreeRTOS is available for 35 different hardware architectures ranging from 8-bit to 32-bit microcontrollers, particularly targeting small embedded systems [7].

The FreeRTOS kernel supports cooperative, preemptive and hybrid scheduling, supporting static and dynamic task priorities. Round-Robin (RR) is considered in its fixed-priority preemptive scheduling.It supports multitasking by any number of tasks [12]. The context-switching in FreeRTOS kernel is efficient and its library contains efficient commands such as binary, counting and recursive semaphores; Mutexes for resource protection and synchronization; and queues for message passing among tasks.

B. Tracealyzer

Tracealyzer, developed by Percepio AB since 2004 [13], is a stand-alone application for tracing and visualizing embedded software executions. Tracealyzer for FreeRTOS is designed for 32-bit processors, including MCUs, and is configured for minimal RAM and ROM usage. It has plug-ins and integrations for common development tools such as Atmel Studio 7, SEGGER J-Link debug probes, Microchip MPLAB X IDE and IAR Embedded Workbench.

Tracealyzer offers more than 25 graphical views of system behavior in order to give insight into the firmware at runtime, making it easier to reveal errors and bottlenecks, speed up debugging, validation and optimization. It offers two main tracing modes: snapshot and streaming mode. In snapshot mode, the trace data is kept in a target-side RAM buffer until explicitly uploaded. In streaming mode, the data is transferred continuously to the host PC, allowing for very long trace durations.

C. Terminology

The terminology for software problems are not entirely consistent and different terms like fault, error, bug, failure are sometimes used interchangeably. **Bug** is a problem which impairs or prevents the functions of the software [14]. In this article, we use the term *bug* to refer to an observed behavior in the embedded software under test, although this may not be entirely in line with the mentioned terminology.

Leucker and Schallhart defined **runtime verification** [15] as "the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a run of a system under scrutiny satisfies or violates a given correctness property". In other words, runtime verification is an analysis and execution approach based on extracting information from a running system

and using it to detect, and possibly react to, observed behaviors satisfying or violating certain properties.

Concurrency bugs are problems that may not occur in sequential software. Examples of concurrency bugs include deadlocks and data races. In our previous study [16] we proposed a taxonomy for concurrency bugs by classifying the bugs in a common structure considering relevant observable properties. This classification is based on an assumption that a concurrency bug has occurred, i.e., the properties of each bug may not be sufficient to identify a bug, but once a concurrency bug has occurred the properties can be used to uniquely identify which type of bug it is [17]. Three classes of the proposed taxonomy are considered in this study viz., *Deadlock*, *Starvation* and *Suspension-based locking (also named Blocking suspension)*. The detailed information of each type and their properties are described in [16], [6].

Deadlock is a condition where a task in a program cannot proceed because it needs to obtain a resource which is held by another task while itself is holding a resource that the other task(s) needs [6]. During deadlock, all involved tasks are in a waiting state.

Starvation is a condition where a task in a program delayed because other processes are always given preference [6] while this delay is not accepted by the program's users (or testers or developers). At least one of the involved tasks remains in the ready queue during starvation bug.

Suspension-based locking occurs when a calling task waits for an unacceptably long time in a queue to acquire a lock for accessing a shared resource [6]. We call it *Suspension* bug in this article.

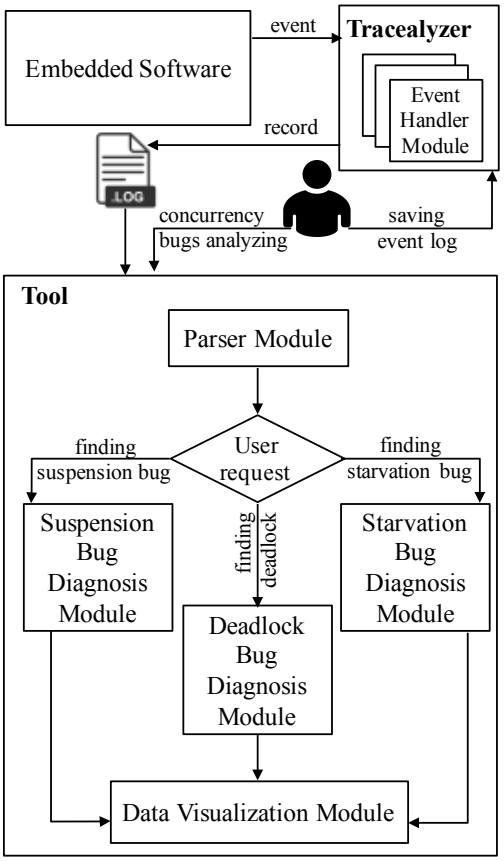

Figure 1. Functional architecture of the runtime verification for detecting concurrency bugs in embedded software

IV. Tool Implementation

The primary contribution of this paper is a runtime verification tool for concurrency bug detection in FreeRTOS embedded software. The implementation details of the tool are presented in this section. We describe the functional architecture of the proposed tool with detailed information of each module. We also describe the algorithms used for detection of each type of concurrency bug.

A. Functional Architecture

As shown in Figure 1, the tool's functional architecture is comprised of five separate modules, viz., *Parser Module*, *Starvation Bug Diagnosis Module*, *Deadlock Bug Diagnosis Module*, *Suspension Bug Diagnosis Module*, and *Data Visualisation Module*.

During execution of the target system, Tracealyzer will trace the system-level control flow events (e.g., task switches, synchronization calls, etc.) of the embedded software execution. Upon a user request, the Tracealyzer will save the event log file. The log file has a defined template which we found by contacting the Percepio AB company [13]. The collected data can describe the fraction of the execution time spent in each task over some period of time. The log files tend to be large, thus an effective analysis is needed. If the user sends a request to the tool in order to analyze the log file for detecting concurrency bugs, then first the *Parser Module* will parse the log file and extract (or calculate) the relevant data. The following list illustrates each field and Table I presents the format of extracted data from *Parser Module*.

- 'TaskName" is the name of one of the available task(s) saved in the log file.
- "Status" is the current status of the task (*Running*, *Suspended* or *Ready*). We analyse the extracted information and each line of log file to set the value. For instance, when we reach a context switch in the log file, we will update the status of other task(s) to *Ready* (if it is not *Suspended*) and set the status of current task to *Running*.
- "FromTime" indicates the time since the status of the task was changed.
- "WaitingReason" shows the reason which holds the task in *Suspended* status (either of *User Requested*, *Semaphore* or *Queue*). We set this value by analysing each line of the log file. For instance, if the status of a task changed to *Suspended* due to asking for a semaphore while the semaphore is already taken by other task then we set this value to *Semaphore*.
- If a task is suspended the "WaitingForObject" shows which object is keeping the task in waiting. We set this value by analysing each line of the log file. For instance, if the status of a task changed to *Suspended* due to asking

for semaphore5 while the semaphore5 is already taken by other task then we set this value to *Semaphore5*.

- The current object which holds the task is saved in "TakenObject". We can check the objects are taken by each task during parsing the log file. For instance, when we reach *XSemaphoreTake sem3* in the log file, we will set the "TakenObject" value to *sem3* and if we reach the *XSemaphoreGive sem3* for the same task after then, we will update the value of this filed by removing the *sem3* from the field's value.
- "MaxInReady" represents the maximum time that a task is spending in *Ready* status. We will calculate the maximum time that each task has spend in *Ready* status during parsing the log file.
- "MaxInReady FromTime" presents the start time of the "MaxInReady" for the task.
- "MaxInSuspend" shows the maximum time that a task is continuously spending in *Suspended* status.
- The "MaxInsuspend FromTime" indicates the start time of the "MaxInSuspend" for the task. We will calculate the maximum time that each task has spend in *Suspended* status during parsing the log file

For each of the three bug diagnosis modules, the extracted data will be analysed and compared to the corresponding bug property patterns. In case there is a match, the bug (and its type) is reported.

For the starvation- and suspension-type bugs, a *delay tolerance* will be requested from the user by the tool. This value represents the maximum acceptable time for the task to stay in *Ready* and *Suspended* states respectively. A user can set distinct values for all tasks independently of each other.

Data Visualisation Module is a combination of functions and procedures for displaying the results of the *Parser* , *Starvation Bug Diagnosis*, *Deadlock Bug Diagnosis* and *Suspension Bug Diagnosis* modules.

B. Overview of Bug Detection Algorithms

The pseudocode depicted in Algorithm 1, 2 and 3 respectively, provides overviews of the inner workings of the *Deadlock Bug Diagnosis*, *Starvation Bug Diagnosis* and *Suspension Bug Diagnosis* modules.

The Deadlock detection algorithm (Algorithm 1) takes input in the form of a dataset (suspendedTasks) which contains extracted data from *Parser Module* for the tasks that are in state Suspended and with WaitingReason Semaphore or Queue. Table I shows the general data structure. "deadlockSet" holds the length of the set of tasks that have the deadlock bug. The variables "deadlockSet", "isDeadlock" and "possibleBug" are global variables that are used by both procedures, DeadlockDetection and findNode. It also returns a dataset (ResultDataset), which contains the information of the tasks causing the deadlock.

The pseudocode of the algorithm for detecting starvation bugs is presented in Algorithm 2. Similar to Algorithm 1, this algorithm has an input dataset (extractedData) and an output dataset (ResultDataset). As explained in Section IV-A,

Algorithm 1 for detecting Deadlock bugs

```
1: procedure DeadlockDetection (suspendedTasks) {
2: suspendedTasks ← select the tasks with Status = 'Suspended' and
                    WaitingReason =! 'User request' from suspendedTasks
3: if (suspendedTasks is not null) then
4: { deadlockSet = 1 , isDeadlock = False
5:   while (suspendedTasks is not null)
6:   { waitObj = suspendedTasks[0][waitingForObject]
7:     possibleBug ← findNode(suspendedTasks[0], waitObj , deadlockSet )
8:     possibleBug ← null
9:     suspendedTasks remove the fist task }*/ end of while /  } */ end of if /
10: return ResultDataset }
```

```
100: procedure findNode(checkingTask, waitObj , deadlockSet ) {
101: possibleBug.add(checkingTask)
102: checkingDataset ← select the tasks with waitingForObject =
checkingTask[takenObj] from suspendedTasks
103: if (checkingDataset.rows.count > 0)
104: { for (i= 0 to checkingDataset number of tasks -1; i++)
105:   { if (checkingDataset[i][takenObj] contains waitObj )
106:     { possibleBug.add(checkingDataset[i])
107:       isDeadlock = True }
108:     else
109:     { possibleBug ← findNode(checkingDataset[i], waitObj,
                                  deadlockSet )
110:       possibleBug remove the last task}
111:     if (possibleBug is not null)
112:       possibleBug remove the last task} */ end of for
      } */ end of if (checkingDataset is not null)/
113: if (isDeadlock = True)
114: { ResultDataset.add (all tasks from possibleBug with all id = deadlockSet)
115:   deadlockSet ++ , isDeadlock = False  }
116: return possibleBug }
```

the *Parser Module* is responsible for calculating the maximum time that a task has spent in *Ready* state from the log file. This value is essential to detect the Starvation bugs in Algorithm 2. The algorithm shows that the delay tolerance value can be asked from a user for each task (Line 3). Then the algorithm will compare the MaxInReady of each task to its delay tolerance value given by the user. If the MaxInReady of each task is bigger than user acceptation of the task then the task is subject to starvation and we add its data to the ResultDataset.

Algorithm 2 for detecting Starvation bugs

```
1: procedure StarvationDetection (extractedData)
2: TaskNameSet ← select all TaskName from extractedData
3: read UserDelayTolerance for each task of TaskNameSet
4: for (i= 0 to count of TaskNameSet ; i++) {
5:   selectedTask ← select the task with TaskName = TaskNameSet[i] from extractedData
6:   if (selectedTask[MaxInReady] ≧ TaskNameSet (UserDelayTolerance )[i])
7:     selectedTask add to ResultDataset  } /* end of for
9: return ResultDataset
```

Algorithm 3 presents the pseudocode of the algorithm for detecting Suspension bugs, also with an input dataset (extractedData) and an output dataset (ResultDataset). The calculated value for MaxinWaiting field is essential to detect the Suspension bugs in this algorithm. MaxinWaiting value is calculated in *Parser Module* to shows the maximum time that a task has waited in *Suspended* state. If the MaxinWaiting for each task is bigger than user expectation (delay tolerance value) then the task is subject to a suspension bug and the task's data will be added to the ResultDataset.

Table I
THE DATA STRUCTURE OF EXTRACTED DATA FROM *Parser Module*

Field	Type	Description
TaskName	String	Represents text as a sequence of UTF-16 code units
FromTime	String	The value is based on the H_1H_2:M_1M_2:S_1S_2.$m_1m_2m_3$.$\mu_1\mu_2\mu_3$ format. That represents hour in two digits separated by ":" from minute in two digits separate by ":" from second in two digits separate by "." from millisecond in three digits separated by "." from microsecond in three digits.
Status	Enum	The value can be one of the value from the set {Running, Suspended, Ready}
WaitingReason	Enum	The value can be one of the value from the set {User Requested, Semaphore, Queue}
WaitingForObject	String	The value can be ether the name of an object or "No Object"
TakenObject	String	The value can be a name of object. If there is more than one object then all names will add to this field and will separate by ";"
MaxInReady	Long	Unsigned number with the range of 0 to 4,294,967,295
MaxInReady FromTime	String	The value is based on the H_1H_2:M_1M_2:S_1S_2.$m_1m_2m_3$.$\mu_1\mu_2\mu_3$ format. That represents hour in two digits separated by ":" from minute in two digits separate by ":" from second in two digits separate by "." from millisecond in three digits separated by "." from microsecond in three digits.
MaxInSuspend	Long	Unsigned number with the range of 0 to 4,294,967,295
MaxInsuspend FromTime	String	The value is based on the H_1H_2:M_1M_2:S_1S_2.$m_1m_2m_3$.$\mu_1\mu_2\mu_3$ format. That represents hour in two digits separated by ":" from minute in two digits separate by ":" from second in two digits separate by "." from millisecond in three digits separated by "." from microsecond in three digits.

Algorithm 3 for detecting Suspension bugs

```
1: procedure SuspensionDetection (extractedData)
2: TaskNameSet ← select all TaskName from extractedData
3: read UserDelayTolerance for each task of TaskNameSet
4: for (i= 0 to count of TaskNameSet ; i++) {
5:   selectedTask ← select the task with TaskName = TaskNameSet[i] from extractedData
6:   if (selectedTask[MaxInWaiting] ≧ TaskNameSet (UserDelayTolerance )[i])
7:       selectedTask add to ResultDataset  } /* end of for
9: return ResultDataset
```

V. TOOL UI AND EXAMPLES

Figure 2 presents the snapshot of the tool after running the *Parser Module*. This module also is able to differentiate between the kernel and user tasks. If a user wants to detect concurrency bugs among the user tasks then he/she can select the "User Task" button. After browsing and selecting the saved log file, the tool (*Parser Module*) will add the extracted data into a dataset and present them via a data grid.

To run the initial evaluation and to demonstrate the result of analysing and detecting the three mentioned concurrency bugs, we developed practical examples with injected concurrency bugs with the help of expert embedded software developers. Each example is developed to inject one of the deadlock, starvation and suspension bugs. Test Scenario 1 (Section V-A) illustrates the evaluation for deadlock, Test Scenario 2 (Section V-B) demonstrates the evaluation for starvation and Test Scenario 3 (Section V-C) shows the output of the tool after injecting a suspension bug example into the embedded software code. Then we traced and saved the event log files during software execution time for each example. All measurements are performed on the target platform Xplained SAM4S [18].

A. Test Scenario 1

In this section, we explain the generated deadlock example implemented in Atmel Studio 7[2] by presenting the tool's output after tracing and analysing the injected deadlock bug. For clarity, we use a simple example. As shown in Figure 3, the deadlock bug example is implemented by creating two tasks, both of which have access to two shared recourses (semaphores). The tasks are created by calling the xTaskCreate() and the scheduler started by calling vTaskStartScheduler(). The deadlock bug example contains two tasks with the same priority. Each task is able to run a function while both of them are designed to run concurrently.

We use binary semaphores in the examples, created by vSemaphoreCreateBinary(), and taken and released by xSemaphoreTake() and xSemaphoreGive(), respectively. xSemaphoreTake() has a parameter timeout parameter determining how long to wait for the semaphore. We set this parameter to portMAX_DELAY which makes the task to block indefinitely without a timeout. In order to connect the examples to the Tracealyzer, we add trace a recorder library from Tracealyzer, and specify the Tracealyzer recording mode to Snapshot.

In order to start the trace recording by Tracealyzer, we use *vTraceEnable(TRC_START)* which is added at the beginning of the non-returning function (typically main()).

We execute the injected embedded software and save the event log file by Tracealyzer during execution. The save log file is considered as a raw data for our tool, thus, we parse and analyse the obtained log file. Figure 6 shows the output of our tool after analysing the extracted data from the log file in order to detect Deadlock bugs. As we expected, Task1 and Task2 are the causes of detected deadlock bug.

B. Test Scenario 2

We implement a simple starvation example by calling three tasks while all of them access to two shared variables. The following APIs define the shared variable in the Atmel Studio where *gCounter* defined for keeping a number (with the value of 100 as the initial value) and *cTextGlobalVariable* is supposed to keep the task's name (it initially is empty).

```
int gCounter = 100;
char *cTextGlobalVariable = "";
```

[2]Atmel Studio 7 is an integrated development platform (IDP) for developing and debugging Atmel SMART ARM-based and Atmel AVR microcontroller (MCU) applications [19].

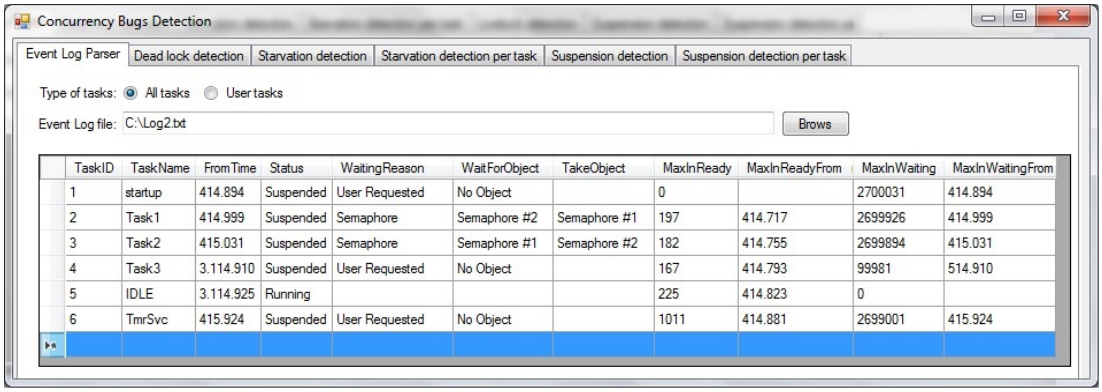

Figure 2. A snapshot of the tool's output after parsing an event log file

```
void vTaskFun1()
{  while (true)
   {
     xSemaphoreTake(xBinarySemaphore1, portMAX_DELAY );
     taskYIELD();
     xSemaphoreTake(xBinarySemaphore2, portMAX_DELAY );
     printf("Function 1");
     xSemaphoreGive(xBinarySemaphore2);
     xSemaphoreGive(xBinarySemaphore1);
     taskYIELD();
   }
}
```

```
void vTaskFun2()
{  while (true)
   {
     xSemaphoreTake(xBinarySemaphore2, portMAX_DELAY );
     taskYIELD();
     xSemaphoreTake(xBinarySemaphore1, portMAX_DELAY );
     printf("Function 2");
     xSemaphoreGive(xBinarySemaphore1);
     xSemaphoreGive(xBinarySemaphore2);
     taskYIELD();
   }
}
```

Figure 3. A simple *Deadlock* bug example implemented in Atmel Studio injected to a FreeRTOS embedded software

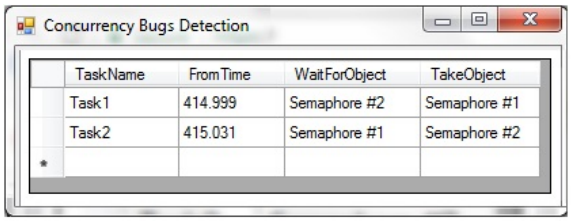

Figure 4. A snapshot of the tool's output for detecting a *Deadlock* bug

The injected Starvation bug example consists of three tasks (TaskA, TaskB, and TaskC) with two priorities. The following block is added in the main() function in order to create and schedule these three tasks. It shows TaskA and TaskB have same and higher priority (2) and TaskC has lower priority (1). The FreeRTOS scheduler ensures that the task placed into the *Running* state is always the highest priority task and the *Ready* state tasks with equal priority will share the available processing time using a time sliced round robin scheduling scheme.

```
xTaskCreate(vTaskFun1, "TaskA", 512, , 2, NULL);
xTaskCreate(vTaskFun2, "TaskB", 512, , 2, NULL);
xTaskCreate(vTaskFun3, "TaskC", 512, , 1, NULL);
vTaskStartScheduler();
```

The functions in Figure 5 indicate that TaskA will add one to the *gCounter* and assign "Task A" to the *cTextGlobalVariable* and then send a request for context switching to another task (by taskYIELD()). TaskB will subtract one from the *gCounter* variable, assign "Task B" to the *cTextGlobalVariable* and then send a request to another task for context switching. TaskC will display the current values of *cTextGlobalVariable* and *gCounter* in the screen and then send a request to next task for context switching.

```
void vTaskFun1()
{  while (true)
   {
     xSemaphoreTake(xBinarySemaphore1, portMAX_DELAY );
     cTextGlobalVariable = "Task A ";
     gCounter = gCounter + 1;
     xSemaphoreGive(xBinarySemaphore1);
     taskYIELD();
   }
}
```

```
void vTaskFun2()
{  while (true)
   {
     xSemaphoreTake(xBinarySemaphore1, portMAX_DELAY );
     cTextGlobalVariable = Task B ";
     gCounter = gCounter - 1;
     xSemaphoreGive(xBinarySemaphore1);
     taskYIELD();
   }
}
```

```
void vTaskFun3()
{  cTextGlobalVariable = "Task C ";
   while (true)
   {
     printf("%s ", cTextGlobalVariable);
     printf("%d\n", gCounter);
     taskYIELD();
   }
}
```

Figure 5. A simple *Starvation* bug example implemented in Atmel Studio injected to a FreeRTOS embedded software

Suppose we expect to get the result of the injected example before 0.2 seconds (200000 microseconds). Otherwise, we consider a concurrency bug. In order to investigate if a bug has occurred, we execute the embedded software and save the event log file by Tracealyzer during execution. Then we parse and analyse the obtained log file with the proposed tool. Figure 6 presents the output of the proposed tool. As it is shown the maximum time that TaskC is spent in *Ready* status is longer than the user delay tolerance (0.2 second) thus it can be the cause of detected Starvation bug.

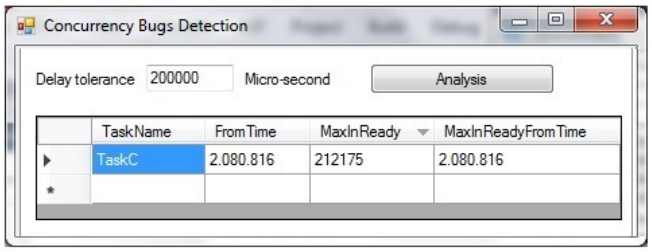

Figure 6. A snapshot of the tool's output for detecting a *Starvation* bug

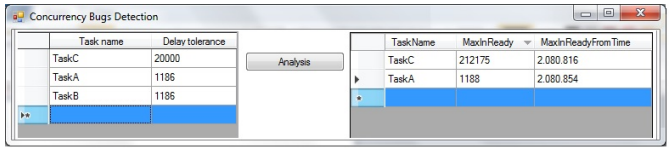

Figure 7. A snapshot of the tool's output for detecting a *Starvation* bug

C. Test Scenario 3

In this third example, we explain a simple suspension bug and the proposed tool's output after tracing and analysing the injected Suspension bug into an embedded software. The injected Suspension example contains three tasks with the same priority (1).

The injected example is race-free, meaning that we manage to protect the shared variables by using a binary semaphore (*xBinarySemaphore1*) and any data corruption during runtime will not be caused by a data race.

However suppose we expect to get the result of TaskM and then TaskN respectively. Figure 9 presents a snapshot of the example's output and it shows TaskM and TaskN did not execute in the expected order. Therefore, it shows a data corruption has happened due to getting the result of TaskN in two consecutive printouts.

In order to detect this bug with our proposed tool, we execute the embedded software and save the event log file by Tracealyzer during execution. Then we parse and analyse the obtained log file with the tool. Figure 10 indicates the output of the tool by setting 200 microseconds as our delay tolerance for each task. In the other words, if a task stays longer than 200 microseconds in suspended state then we expect a suspension bug possibility for the task. Figure 10 shows TaskM and TaskN stayed in suspension state longer than 200 microseconds while TaskM waited longer in compare to TaskN, therefore, TaskN could be the cause of the detected bug. The tool also is able to detect the cause of Suspension bug by setting different delay tolerance for each task.

```
void vTaskFun1()
{   while (true)
    {
      xSemaphoreTake( xBinarySemaphore1, portMAX_DELAY );
      cTextGlobalVariable = "Task M ";
      gCounter = gCounter + 1;
      xSemaphoreGive( xBinarySemaphore1);
      taskYIELD();
    }
}
```

```
void vTaskFun2()
{   while (true)
    {
      xSemaphoreTake( xBinarySemaphore1, portMAX_DELAY );
      cTextGlobalVariable = Task N ";
      gCounter = gCounter - 1;
      xSemaphoreGive( xBinarySemaphore1);
      taskYIELD();
    }
}
```

```
void vTaskFun3()
{   while (true)
    {
      xSemaphoreTake( xBinarySemaphore1, portMAX_DELAY );
      printf("%s %d\n ", cTextGlobalVariable, gCounter);
      xSemaphoreGive( xBinarySemaphore1);
      taskYIELD();
    }
}
```

Figure 8. A simple *Suspension* bug example implemented in Atmel Studio injected to a FreeRTOS embedded software

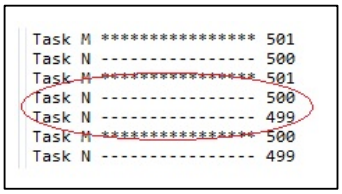

Figure 9. A snapshot of the example's output in Atmel Studio

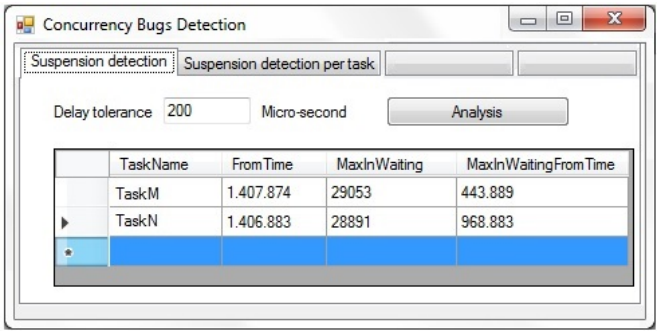

Figure 10. A snapshot of the tool's output for detecting a *Suspension* bug

VI. Conclusion and Future Work

In this article, we proposed a functional architectural model for runtime verification of three types of concurrency bugs,

i.e., *Deadlock*, *Starvation* and *Suspension* bugs. These bugs are detectable if their properties can be observed, monitored and derived from the collected runtime verification data. This model act as the basis for developing a tool which can detect and identify concurrency bugs during execution of embedded software. Thus, we implemented a tool based on the proposed model using the FreeRTOS framework.

We have verified our implementation and performed an initial evaluation on an ARM Cortex-M-based micro-controller. We have checked the tool by injecting three predefined types of concurrency bugs in Atmel Studio and saved the log file during execution time usin the Tracealyzer tool. Then we used the log file as input to our tool in order to detect the injected bugs. As described in SectionV, the implemented tool was able to detect the injected bugs.

Evaluating the tool with complete embedded software (and not only inject predefined bugs into some part of the software) in order to find the real bugs would be an interesting future direction. Moreover, extending the proposed functional architecture model and tool for detection other type of concurrency bugs (i.e., *Data race*, *Atomicity violation*, *Order violation* and *Livelock*) based on their distinct properties would be another interesting direction for future work.

ACKNOWLEDGMENT

We acknowledge the Swedish Research Council (VR, EXACT project) for supporting this work. Additionally, we would like to thank Percepio for giving permission to use their tool (Tracealyzer) and for their valuable discussions, specially *Dr. Johan Kraft* and *Niclas Lindblom*.

REFERENCES

[1] M. d'Amorim and K. Havelund, "Event-based runtime verification of java programs," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–7, May 2005.

[2] C. Artho, D. Drusinksy, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, G. Roşu, and W. Visser, *Experiments with Test Case Generation and Runtime Analysis*. Springer Berlin Heidelberg, 2003, pp. 87–108.

[3] F. Chen and G. Rou, "Towards monitoring-oriented programming: A paradigm combining specification and implementation," *Electronic Notes in Theoretical Computer Science*, vol. 89, no. 2, pp. 108 – 127, 2003.

[4] D. A. Weiser, *Hybrid Analysis of Multi-threaded Java Programs*. ProQuest, 2007.

[5] P. Godefroid and N. Nagappan, "Concurrency at Microsoft: An exploratory survey," in *CAV Workshop on Exploiting Concurrency Efficiently and Correctly*, Princeton, USA, 2008.

[6] S. Abbaspour Asadollah, D. Sundmark, S. Eldh, and H. Hansson, "Concurrency bugs in open source software: a case study," *Journal of Internet Services and Applications*, vol. 8, no. 1, p. 4, Apr 2017.

[7] "About freertos," https://www.freertos.org/RTOS.htmll, accessed: 201-02-15.

[8] K. Havelund and G. Rosu, "Java pathexplorer-a runtime verification tool," 2001.

[9] K. Havelund and G. Roşu, "Monitoring java programs with java pathexplorer," *Electronic Notes in Theoretical Computer Science*, vol. 55, no. 2, pp. 200–217, 2001.

[10] W. Gu, G. Eisenhauer, E. Kraemer, K. Schwan, J. Stasko, J. Vetter, and N. Mallavarupu, "Falcon: On-line monitoring and steering of large-scale parallel programs," in *Frontiers of Massively Parallel Computation, 1995. Proceedings. Frontiers' 95., Fifth Symposium on the*. IEEE, 1995, pp. 422–429.

[11] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim, "Jass java with assertions," *Electronic Notes in Theoretical Computer Science*, vol. 55, no. 2, pp. 103 – 117, 2001.

[12] R. Inam, J. Carlson, M. Sjödin, and J. Kunar, "Predictable integration and reuse of executable real-time components," *Journal of Systems and Software*, vol. 91, pp. 147 – 162, 2014.

[13] "Tracealyzer for freertos," https://percepio.com/tz/freertostrace/, accessed: 2018-02-18.

[14] S. A. Asadollah, "Bugs and debugging of concurrent and multicore software," pp. 1–147, May 2016. [Online]. Available: http://www.es.mdh.se/publications/4434-

[15] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293 – 303, 2009.

[16] S. Abbaspour A., H. Hansson, D. Sundmark, and S. Eldh, "Towards classification of concurrency bugs based on Observable properties," in *International Workshop on Complex Faults and Failures in Large Software Systems*, Italy, 2015.

[17] S. Abbaspour Asadollah, D. Sundmark, S. Eldh, H. Hansson, and W. Afzal, "10 years of research on debugging concurrent and multicore software: a systematic mapping study," *Software Quality Journal*, pp. 1–34, 2016.

[18] "Atmel: Smart sam4s series mcu," http://ww1.microchip.com/downloads/en/devicedoc/atmel-11177-atmel-smart-sam4s_flyer.pdf, accessed: 2018-02-10.

[19] "Atmel studio," https://www.microchip.com/avr-support/atmel-studio-7, accessed: 2018-01-22.