This is the submitted version of a paper published in.

Citation for the original published paper (version of record):

Abbaspour Asadollah, S., Enoiu, E P., Causevic, A., Daniel, S., Hansson, H. [Year unknown!]
A Runtime Verification based Concurrency Bug Detector for FreeRTOS Embedded Software
[Journal name unknown!]

Access to the published version may require subscription.

N.B. When citing this work, cite the original published paper.

Permanent link to this version:
http://urn.kb.se/resolve?urn=urn:nbn:se:mdh:diva-41236
A Runtime Verification based Concurrency Bug Detector for FreeRTOS Embedded Software

SARA ABBASPOUR ASADOLLAH¹, (Student Member, IEEE), EDUARD PAUL ENOIU¹, (Member, IEEE), ADNAN ˇCAUŠEVI ´C¹, (Member, IEEE), DANIEL SUNDMARK¹, (Member, IEEE), HANS HANSSON¹, (Member, IEEE)

¹ Mälardalen University, Västerås, Sweden, (e-mails: sara.abbaspour, eduard.enoiu, adnan.causevic, daniel.sundmark, hans.hansson)@mdh.se

Corresponding author: Sara Abbaspour Asadollah (e-mail: sara.abbaspour@mdh.se).

ABSTRACT When developing embedded software, detecting bugs as early as possible is important. Concurrency bugs is a particularly problematic class of bugs. Several methods have been proposed to detect such bugs, but few of these methods have been implemented in tools and even fewer have been evaluated systematically using realistic software logs. In this paper we present a novel method and tool called DeCoB, which uses runtime verification to detect concurrency bugs in embedded software. DeCoB is tailored for the open source real-time operating system FreeRTOS, and detects and diagnoses concurrency bugs, such as deadlock, starvation, and suspension-based-locking, by analysing runtime traces provided by the Tracealyzer tool, i.e., without debugging and tracing the source code.

This paper presents the implementation of the tool in detail, as well as its functional architecture, together with illustrations of its use in practice. The DeCoB tool can be used during program testing for identifying concurrency bugs using information about the software executions. We experimentally evaluate the DeCoB tool using realistic FreeRTOS test scenarios and 21726 automatically generated logs using our own generator based on the UPPAAL model checker. Our results suggest that the DeCoB tool is effective at detecting whether a diverse set of logs contains concurrency bugs.

INDEX TERMS Bug detector, Concurrency bugs, Embedded software, FreeRTOS, Runtime verification tool.

I. INTRODUCTION

Concurrent embedded systems are prone to bugs, and bugs related to the concurrent execution of such programs are especially challenging to detect and analyze. As embedded software grows increasingly popular [1], developing effective and efficient practical approaches for detecting concurrency bugs in an off-line manner based on logs, as well as at runtime, is vital.

According to d’Amorim and Havelund [2], runtime verification checks traces of programs against properties formulated in a property language, e.g. a logic. On violation of one of these properties, the program is expected to act in order to handle the situation. However, a limitation is that runtime verification only considers states that have actually been reached at runtime. Still, it is an effective approach in testing, as well as in monitoring. For instance, as pointed out by Artho et al. [3], bugs can be automatically revealed by runtime verification based on test cases tailored find the specific bugs. Also, related to monitoring [4], runtime verification can under some circumstances be used to control the program execution away from executing the faulty code (the bug) upon detection of a violation of the corresponding property.

Since concurrency bugs manifest themselves at runtime, and since the complexity of Embedded Systems is increasing, detecting possible concurrency bugs early during software development becomes more difficult. Increasing the trust in the correctness of the software is traditionally achieved by techniques such as testing, model checking and theorem proving, frequently based on requirements formulated in formal or semi-formal languages, although such techniques do not necessarily cover all types of bugs. In order to address these problems, a runtime verification and reflection technique [5] has been previously developed. This technique operates at runtime, which makes it possible for engineers to properly react whenever a software behaves incorrectly.
Since detecting and monitoring faults at runtime is made feasible, this kind of techniques could be suitable for concurrent and parallel software with unexpected behavior or nondeterministic output.

In the case of concurrency bugs [6], introduced by concurrent programming, we are faced with a new set of bugs that typically appear in specific situations. Their appearance are in addition often nondeterministic, for instance as the result of unpredictable thread-interleavings between accesses to shared memory. The result could be dramatic, as the bug can ripple through the software and possibly make it crash, hang or produce erroneous outputs. Concurrency bugs are often categorized as problematic [7], [8], due the difficulty in reproducing them.

In this paper, we show how a tool can be used for monitoring embedded software and for detection of concurrency bugs. We have chosen FreeRTOS as the target environment for our tool since it is a widely used open source operating system that offers support for different hardware architectures in the embedded system domain. This paper is an extension of our previous published paper [9] in which we proposed a runtime verification method for detecting concurrency bugs in embedded software. In particular, we extend the previously published paper with an experimental evaluation of the DeCoB tool using 21,726 automatically generated logs using our own automated generator based on the UPFAAL model checker. In addition, we provide an update on the deadlock detection algorithm.

In the evaluation of this tool we use, in addition to a few realistic software examples, an abstract model of a FreeRTOS embedded software implemented as a network of timed automata fed into the UPFAAL model-checker [10] for log generation. UPFAAL checks that a reachability property describing a concurrency bug goal is satisfied and generates a corresponding log. The main goal of using UPFAAL is to automatically generate a large set of diverse and realistic traces based on timed automata models using UPFAAL’s model checking and simulation support. This allowed us to investigate and evaluate our tool implementation.

A. PAPER CONTRIBUTIONS
In this paper we describe and evaluate a runtime-verification based method for detecting concurrency bugs for embedded software and its tool support. The main contributions of the paper are:

1) A runtime verification method and a tool architecture for detecting concurrency bugs in embedded software.
2) A tool named DeCoB (Detecting Concurrency Bugs) based on the proposed architecture, covering deadlock, starvation and suspension-based-locking bugs for software running under FreeRTOS.
3) An evaluation of the implemented runtime verification tool for detecting concurrency bugs. The initial evaluation is conducted on FreeRTOS running on a SAM4S Xplained platform. In addition, we have used the UPFAAL model checker and a model of an abstract embedded application to generate a diverse set of logs.

B. PAPER ORGANIZATION
The rest of the paper is organized as follows. Background information about the FreeRTOS and Tracealyzer, UPFAAL model checker as well as the basic terminology is presented in Section II. Section III describes the proposed tool workflow and architecture as well as the proposed algorithms. The overview of the experimental evaluation is presented in Section IV. The results of a proof-of-concept evaluation using manually handcrafted FreeRTOS examples is illustrated in Section V. In addition, we describe the results of the evaluation of the tool based on automatically generated logs using the UPFAAL model checker in Section VI. In Section VII we discuss the results of our two-fold evaluations. The related work is given in Section VIII. Finally, the conclusion of the paper is presented and the direction of the future work is highlighted in Section IX.

II. PRELIMINARIES
In this section, we present FreeRTOS, Tracealyzer, UPFAAL model-checker and the terminology used in this paper. FreeRTOS’ multitasking environment allows applications to be constructed as a set of independent tasks. It provides the fundamental mechanism to control and react to multiple, discrete real-world events and it creates the appearance of many concurrently executing tasks by interleaved execution of the tasks. We also utilize the Tracealyzer tool as a separate stand-alone application to log the application events. The following subsections provide more details.

A. FREERTOS
FreeRTOS is a modified GPL-licensed open source real-time operating system, developed by Real Time Engineers Ltd. FreeRTOS is available for a wide range of micro-controllers, particularly targeting small embedded systems [11]. The results of a survey performed on professional engineers in 2017 [12] puts FreeRTOS as the first and second choice for the questions “Which operating systems are you currently using?” and “Which operating systems are you considering using in 12 months?”. The FreeRTOS popularity is also visible in the Embedded Market Survey (probably the most established and trusted study in the embedded industry) [13].

FreeRTOS supports a variety of scheduling strategies, including cooperative, preemptive and hybrid scheduling with static and dynamic task priorities. [14].

B. TRACEALYZER
Tracealyzer is a stand-alone application for visualizing and tracing embedded software executions and is developed by Percepio AB since 2004 [15]. Currently, Tracealyzer is designed for FreeRTOS and Linux. Tracealyzer for FreeRTOS is considered for this study which is designed for 32-bit processors. It supports plug-ins and integrations for common development tools such as Atmel Studio 7.
Tracealyzer has two main tracing modes, streaming and snapshot mode. In streaming mode, the data is transferred continuously to the host PC, allowing for very long trace durations. In snapshot mode, the trace data is kept in a target-side RAM buffer until explicitly uploaded.

C. TIMED AUTOMATA AND THE UPPAAL MODEL CHECKER

We use the UPPAAL model-checker to generate traces. UPPAAL uses timed automata as the input modeling language. The UPPAAL verifier language supports the use of reachability properties. We use this verifier to automatically generate suitable logs using UPPAAL and use of UPPAAL’s trace generator for a submitted reachability property. UPPAAL can generate traces using three options for its diagnostic traces: any trace leading to a goal state, the shortest trace containing a minimum number of states and transitions, and the fastest trace in terms of the time delay.

An UPPAAL timed automaton is a finite-state automaton extended with time clocks. Alur and Dill [16] introduced the model and has become a very popular language for modeling real-time systems. Further information can be found in [17].

The UPPAAL algorithms perform reachability analysis to check for properties of the form \( \exists \diamond \beta \). \( \exists \) is the existential quantifier, \( \diamond \) is the temporal operator used for checking if a requirement eventually holds, and \( \beta \) is a formula capturing a particular type of log containing a concurrency bug. The reachability property checks if there exists a path \( \sigma \) in the timed automaton states and transitions such that \( \beta \) eventually holds. The property is used by the model checker to find a path through the model that satisfies the verification property. A model represented as a network of timed automata \( M_0 \parallel \ldots \parallel M_{n-1} \) is a parallel composition of \( n \) timed automata and several synchronization channels (i.e., \( s! \) corresponds to \( s? \)).

D. TERMINOLOGY

In this section we present the terminology followed in this paper with regard to software problems. In the literature, these definitions are not entirely consistent and are used interchangeably and differently (e.g., in terms like errors, bugs, faults, failures). A software bug is a problem that prevents or impedes the correct functionality of the software [18]. We use the term bug in this paper when referring to problems in the observable behavior of the embedded software.

Leucker and Schallhart defined runtime verification [19] 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”. That is to say, runtime verification is a method used for information extraction at runtime and using this information for bug detection and possible reaction to these with regard to certain properties.

Concurrency bugs are software problems which happen on multithreaded programs (e.g., deadlocks and data races). Previously, we presented a concurrency bug taxonomy [20] in which we classified the bugs based on their common characteristics in terms of observable properties (i.e., even if the observable characteristics of a bug are not sufficient for detection, the observable properties can be used to precisely determine the type of bug [21]). In this paper we use the following three taxonomy classes: Deadlock, Starvation and Suspension (also known as blocking suspension or suspension locking). For more information on these type of bugs and their characteristics we refer the reader to the taxonomy proposed by Abbaspour et al. [8], [20].

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” [8]. During deadlock, all involved tasks are in a waiting state.

Starvation is a condition in which a task in a program is delayed because other processes are always given preference [8] while this delay is not accepted by the ones using the program. At least one of the involved tasks remains in the ready queue during a 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 [8]. In this paper we call this type of locking a Suspension bug.

III. DECOB: DETECTING CONCURRENcy BUGS

In this paper we present a runtime verification method to detect concurrency bugs for embedded software and the detailed implementation architecture including the functionality of each module and the detection algorithms for each category of concurrency bugs.

A. DECOB WORKFLOW

DeCoB is based on a runtime verification technique that monitors embedded software in order to detect concurrency bugs. The previous verification methods such as model checking might behave differently and even incorrectly at runtime due to compiler bugs or mismatches between the actual and expected behavior of the execution environment with respect to resource availability, timing issues or memory behavior.

The logical architecture for detecting concurrency bugs in embedded software is shown in Figure 1. It is decomposed into four layers viz., Logging, Monitoring, Concurrency Bugs Diagnosis, and Mitigation. This architecture is based on the architecture given in [22] and [23].

The Logging layer observes the embedded software events and records the data required by the Monitoring layer. The Monitoring detects the presence of bugs in the software without affecting its behavior. It considers fault detection and could consist of a number of monitors that observe the stream of embedded software events provided by the Logging layer. The Concurrency Bugs Diagnosis layer compares the results of the Monitoring layer to concurrency bugs’ properties. This
layer can diagnose the potential concurrency bugs when the extracted data (properties) from the Monitoring layer is as same as the concurrency bugs’ properties discussed in our previous study [20].

The Mitigation layer re-configures the embedded software in order to mitigate (if possible) the different concurrency bugs by applying the results of the Concurrency Bugs Diagnosis layer and re-establishing a determined system behavior.

It is worth mentioning that the proposed tool is based on the explained runtime verification framework and covers the first three layers for detecting the concurrency bugs. However, reconfiguring the software and fixing the bugs is considered for the future work due to the nature of the embedded software.

B. DECOB ARCHITECTURE

The proposed architecture of the tool is comprised of five separate modules, viz., Parser Module, Starvation Bug Diagnosis Module, Deadlock Bug Diagnosis Module, Suspension Bug Diagnosis Module, and Data Visualization Module. An outline of the tool architecture is illustrated in Figure 2.

The Parser Module parses a log file which is already saved by a monitor and uses a defined template compatible with Tracealyzer event log file. The role of the Parser Module is to extract and calculate the relevant data. The following list illustrates each field and Table 1 presents the format of extracted data from the Parser Module using the following fields:

- “TaskName”: This string typed filed saves the name of a task.
- “FromTime”: This string typed filed saves the time when the Status of a task changes.
- “Status”: This enum typed filed keeps one of the Running, Suspended or Ready values and saves the current status of a task. Running indicates the CPU is busy by executing the task. Ready indicates the CPU is busy by executing other task(s). Suspended indicates the task is blocked due to lack of resources.
- “WaitingReason”: This enum typed filed keeps one of the Semaphore, Queue or User Requested values if the Status of a task is Suspended. Semaphore means the task asked for a semaphore and the semaphore is taken by other task. Queue means the task asked for a queue and the queue does not have any space. User Requested means the user wanted to change the Status of a task to Suspended by a command such as Sleep().
- “RequestedForObject”: This string typed field saves the name of objects which keeps the task in Suspended status.
- “TakenObject”: This string typed field saves the name of an object which holds by a task. If a task holds more than one objects, then the names of them save to this field with “;” as a separator.
- “MaxInReady”: This long typed field saves the maximum duration time when a task is spending in Ready status.
- “MaxInReady FromTime”: This string typed field indicates the begin time of “MaxInReady” for a task.
- “MaxInSuspend”: This long typed field saves the maximum duration time when a task is spending in Suspended status.
- The “MaxInSuspended FromTime”: This string typed field shows the begin time of “MaxInSuspend” for a task.

The Parser Module has another feature to distinguish between the user and kernel tasks. Parser Module is able to extract the information from all tasks i.e., kernel tasks and user tasks, if a user selects the “All” as a type of tasks. Parser Module is able to extract the information from user created tasks if the “User Task” is selected by the user as a type of tasks before browsing a log file.

The extracted data from the Parser Module can be analyzed in each bug diagnosis module. Analyzing the data means comparing the extracted data to the corresponding property of concurrency bugs. If there is a match, then a concurrency bug and its type can be detected and reported.
DeCoB tool needs a value (delay tolerance) from the user as an input value to detect the starvation and suspension bugs. This value shows the maximum acceptable time duration for a task to stay in Ready status in starvation-type bug detection. It also shows the maximum acceptable time duration for a task to stay in Suspended status in suspension-type bug detection. It would be possible to set distinct values for each task independently.

Data Visualization Module is designed for graphic presentation. Basically, it consists of functions and procedures to present the outcomes of the Deadlock Bug Diagnosis, Starvation Bug Diagnosis, Suspension Bug Diagnosis and Parser modules.

### C. OVERVIEW OF THE BUG DETECTION ALGORITHMS

Overview of the Deadlock Bug Diagnosis, Starvation Bug Diagnosis and Suspension Bug Diagnosis approaches are shown respectively in Algorithm 1, 2, and 3 as a pseudocode describing their inner working.

The algorithm for deadlock detection (listed in Algorithm 1) requires an input (suspendedTasks) to be provided as a dataset. This input consists of data extracted from the Parser Module for all tasks which are in the Suspended state, and also with WaitingReason Semaphore or Queue. The general data structure is presented in Table 1. This means that “suspendedTasks” is a two-dimensional array which each row containing an available task. The structure of the column is similar to the Table 1 for keeping task data, “deadlockSet” holds the length of the set of tasks that are involved in the deadlock bug. The variables “deadlockSet”, “isDeadlock” and “possibleBug” are global variables that are used by both procedures, DeadlockDetection and findNode. DeadlockDetection procedure returns a dataset (ResultDataset) which contains the information of the tasks causing the deadlock.

The algorithm for starvation bugs detection pseudocode is listed in Algorithm 2. As in the case of Algorithm 1, this algorithm requires dataset as an input (extractedData) and provides dataset as an output (ResultDataset). The Parser Module performs calculation of the maximum time a task can spend in a Ready state, based on the log file (details on how this is done are described in Section III.B). This value is an important factor for the starvation bugs detection algorithm. As stated in the Algorithm 2 (Line 3), the delay tolerance value can be user-defined for each task. Additionally, the algorithm relates the MaxInReady, for each task, with its user-defined delay tolerance value. In case the MaxInReady, of each task, is greater than the user-defined value for that specific task, we can conclude that the task is subject to starvation and thus its data are added to ResultDataset.

The algorithm for detecting Suspension bugs (listed as pseudocode in Algorithm 3) is also using dataset (extractedData) as its input, and dataset (ResultDataset) as its output.
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 = TaskName[i] from extractedData
6: if (selectedTask[MaxInWaiting] ≥ TaskNameSet(UserDelayTolerance) [i])
7: selectedTask add to ResultDataset /* end of for
9: return ResultDataset

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

The value for MaxInWaiting field is calculated in the Parser Module, in order to define the maximum time a specific task is in the Suspended state. Calculating MaxInWaiting field value in this algorithm is a critical step for the detection of the Suspension bugs. In case the MaxInWaiting for a specific task is larger than the user-defined value (i.e., the delay tolerance value), we can claim the task is subject to a suspension bug and its data is added to the ResultDataset.

IV. EVALUATION DESIGN
In this section, we describe the DeCoB tool experimental evaluation process. Figure 3 presents the evaluation design which includes two distinct approaches: a proof-of-concept evaluation using realistic FreeRTOS logs and a systematic evaluation using automatically generated logs using the UPPAAL model checker.

In the Evaluation using FreeRTOS Examples approach (the left part of Figure 3), we developed realistic logs of concurrency bugs with the help of professional embedded software developers (step (1) in Figure 3). Next, we injected these examples into an embedded software. We developed three different examples, which in turn introduced three different types of concurrency bugs (i.e., Deadlock, Starvation, and Suspension) in order to evaluate the DeCoB tool. We then executed the embedded software on an Atmel SAM4S [24] platform. During the embedded software execution, the Tracealyzer tool traced (step (2) in Figure 3) the system level control flow events (e.g., task switches, synchronization calls) of the software. The traces were recorded (step (3) and (4) in Figure 3) into an event log, which was then used as an input for the DeCoB tool. The log file has a predefined format which was provided by engineers working at Percepio AB [15]. The collected data describes the fraction of the execution time spent on each task over some period of time. During the evaluation, the snapshot mode of Tracealyzer (see Section II-B) was used. The DeCoB tool is parsing and analyzing the log example (step (5) in Figure 3) to detect the injected bugs. Finally, a manual verification of the results (step (6) in Figure 3) was used in order to ensure that DeCoB was able to accurately detect all manually crafted bugs.

In the Evaluation using UPPAAL Model Checker approach (the right part of Figure 3), we evaluate the DeCoB tool using a large set of automatically generated log files containing different types of concurrency bugs. In addition, we generated logs not containing such concurrency bugs to evaluate if the DeCoB tool is able to check such cases. In DeCoB, a user is able to identify the properties of detected bugs.

In order to perform the evaluation, we generated a set of trace files (step (a) in Figure 3) by using the UPPAAL model checker and creating an UPPAAL model of a simple system containing tasks, semaphores and a scheduler. The UPPAAL model together with several properties are used to generate six different set of traces viz., Deadlock, Deadlock-Free, Starvation, Starvation-Free, Suspension, and Suspension-Free type of logs. More details on the UPPAAL model and the properties used are given in Section VI. We used the UPPAAL model checker to generate UPPAAL traces (step (b) in Figure 3) for the three types of concurrency bugs (Deadlock, Starvation, and Suspension). Since the UPPAAL trace file(s) are not compatible with the Tracealyzer log file template, we implemented a transformation tool (i.e., a trace generator) in Java in order to translate (step (c) in Figure 3) the UPPAAL traces into log files supported by the Tracealyzer.
file template, thus creating valid input log files (step (d) in Figure 3) for the DeCoB tool. Lastly, the log files are used as an input (step (e) in Figure 3) for the DeCoB tool. The tool is parsing and analyzing (step (f) in Figure 3) the log files to detect the generated bugs. Finally, the tool checks (step (g) in Figure 3) the expected bugs for each log (as given by the UPPAAL model checker) against the number of actual detected bugs (as identified by the DeCoB tool).

V. DECOB EVALUATION USING FREERTOS EXAMPLES

This section presents the results of a proof-of-concept evaluation using manually handcrafted FreeRTOS examples. As described in Section III-B, the DeCoB architecture contains a Parser Module used for parsing and extracting the data required by the other DeCoB modules. Figure 4 presents a screenshot of the DeCoB tool after running the Parser Module.

To run the initial evaluation and to demonstrate the result of analysing and detecting the concurrency bugs, we first developed practical examples for each type of concurrency bug. Next, we injected them into an embedded application. We exemplify each type of concurrency bugs in the form of a test scenario. Section V-A illustrates the evaluation for Deadlock, Section V-B demonstrates the evaluation for Starvation and Section V-C demonstrates the evaluation of a Suspension bug example. Finally, we traced the event logs during software execution for each example. All measurements are performed on the target platform Xplained SAM4S [24].

A. DEADLOCK TEST SCENARIO

The deadlock test scenario is presented in Figure 5 and implemented by using two tasks and two semaphores. We use within Atmel Studio 7 xTaskCreate() for the creation of tasks and vTaskStartScheduler() for starting the scheduler. The deadlock test scenario is using two tasks with equal priority executing a function concurrently. The following block is a piece of code that we add into the main() function to create and execute these two tasks:

```c
xTaskCreate(vTaskFun1, "Task1", 512, , 1, NULL);
xTaskCreate(vTaskFun2, "Task2", 512, , 1, NULL);
vTaskStartScheduler();
```

Binary semaphores are used in these scenarios, by using the vSemaphoreCreateBinary() function. These semaphores are taken and released by using xSemaphoreTake() and xSemaphoreGive() functions. We need to mention here that xSemaphoreTake() is using a timeout parameter to determine the semaphore waiting time. By setting the parameter to MAX_DELAY, the task is blocking endlessly. When using the Tracealyzer tool, we use a library for trace analysis, and set the recording condition to Snapshot. We use the Snapshot mode by setting the following API in a trace recorder library file (trcConfig.h).

```c
#define TRC_CFG_RECORDER_MODE_SNAPSHOT
TRC_RECORDER_MODE_SNAPSHOT
```

Within the Tracealyzer tool we use vTraceEnable(TRC_START) to record the trace and add it to the start of the main() function.

During execution, we inject the example to a embedded software and log the events. The resulting log is the input to parse and analyze by the DeCoB tool. The outcome of using the DeCoB tool for log analysis is depicted in Figure 6. In this case, Task1 and Task2 are causing a Deadlock bug.

B. STARVATION TEST SCENARIO

A simple starvation example is implemented by calling three tasks. The tasks have access to two shared variables. The following API information defines the shared variables in Atmel Studio in which gCounter is defined for keeping the value of a variable (with the initial value of 100) and cTextGlobalVariable keeps the task’s name (initially empty):

```c
int gCounter = 100;
char cTextGlobalVariable[50] = "";
```

The example with injected Starvation bug is constructed with three tasks (TaskA, TaskB, and TaskC) and two priorities. The next block of code is added to the main() function such that it is possible to create and schedule these three tasks. Priority for TaskA and TaskB is set to (2) while TaskC has a priority (1). The FreeRTOS scheduler guarantees that the task positioned into the Running state is always the task of the highest priority. Tasks with the same priority and in the Ready state will share the provided processing time with the help of a time sliced round robin scheduling scheme.

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

The code segments listed in Figure 7 highlight that TaskA is incrementing the gCounter by one, allocates “Task A” to the cTextGlobalVariable and afterwords emits a request (via taskYIELD()) to perform context switching to another task. TaskB decrement by one the gCounter variable, allocates “Task B” to the cTextGlobalVariable and then emits a request for context switching to another task. Finally, TaskC states the current values of cTextGlobalVariable and gCounter on the terminal screen and emits a request for context switching to the next task.

If we assume to obtain the output of the example within 200,000 microseconds (0.2 seconds), we consider a concurrency bug as the case when this assumption is false. To determine if we have an occurrence of a bug, the first step is to execute the embedded software and, with the help of Tracealyzer, save the event log file while the software is being executed. Afterwards, we use the provided log file as an input to the DeCoB tool for parsing and analysing. Figure 8 shows how the output of the DeCoB tool is presented to the user, emphasizing the maximum time that TaskC spends in Ready state. Since the value is longer than the user-defined tolerance value (i.e., 0.2 seconds), this indicates the presence of a starvation bug.
Figure 4: A screenshot of DeCoB’s output after parsing an event log file

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

Figure 6: A screenshot of DeCoB’s output for detecting a Deadlock bug.

If we assume the assignment of different user tolerance delays to different tasks we can also check if any task is causing a starvation bug. We execute the embedded software and save the event log using Tracealyzer during the execution of the application. We parse the event log and analyse the obtained log file using DeCoB. Finally, as shown in Figure 7 we assign a different delay tolerance for each task and use the Analyze function for detecting Starvation bugs based on the user delay tolerance values. Figure 8 displays the case in which TaskC and TaskA stayed in the Ready state longer than the assumed delays. In this way one can identify two
C. SUSPENSION TEST SCENARIO

In this section, we provide an example of a simple suspension bug, and the corresponding output of the DeCoB tool after tracing the execution of the bug in the embedded software, and analysing the log. The example (shown in Figure 10) contains three tasks of the same priority.

In the execution, We expect to obtain the result of TaskM followed by the result of TaskN. Figure 11 presents a snapshot of the output of this example in which TaskM and TaskN did not execute in the expected order. This case shows that a data corruption has happened due the result of TaskN showing in two consecutive printouts.

In order to detect this bug with DeCoB, we execute the embedded software and save the event log file using Tracealyzer during execution. Next, We parse and analyse the log file using DeCoB. The output of the tool is provided in Figure 12. The delay tolerance for each task is set to 200 microseconds. In other words, if a task stays longer than 200 microseconds in a suspended state, a suspension bug should be detected. Figure 12 shows that TaskM and TaskN stayed in a suspension state longer than 200 microseconds while TaskM waited longer in comparison to TaskN. As a result, TaskN could be causing a bug. DeCoB is able to detect the cause of the Suspension bug by setting different delay tolerance values for each task.

VI. DECOB EVALUATION USING THE UPPAAL MODEL CHECKER

The objective of the DeCoB tool is to be able to automatically detect concurrency bugs (currently Deadlock, Starvation and Suspension bugs) from execution traces. Further, this needs the cause of the Suspension bug by setting different delay tolerance values for each task.

checkBugs(bugType,c)

A. EVALUATION PREPARATION

The model composition is a network of tasks, semaphores and a scheduler closed under a specific behavior. The number of semaphores and tasks used during log generation (i.e., T and S stand for the number of desired tasks and semaphores in the system model respectively) is automatically defined and modified directly in the template declaration. This pre-determined values of T and S directly dictate how the model executes and how the UPPAAL properties are verified. A task executes, once it initiates from the initial location Ready, and once it is scheduled (i.e., schedule!) can go to any of the following locations: Running, StillRunning, Check and Waiting. A task is in Running location when it has control of the CPU and executes. In Figure 13(a) an integer variable cl is modeling a counter of how many time units the task has executed. In addition, the channels free! and schedule! are introduced for synchronization with the semaphore and scheduler timed automata. A task is identified by its id and each task is an instantiation of the task model giving a value to the id. A task releases itself and it communicates with the scheduler through synchronization channels. In addition, the semaphore model shown in Figure 13(c) relies on the use of (binary) semaphore synchronizations with a semaphore being allocated for a task. Each task must wait for the other task to release the semaphore before it can start its execution and take the semaphore if needed. The model contains two locations, Free and Taken corresponding to the two states of the semaphore. It can be used to ensure that only one task at a time takes a certain semaphore. A semaphore S is modeled as an automaton that interacts with a task via two types of synchronization actions: semTake! and semRelease!. In addition, the model contains two functions keepvalues() (i.e., function used for updating the values of the clock variable cl) and checkBugs(bugType,c) (i.e., function used for identifying states in which a task is delayed for some time c and a suspension or starvation bug occurs).

The scheduler is modeled as shown in Figure 13(b). The model contains two locations, Idle and TaskRunning. Initially, when the task is not ready, the scheduler timed automaton is in the Idle location. The scheduler goes to the TaskRunning location once it selects a certain task to run, i.e., scheduler is in the TaskRunning location after the communication with the task over synchronization channel free!. When the current task running finishes its execution, the scheduler is aware of this through the schedule! channel and then returns to the Idle location.

We use model checking and reachability analysis on our network of timed automata for this purpose. UPPAAL uses TCTL language for specifying properties to verify. We generate logs by showing that a certain state corresponding to a specific log type (i.e., Deadlock, Deadlock-Free, Starvation, Starvation-Free, Suspension, Suspension-Free) is reachable. Since UPPAAL is a tool tailored to model checking, it is not directly adaptable to log generation for the DeCoB tool. We demonstrate how we adapt UPPAAL to automatically generating traces for different log types and how we transform these abstract traces to actual logs. The basic approach to generating logs containing a certain bug using model-checking is to use a finite executing path given by the model checker as a log. By characterizing a certain bug as a temporal logic property, UPPAAL model checker can generate an execution path for a reachability property obligation. For example, as shown in Table 2 a log containing a Deadlock bug is expressed as an UPPAAL property by checking if there is a task n that cannot execute because it needs to obtain a semaphore s which is used by another task n-1 and all involved tasks are in a waiting state. In this property we used the exits and forall quantifiers for evaluating if the logical expression is true for all values of the tasks and semaphores. For the property used for generating a deadlock-free log we need to verify that the system never reaches a state where the
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
 ", cTextGlobalVariable, gCounter);
 xSemaphoreGive ( xBinarySemaphore1);
 taskYIELD ();
}
}

deadlock bug happens. For our model, such an expression looks like the negation of the formula given for the Deadlock log type.

The basis of our log generation method is the ability of UP*PAAL to generate witness traces for reachability properties. To achieve this, we developed our own tool implementation that takes as input the abstract formal model represented in UP*PAAL (i.e., depicted in Figure 13) and a bug type property, formalized as a TCTL reachability property. In TCTL, reachability is encoded as: $E <> p$, and the tool checks if a given state formula $p$ (e.g., such as the one in the following starvation bug type property: $E <\exists n : tasks. (Task(n).maxdistanceReady > = UserDelayTolerance)$) may be eventually satisfied. For each reachability property that is satisfied by the model, the UP*PAAL model checker generates a witness trace, representing our abstract log with respect to the property. All traces are collected into an abstract log, and are provided to a translation tool that outputs a log file that can be opened by the DeCoB tool.

A trace generated by the UP*PAAL model checker for a given reachability property represent the executed set of states and transitions. An example of a UP*PAAL generated trace is shown in Figure 14. The trace represents the sequence of states and transitions taken in the network of timed automata. Since this simulation trance is not compatible with the log file format which DeCoB tool requires as input for detecting concurrency bugs, we implemented a tool implementation (i.e., Trace Generator) to convert the UP*PAAL traces to the required log file template. A simple example of a generated log as transformed by the translation tool is given in Figure 15. This example log is transformed based on the UP*PAAL trace given in Figure 14. The tool implementation of the Trace Generator is available online [26].

Finally, we used a set of generated logs for each type of bug in order to systematically evaluate if the DeCoB can identify the presence or absence of bugs. We considered six different set of log files (Set in Table 4). Each set of bugs is performed as a three-variable mixed design with two within-subject variables: the number of tasks ($\#task$), the number
of semaphores (# semaphore), and the type of bug. The two between-subject variables are: the number of created bugs by UPPAAL and the number of detected bugs by DeCoB.

### B. EVALUATION RESULTS

Our strategy for selecting #task and #semaphore is based on three levels which depend on the demand and availability of resources: Easy level (#task < #semaphore), Moderate level (#task = #semaphore) and Hard level (#task > #semaphore). For the Easy level, we assigned a random value between 2 and 5 to #task and a random value between 5 and 10 to #semaphore. For the Moderate level, we assigned a random value between 5 and 10 to both #task and #semaphore. For the Hard level, we assigned a random value between 5 and 10 to #task and a random value between 2 and 5 to #semaphore.

Sets (1) and (2) of generated logs are used to investigate the detection of Deadlock bugs. Sets (3) and (4) are used for evaluating Starvation bugs and Sets (5) and (6) are used for evaluating Suspension bugs.

Overall, in this evaluation we automatically generated 21726 logs based on our strategies for creating six different set of logs. The number of log files for each set is 3621. Each set of logs is generated using various random values for the number of tasks and semaphores. For instance, the log files for our first set (i.e., Deadlock (1)) is decomposed into three strategies: 1704 created log files for the Easy strategy with 2 to 5 tasks and 5 to 10 semaphores, 213 created log files for the Moderate strategy with 5 to 7 tasks and 5 to 7 semaphores, and 1704 created log files for the Hard strategy with 5 to 10 tasks and 2 to 5 semaphores. Half of the total number of log
files (i.e., 10863) are examples of logs containing the three types of concurrency bugs, with the other half (i.e., 10563 log files) being logs that do not contain concurrency bugs. Using this number of logs we evaluated the DeCoB tool in terms of detecting the three types of concurrency bugs. We separately ran each set to detect the relevant concurrency bugs. Since the manual execution of 21726 examples is not feasible, we implemented a feature for the DeCoB tool that supported the running of a set of logs as a batch. In this way a user can set the location of the examples and then the DeCoB is parsing and analysing (by using the Parser and the Bug Diagnosis Modules) each log file. The list of not detected bugs is shown in "Not found bugs" list. During the evaluation, we found some logs listed in "Not found bugs" list and we manually checked the logs to find the cause of the problem.

The results of the evaluation are presented in Table 4. All generated examples (log files) for each set are available online [26]. In summary, the results of the evaluation shows that there were no false negative or false positive detections of concurrency bugs. It should be noted that during the evaluation of DeCoB, a small number of bugs was found and corrected in the tool. The details of this process is explained in the SectionVII.

Table 3: Generated sets of logs according to the bug type and strategy used.

<table>
<thead>
<tr>
<th>Set</th>
<th>Bug type</th>
<th>Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>Deadlock</td>
<td>Easy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Moderate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hard</td>
</tr>
<tr>
<td>(2)</td>
<td>Deadlock-Free</td>
<td>Easy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Moderate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hard</td>
</tr>
<tr>
<td>(3)</td>
<td>Starvation</td>
<td>Easy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Moderate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hard</td>
</tr>
<tr>
<td>(4)</td>
<td>Starvation-Free</td>
<td>Easy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Moderate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hard</td>
</tr>
<tr>
<td>(5)</td>
<td>Suspension</td>
<td>Easy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Moderate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hard</td>
</tr>
<tr>
<td>(6)</td>
<td>Suspension-Free</td>
<td>Easy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Moderate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hard</td>
</tr>
</tbody>
</table>

Table 4: Data collection from UPPAAL model and the DeCoB tool.

<table>
<thead>
<tr>
<th>Set</th>
<th>#Task</th>
<th>#Semaphore</th>
<th>#Created bugs by the Uppaal model</th>
<th>#Detected bugs by tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deadlock (1)</td>
<td>2-5</td>
<td>5-10</td>
<td>1704</td>
<td>1704</td>
</tr>
<tr>
<td></td>
<td>5-7</td>
<td>5-7</td>
<td>213</td>
<td>213</td>
</tr>
<tr>
<td></td>
<td>5-10</td>
<td>2-5</td>
<td>1704</td>
<td>1704</td>
</tr>
<tr>
<td>Deadlock-Free (2)</td>
<td>2-5</td>
<td>5-10</td>
<td>1704</td>
<td>1704</td>
</tr>
<tr>
<td></td>
<td>5-7</td>
<td>5-7</td>
<td>213</td>
<td>213</td>
</tr>
<tr>
<td></td>
<td>5-10</td>
<td>2-5</td>
<td>1704</td>
<td>1704</td>
</tr>
<tr>
<td>Starvation (3)</td>
<td>2-5</td>
<td>5-10</td>
<td>1704</td>
<td>1704</td>
</tr>
<tr>
<td></td>
<td>5-7</td>
<td>5-7</td>
<td>213</td>
<td>213</td>
</tr>
<tr>
<td></td>
<td>5-10</td>
<td>2-5</td>
<td>1704</td>
<td>1704</td>
</tr>
<tr>
<td>Starvation-Free (4)</td>
<td>2-5</td>
<td>5-10</td>
<td>1704</td>
<td>1704</td>
</tr>
<tr>
<td></td>
<td>5-7</td>
<td>5-7</td>
<td>213</td>
<td>213</td>
</tr>
<tr>
<td></td>
<td>5-10</td>
<td>2-5</td>
<td>1704</td>
<td>1704</td>
</tr>
<tr>
<td>Suspension (5)</td>
<td>2-5</td>
<td>5-10</td>
<td>1704</td>
<td>1704</td>
</tr>
<tr>
<td></td>
<td>5-7</td>
<td>5-7</td>
<td>213</td>
<td>213</td>
</tr>
<tr>
<td></td>
<td>5-10</td>
<td>2-5</td>
<td>1704</td>
<td>1704</td>
</tr>
<tr>
<td>Suspension-Free (6)</td>
<td>2-5</td>
<td>5-10</td>
<td>1704</td>
<td>1704</td>
</tr>
<tr>
<td></td>
<td>5-7</td>
<td>5-7</td>
<td>213</td>
<td>213</td>
</tr>
<tr>
<td></td>
<td>5-10</td>
<td>2-5</td>
<td>1704</td>
<td>1704</td>
</tr>
</tbody>
</table>
VII. DISCUSSION

Our two-fold evaluation shows that DeCoB is able to successfully detect bugs in the examined logs. In total, DeCoB is able to correctly detect whether the 21726 automatically generated logs contain concurrency bugs. Nevertheless, we have to mention that DeCoB supports for now the detection of deadlock, starvation and suspension type of bugs. The proposed method behind the DeCoB tool has the potential to be expanded for detection of other types of concurrency bugs during runtime (e.g., Datarace, Atomicity violation and Order violation).

The DeCoB tool covers only these specific types of concurrency bugs in the current implementation of the tool, since we wanted to propose a method which is directly targeting the detection of bugs that have not received much attention lately in research. According to our previous investigation [21], debugging suspension bugs, when compared to debugging other types of concurrency bugs, has not attracted too much attention and the current body of knowledge lacks any studies on suspension bugs between 2005 and 2014. During the same time span, the number of published papers with a focus on debugging starvation bugs was 63 times smaller compared to data race bugs. However, an investigation of open source software [8] indicates that, in addition to the data race type of bugs, suspension bugs occur more often than the other types of concurrency bugs, with 15% of the overall number of bugs found being of suspension type.

As part of our study (and shown in Section V), we experimentally evaluated the DeCoB tool by injecting three realistic examples into an embedded application not containing any known bugs. We observed that the tool is able to detect all seeded bugs and is working as expected. Once we conducted the second phase of the evaluation, in which the DeCoB tool dealt with significantly more diverse and complex examples that contained multiple interleavings of different number of tasks and semaphores, we observed that the DeCoB tool exhibited some problems. Interestingly enough, we were able to detect and fix several bugs in the interface implementation of the DeCoB tool. These bugs were not related to flaws in the proposed method behind the tool. In Figure 16(a) we show a snapshot of the DeCoB interface in which the tool does not consider more than one object for analyzing deadlock-free logs. In Table I we describe how multiple objects are taken by a thread, and how the name of the objects is saved to this field and separated by “;”.

We evaluated the tool using 21726 log files. During our evaluation of traces generated by the UPPAAL model checker. We evaluated the tool using 21726 log files. During our evaluation we found a case in which traces which UPPAAL could not completely generate due to the verification process not finishing in time. This caused the trace generator to create an uncompleted trace and file. Thus, DeCoB faced some log files which were not saved in the right format. In order to fix this error, we regenerated the files using UPPAAL and the Trace generator application. After executing the same experiment with the newly created log files, DeCoB was able to find all the bugs. The latest version of the DeCoB tool is available [26] for experimentation and replication alongside the UPPAAL-based trace generator and the data used in this study.

Figure 16(b) shows the DeCoB interface after fixing this error and executing sets (1) and (4) in order to detect any bugs in logs containing deadlock bugs and deadlock-free logs. Another error was detected in the DeCoB tool during the evaluation of traces generated by the UPPAAL model checker. We evaluated the tool using 21726 log files. During our evaluation we found a case in which traces which UPPAAL could not completely generate due to the verification process not finishing in time. This caused the trace generator to create an uncompleted trace and file. Thus, DeCoB faced some log files which were not saved in the right format. In order to fix this error, we regenerated the files using UPPAAL and the Trace generator application. After executing the same experiment with the newly created log files, DeCoB was able to find all the bugs. The latest version of the DeCoB tool is available [26] for experimentation and replication alongside the UPPAAL-based trace generator and the data used in this study.

VIII. RELATED WORK

Although many frameworks have been proposed for runtime-monitoring, just a few runtime verification tools are available for use. The frameworks are typically proposed for specification-based runtime monitoring. We can classify these approaches in four main categories: rule-based approaches [27], [28], automaton based approaches [2], [29]–[31], temporal logic-based approaches [32]–[36], and regular expression and grammar-based approaches [37].

Rule-based approaches:

As an example of a rule-based approaches to run-time verification, Havelund and Rosu [38], [39] have proposed PathExplorer (JPAX), a tool that targets monitoring executions of sequential and concurrent programs written in Java. JPAX has been applied to a core of the planetary Rover K9 [38] developed by NASA. The tool operates as follows: events are extracted from the executing program and then analyzed by an observer process at a remote location. The Java byte code is instrumented With code that implements the communication with the remote observer, which performs analysis both by error pattern analysis and logic-based monitoring. The latter being analysis based on a set of user-defined application specific rules expressed in a logic. The error pattern analysis uses standard algorithms that for instance detect potential concurrency bugs, such as race conditions and deadlocks, by analysing the execution traces of the program.

Automaton-based approaches:

Another tool for on-line monitoring that targets large distributed parallel programs is Falcon [40], which includes a monitoring subsystem covering two levels: view specifications at a higher level and sensor specifications at the lower level. At the lower level the behaviour of programs

102: checkingDataset select the tasks with waitingForObject = checkingTask[takenObj] from suspendedTasks

had to be changed to:

102: checkingDataset select the tasks with (waitingForObject Like % checkingTask[takenObj]) AND (takenObject
are captured by application specific instrumentations (sensors), while users are supported to implement on-line display systems that provide user-friendly graphical presentations of various program related information, such as data structures, program execution, and performance metrics. The Falcon implementation builds on the C threads library and is available on a number of hardware platforms.

Monitoring-based approaches:
Among monitoring-based approaches targeting concurrent programs, Jass (Java with Assertions) [41] is a tool that expands program annotations added to Java code to pure Java. The code that is generated in this way, dynamically checks the validity of the original annotations/assertions at run-time. Assertions are one of several types, including class invariants, loop invariants, and post and pre-conditions. Jass can 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 can detect it.

Regular expression and grammar-based approaches:
DeadlockFuzzer [42] is implemented for Java to control the thread scheduler and to observe various events by instrumenting the Java bytecode. It combines a dynamic analysis technique with randomized thread scheduler to create real deadlocks with high probability. Joshi et al. implement DeadlockFuzzer in a way which first uses an algorithm to discover potential deadlock cycles in a concurrent program, then executes the program with a random scheduler to create a real deadlock. However, the execution platform of DeadlockFuzzer is not similar to DeCoB, but both of these exercise target programs when a bug occurs and then the tools are able to replay those bugs.

CheckMate [43] is also implemented for Java and it is applied for twelve real-world multi-threaded Java programs. Unlike the tools based on classic deadlock detection techniques, CheckMate detects deadlocks, without relying on any specific synchronization paradigm. This is achieved by extracting traces from program executions. Based on the traces, interleavings of the concurrently executing programs are analyzed to reveal execution patterns that could lead to deadlocks.

EnforceMOP [44] is based on an enforcement mechanism that exploits user-specified properties to generate local monitors and can influence the executions. It is an extension of JavaMOP [45] for monitoring multi-threaded programs in Java. The monitoring generation includes the decomposition of the property into local decentralized monitors for each of the threads. EnforceMOP can check the violation of temporal logic properties and properties such as mutual exclusivity by blocking the threads whose immediate next action violates
these properties. This tool is also used to tolerate concurrency bugs that are caused by unexpected interleavings. However, there are some situations in which EnforceMop may lead to artificial deadlocks while there is no guarantee to find the correct interleaving. One of these situations is explained in the following study [40].

As briefly surveyed in this section, in the literature, few runtime verification tools for concurrency bug detection have been proposed (i.e., most of them focusing on Java programs). The body of knowledge in runtime verification tool for embedded software for detection of concurrency bugs is limited, in particular in regards to tool support, empirical evidence for its applicability, usefulness, and evaluation in practice. As a consequence, the evidence regarding the implementation, applicability, and evaluation of runtime verification tools for concurrency bug detection for embedded software in general, and for FreeRTOS in particular, is limited.

In one related attempt, JPAX is used as a runtime verification tool for monitoring and detecting potential concurrency errors in Java programs. The JPAX tool lacks, as far as we can determine, support for detecting these concurrency bugs for embedded software running under FreeRTOS. In addition, JPAX is able to detect the potential concurrency errors including deadlock and data race bugs and cannot detect other types of concurrency bugs such as Starvation and Suspension bugs. Another tool named Jass implements a monitoring approach considering Java applications. In contrast to the DeCoB tool, as far as we understand, this tool is not able to detect if the interferences are protected by synchronization methods [91]. Finally, the Falcon tool’s implementation relies on specific C libraries. To the best of our knowledge, our DeCoB tool is the first effort to implement a tool for detecting concurrency bugs in embedded software tailored for the FreeRTOS platform.

IX. CONCLUSION AND FUTURE WORK

Nowadays, embedded systems can be found in many applications, from home heating controls and digital watches to industrial applications such as traffic lights and factory controllers. In this paper, we introduce a runtime verification framework for detecting bugs in embedded software, with a focus on concurrency bugs. The proposed approach provides a systematic way to detect such bugs.

In addition, we propose a tool architecture for the runtime verification framework for detecting three types of concurrency bugs: Deadlock, Starvation and Suspension bugs. Based on data collected at run-time, we are able to detect properties that indicate the presence of a particular concurrency bug. Our tool architecture provide a foundation for the development of tools (in our case the DeCoB tool) capable to both detect and identify concurrency bugs of the above mentioned types during program execution.

In the evaluation of our tool we used an ARM Cortex-M micro-controller as target system on which the analyzed code is executing. The evaluation was performed by injecting bugs of the three considered types in existing software. We then used the Tracealyzer tool to generate traces from real executions of the software that our tool could analyse. As presented in Section V the evaluation was successful, in the sense that our tool was able to correctly detect and identify all injected bugs. In addition, we evaluated the implemented tool by 21726 synthesized generated logs using UPPAAL model checker (i.e., as described in Section V). The tool was able to detect all the concurrency bugs (i.e., 21726 created traces) which shows that the tool is effective at detecting concurrency bugs from a diverse set of realistic logs.

Extending the proposed method and generalizing the use of the DeCoB tool for detecting other types of concurrency bugs based on their distinct properties during execution time is an interesting direction for future work.

ACKNOWLEDGEMENT

The work leading to this paper is supported by the Swedish Research Council (VR, EXACT project). Eduard Enoiu is partially funded from the ECSEL Joint Undertaking under grant agreement No.737494 and Vinnova (MegaM@R2). Adnan Čaušević is partially funded from the ITEA3 and Vinnova (TESTOMAT Project). We would like to thank Percepio AB for giving permission to use their tool (Tracealyzer) and Dr. Johan Kraft and Niclas Lindblom at Percepio for valuable discussions, sharing their experience and knowledge on the matter.

References


AUTHORS’ BIOGRAPHY

SARA ABNASPOUR ASADOLLAH is a Ph.D. student at Mälardalen University, Sweden. Her main research interests include debugging, testing and run time verification of concurrent, parallel and multicore software with a focus on concurrency bugs. She has completed her Master’s Degree in Software Engineering at Faculty of Computer Science and Information, University of Malaya, Malaysia. The field of her research during the Master’s Degree was Web Engineering. She also has work experience in various aspects of industrial environment such as mobile development systems, multimedia technologies and E-Learning application, RFID and smart cards technologies.

EDUARD PAUL ENOIU is a postdoctoral researcher at Mälardalen University in Västerås, Sweden, primarily affiliated with the Software Testing Laboratory and the Formal Modelling and Analysis groups at the department of Networked and Embedded Systems. He earned an Engineer’s degree from Polytechnic University of Bucharest in 2009 and a PhD from Mälardalen University in 2016. His research interests span software engineering and empirical research.
ADNAN ČAUŠEVIĆ is a Senior lecturer at Mälardalen University in Västerås, Sweden, member of the Software Testing Laboratory research groups at the department of Networked and Embedded Systems. He received his engineering degree from University of Sarajevo in 2006 and a PhD from Mälardalen University in 2013. His main research focus is within the field of software test automation, developer’s testing as well as empirical evaluation of test efficiency and effectiveness.

DANIEL SUNDMARK is a Professor at IDT and a co-leader of the Software Testing Laboratory. He received a MSc degree (civilingenjör) in Information Technology from Uppsala University in 2003, a licentiate degree from Mälardalen University in 2004, and a PhD degree from Mälardalen University in 2008. Prof. Sundmark research focuses on engineering of embedded software and systems, particularly focusing on embedded systems architecture and software and system testing.

HANS HANSSON is professor in Real-Time Systems at Mälardalen University since 1997. He is director of Mälardalen Real-Time Research Centre and Scientific Leader at the research institute RISE SICS Västerås. He received an MSc (Engineering Physics), a Licentiate degree (Computer Systems), a BA (Business Administration), and a Doctor of Technology degree (Computer Systems) from Uppsala University (UU), Sweden, in 1981, 1984, 1984 and 1992, respectively. Prof. Hansson’s research interests and contributions include component-based design of safety-critical real-time embedded systems, modeling and analysis of real-time communication, real-time testing and debugging, execution-time analysis, development of automotive control SW, formal modeling of timing and probability, and more recently functional safety and cybersecurity.