A POSIX/RTEMS monitoring tool and a benchmark to detect real-time scheduling anomalies

Blandine Djika+*, Georges Kouamou*, Frank Singhoff+, Alain Plantec+
**University of Yaounde 1, Cameroon*
+Lab-STICC UMR CNRS 6285, University of Brest, France
email: blandine.djika@univ-brest.fr, georges.kouamou@gmail.com, frank.singhoff@univ-brest.fr, alain.plantec@univ-brest.fr

Abstract

This article deals with scheduling anomalies in real-time systems. We present MONANO, a POSIX user-level library allowing applications to dynamically detect a pre-identified set of real-time scheduling anomalies.

The MONANO library is based on the modelling of architecture and runtime constraints. MONANO monitors during the runtime the timing behavior of the application and deduces properties needed to identify scheduling anomalies.

We present also a benchmark to evaluate our approach. The benchmark is composed of several programs implementing the most frequent real-time scheduling anomalies.

Keywords: *Real-Time Scheduling Anomalies, Real-Time System, RTEMS, Cheddar.*

1 Introduction

This article focuses on scheduling anomalies in real-time systems. In real-time systems, tasks may have deadlines to meet. As defined by Luis Almeida in [1], a scheduling anomaly refers to a counter-intuitive phenomenon in which increasing the system resources or relaxing the application constraints can lead to missed deadline. While deadline can be verified at software design time, scheduling anomalies may arise at runtime depending on the dynamic behavior of the application.

[Problem Statement] In a previous work [2], we have proposed a model of scheduling anomalies composed of software architecture and runtime constraints. Architecture constraints, called *static* constraints, can be verified prior to execution. However, even after their verification, scheduling anomalies may occur at execution time and deadlines can be actually missed. We called *dynamic* constraints the conditions that may raise scheduling anomalies at runtime. To actually detect and properly handle scheduling anomalies, we have to monitor such dynamic constraints.

[Contribution of this article] In this article, we propose MONANO, a user-level monitoring library which can be used by an application to check dynamic constraints. This library is POSIX compliant. We are experimenting it on the RTEMS operating systems.

In case of an arising scheduling anomaly, MONANO signals the anomaly and allows the application to run specific actions to recover the anomaly. Specific actions can be any operation allowing the application to adapt itself as mixed-criticality theory promotes it [3]. We also provide MONANO with a benchmark implementing most of the real-time scheduling anomalies identified in the literature. We are using this benchmark to validate MONANO. The benchmark may also be used in any research activities related to real-time scheduling anomalies.

The remainder of this article presents background about scheduling anomalies in section 2. MONANO and its companion benchmark are introduced in section 3. Related works and the conclusion are finally presented respectively in section 4 and 5.

2 Background

In this section, we first present with an example what a scheduling anomaly is. Then, we introduce the overall approach we previously proposed in [2].

2.1 Scheduling anomalies

As defined in [1], *a scheduling anomaly* refers to a counter-intuitive phenomenon in which increasing the system resources or relaxing the application constraints can make the application unschedulable.

Let us illustrate a scheduling anomaly with an example from [4]. This example is composed of three periodic tasks scheduled with a non-preemptive fixed-priority scheduler. Each task τ_i is defined by a 5-tuples with its release time $r_i(\tau_i)$, its WCET (Worst Case Execution Time) $C_i(\tau_i)$, its deadline $D_i(\tau_i)$, its period $T_i(\tau_i)$ and its fixed priority $\pi_i(\tau_i)$. Following this notation, the tasks of figure 1 have the parameters given in Table 1 :

Figure 1 presents two schedules of such task set: (a) when the tasks are executed during all their WCET, i.e. each task has

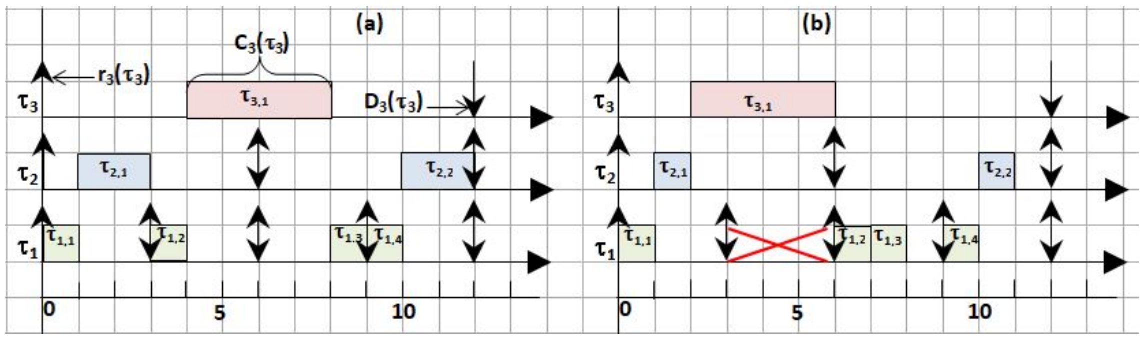

Figure 1: Anomaly when reducing task execution time

Tasks	$r_i(\tau_i)$	$C_i(\tau_i)$	$D_i(\tau_i)$	$T_i(\tau_i)$	$\pi_i(\tau_i)$
τ_1	0	1	3	3	1
τ_2	0	2	6	6	2
τ_3	0	4	12	12	3

Table 1: Task parameters

an execution time equal to its WCET, (b) when the execution times of the task are shortest than their WCET, which is the usual case when running such application. All task deadlines are met in figure 1 (a) while a deadline of τ_1 is missed in figure 1 (b). In figure 1 (b), we can notice that at time 1, τ_2 runs during 1 unit of time while its WCET is about 2 units of time. This real execution time shortest than its WCET, implies that τ_3 executes immediatly after τ_2 at time 2 and cannot be interrupted since the system is a non-preemptive one. Then, when τ_1 is released at time 3, although it is a higher priority task, it has to wait for τ_3 completion at time 6 before starting to work. This finally leads to a missed deadline for τ_1.

Scheduling anomalies were identified and classified by the community in seven types according to how they may occur. Table 2 summarizes them.

Num	Types
1	Reducing the task execution time [5,6,7,8,9,10,11]
2	Changing task priorities [5]
3	Weakening task precedence constraints [5, 6]
4	Increasing processor speed [10]
5	Delaying the execution of the tasks [10]
6	Increasing task period [12, 13, 14]
7	Increasing the number of processors of the execution platform [5]

Table 2: Types of scheduling anomalies

2.2 Modeling scheduling anomalies

Each above scheduling anomalies arises under specific conditions. Such conditions can be modeled as constraints related to the architecture of the real-time system and its behavior at runtime.

In [2], we proposed to model scheduling anomalies according to two types of complementary constraints: *static* constraints and *dynamic* constraints. Static constraints are only related to the architecture design specification. They can be verified prior to execution. We have identified 9 static constraints related to the execution platform and 8 related to the task models. Dynamic constraints are related to particular events that actually raise scheduling anomalies. The verification of such dynamic constraints can only be done at runtime. We have identified the dynamic constraints of each scheduling anomaly described in table 2.

2.3 Proposed anomaly analysis

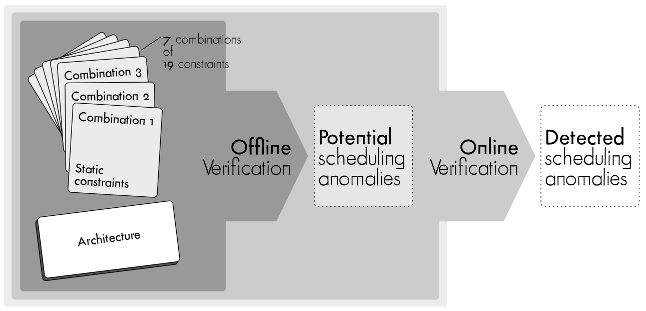

Figure 2: Analysis approach

As shown in figure 2, we propose in [2] a two-step analysis process to detect scheduling anomalies:

1. First, at design time, static constraints are verified with Cheddar [15]. When static constraints hold, it means that scheduling anomalies may occur when dynamic constraints hold at runtime.

2. Second, at runtime, to assess scheduling anomaly, a second analysis is required to check dynamic constraints. In the sequel, we describe MONANO, a monitoring user-level library to verify such dynamic constraints.

3 MONANO monitoring service design and its companion benchmark

This section presents MONANO, a user-level library written in C. Currently, it is implemented on top of RTEMS (Figure 3) but the library is POSIX compliant, then it could be used on many POSIX real-time operating systems. To verify MONANO, we implemented a benchmark based on ROSACE to raise scheduling anomalies identified in the literature. This benchmark is also briefly described in the sequel.

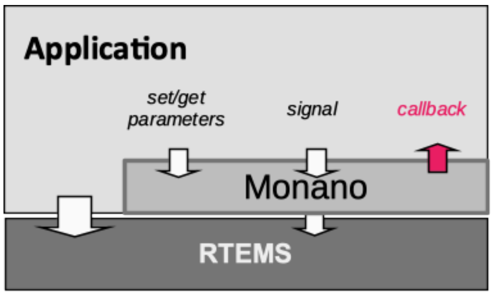

Figure 3: MONANO user-level POSIX library

3.1 In a nutshell

MONANO provides services to create and monitor periodic threads.

MONANO requires application code instrumentation : programmers have to call MONANO services in their programs to monitor the timing behavior of their threads.

MONANO relies on POSIX to create threads, to schedule threads, to implement periodic releases and to monitor threads.

MONANO maintains a static as well as a dynamic view of each monitored thread. MONANO API requires applications to specify the static view of the threads before starting them.

The detection of scheduling anomalies relies on an oracle capable of evaluating the dynamic constraints at runtime. To detect scheduling anomalies during runtime, the oracle checks if dynamic constraints hold for each thread. Dynamic constraints are expressed on runtime events that are either monitored by MONANO or pointed out by the application itself. The monitoring is stopped when the threads are completed.

As described in [2], runtime events that may occur during a thread lifetime and that are part of dynamic constraints leading to scheduling anomalies are priority, precedence or period changes, departure or completion of thread periodic job, or processor speed change.

Regarding scheduling anomaly detection, the interaction between the application and MONANO relies on a specific callback. During its initialization, the application may register a callback that is called by MONANO when an anomaly is detected. The anomaly type together with the involved thread are passed as the callback arguments.

This callback is generated by an oracle integrated into MONANO implementing dynamic constraints leading to scheduling anomalies.

3.2 Application Programming Interface

Figure 4 shows a diagram that depicts the main API elements. The library is composed of two parts: the MONANO manager and views of the application threads. A thread view consists in a *pthread_monano_t* associated with a *pthread_monano_attr_t* struct type instance.

pthread_monano_attr_t stores, for each thread, a set of attributes that are classic static task parameters in real-time scheduling theory. Parameters maintained for each thread are

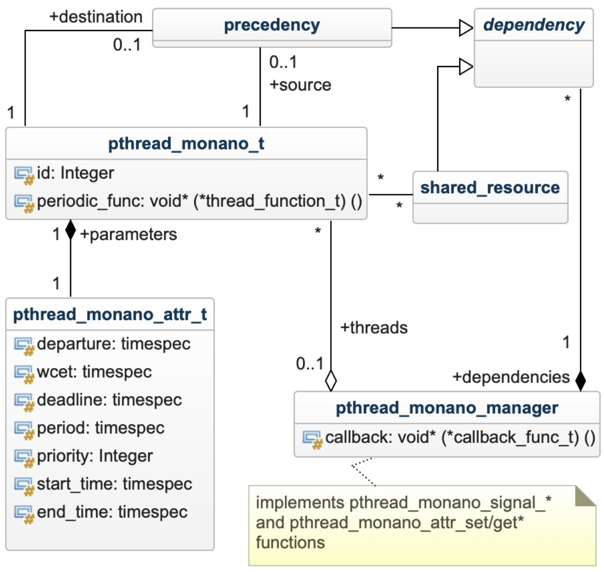

Figure 4: MONANO design

```
#include "monano.h"

struct pthread_monano_t my_monano;
struct pthread_monano_attr_t my_attr;
pthread_monano_id_t id;

void * my_callback(int anomaly_number,
        pthread_monano_id_t tid) {
    printf ("Anomaly %d is detected in thread number
         %d\n", anomaly_number, tid);
    exit (0) ;
}

void * a_periodic_thread(void * arg) {
   pthread_monano_signal_departure_time( & my_monano, id);
   /* Run its periodic program */
   ...
   pthread_monano_signal_end_time( & my_monano, id);
   return NULL;
}

void * POSIX_Init(void * argument) {
   struct timespec period, wcet, ...;
   int priority  =  ...;

   pthread_monano_attr_setperiod( & my_attr, period);
   pthread_monano_attr_setwcet( & my_attr, wcet);
   pthread_monano_attr_setpriority( & my_attr,  priority ) ;
   pthread_monano_register_anomaly_callback( & my_monano,
        my_callback);

   pthread_monano_periodic_thread_create( & my_monano, &
        my_attr, a_periodic_thread, & id, NULL);
   return NULL;
}
```

Figure 5: MONANO program example

namely, the *wcet*, the *deadline*, the *period* which are set as POSIX *timespec structs*. The *priority* is also set as an integer. MONANO also maintains dynamic parameters of each thread such as the last *release time* and the last *completion time* of the thread. Notice that MONANO is only currently supporting periodic threads.

The MONANO manager is in charge of creating threads and of detecting scheduling anomalies. The manager provides 2 functions to respectively create threads and register the callback C function that is called when a scheduling anomaly occurs.

Functions allowing the application to send events to MONANO are named *pthread_monano_signal* in the figure 4.

3.3 MONANO program example

Figure 5 shows a MONANO example program. Lines 24 to 26 setup the static parameters of the MONANO thread to create and line 27 registers the callback that is run when a scheduling anomaly occurs. In this example, the function *my_callback* is called in case of a scheduling anomaly. After such initialization steps, the program creates the MONANO periodic threads and starts to run. If an anomaly occurs, then the callback *my_callback* is called and the RTEMS application is stopped with the *exit* function. Notice that the code of the thread notifies two events to MONANO: when the thread starts to run (with the *pthread_monano_signal_departure_time* function) and when it completes (with the *pthread_monano_signal_end_time* function).

3.4 Benchmark to investigate scheduling anomalies in real-time systems

The first goal of the benchmark is to enforce situations in which scheduling anomalies arise. The second goal is to serve as an extensible example to verify the correctness and the usability of MONANO itself.

In the literature, there is today no benchmark gathering programs that raise the more frequent known scheduling anomalies for real-time systems. In this section, we present the MONANO benchmark that contributes to fulfill such a need.

As explained previously, detecting scheduling anomalies requires verifying if both static and dynamic constraints hold for a given application. The different services implemented in MONANO intend to verify them.

Scheduling anomalies may arise both in uniprocessor and multiprocessor architectures. Several scenarios can lead to a type of scheduling anomaly. We have 19 scenarios for the 7 types of scheduling anomalies identified in both uniprocessor and multiprocessor systems. However in this article, we only focus on uniprocessor systems and then, the benchmark only handles the 5 anomalies occuring in uniprocessor systems. The 5 uniprocessor anomalies occur in 9 scenarios. Each scenario is implemented by a program in the benchmark.

The current MONANO benchmark only implements the dynamic constraints (DC) related to one of the 5 uniprocessor anomalies:

- constraint D1 becomes true when MONANO detects that the thread execution time is reduced (see scheduling anomaly 1 in Table 2).
- constraint D2 becomes true when MONANO detects that a thread has changed its current priority (see scheduling anomaly 2 in Table 2).
- constraint D3 becomes true when MONANO detects that a thread dependency was not met (see scheduling anomaly 3 in Table 2).
- constraint D4 becomes true when MONANO detects that the processor speed has changed (see scheduling anomaly 4 in Table 2).
- and the constraint D5 becomes true when MONANO detects that a thread execution is delayed (see scheduling anomaly 5 in Table 2).

To implement the MONANO benchmark, we adapted ROSACE, an open-source avionic control command software developed by [16] in C. First, ROSACE was adapted to run on RTEMS with the POSIX API. Second, we implemented 9 different versions of ROSACE corresponding to the 9 uniprocessor scheduling anomaly scenarios. Each of these 9 programs is a ROSACE program modified to comply with the static and the dynamic constraints of the related scheduling anomaly.

Table 3 summarizes, for the 9 programs their static constraints and their dynamic constraints that the program implements.

DC	Programs	Static constraints
D1	P1	Threads may have precedence constraints
	P2	Non-preemptive scheduling
	P3	Deadline Monotonic scheduling
		Threads may access shared resources
	P4	EDF scheduling
		Threads are asynchronously released
		Threads may be suspended
D2	P5	Threads are independent
D3	P6	Threads may have precedence constraints
D4	P7	Threads are asynchronously released
		Threads may access shared resources
	P8	Deadline Monotonic scheduling
		Threads are asynchronously released
		Threads may access shared resources
D5	P9	Threads may access shared resources
		Threads may be suspended

Table 3: Static and dynamic constraints of each program of the benchmark

We have implemented the 9 programs on a uniprocessor preemptif fixed priority scheduling RTEMS target. One may notice that some of the programs of Table 3 require a different scheduling policy (e.g. program P4 requires a EDF scheduling). Furthermore, ROSACE is implemented by a set of threads that communicate by flow of data. One may notice also that in Table 3 some of the programs are not compliant with this ROSACE implementation (.e.g program P9 uses shared resources to thread communications).

To experiment all types of scheduling anomalies on the same RTEMS/POSIX operating system and with the same application baseline (ROSACE), we have implemented in the 9 programs specific mechanisms to enforce all static constraints. For example, with programs 4 and 9, scheduling anomalies will occur when threads are suspended while the original ROSACE program does not suspend any thread. To implement suspended threads and to actually raise the corresponding scheduling anomalies, threads of programs 4 and 9 are blocked on a counting POSIX semaphore. Table 4 gives a short description of what we have implemented to make static constraints compliant with ROSACE and RTEMS.

Constraints	**Implementation**
Precedence constraints	Communication implemented with a counting semaphore initialized to 0.
Deadline monotonic scheduling	Priority assignment with $setschedparam$ according to the deadline
Shared resources	Implemented by POSIX mutexes
Non-preemptive scheduling	Non-preemptive scheduling enforced with a mutex shared by all threads
Threads asynchronously released	Release times are delayed with $nanosleep$
Suspended threads	Threads are suspended with $nanosleep$

Table 4: Implementation of the static constraints on POSIX/uniprocessor RTEMS

The static and the dynamic constraints rely on various data that are either given at application startup by the programmer or measured by MONAO during execution or computed at runtime by MONANO. Table 5 gives for each scheduling anomaly data that are either monitored by MONANO or either computed by MONANO. Let consider $S = \{\tau_1, ..., \tau_n\}$, a set of n periodic threads as defined in section 2.1 and monitored by MONANO. Table 5 presents each data continuously updated by MONANO during runtime and if they are computed or measured :

- $start\_time(\tau_i)$ and $end\_time(\tau_i)$ are respectively the start time and the end time of each job of τ_i.
- $execution\_time(\tau_i)$ is the real execution time of a each job of τ_i.
- $blocking\_time(\tau_i)$ is the computed blocking time of τ_i on the shared resources.
- $preemption\_time(\tau_i)$ is the amount of time τ_i is preempted by threads with a higher priority level.
- $suspending\_time(\tau_i)$ is the amount of time τ_i has decided to suspend itself.
- $priority(\tau_i)$ is the current priority of τ_i.
- $dependencies\_list$ stores the real execution order of the threads that are constrained by thread precedencies.
- $processor\_speed$ is the current processor speed.

DC	**Measured data**	**Computed data**
D1	$start\_time(\tau_i)$, $end\_time(\tau_i)$	$execution\_time(\tau_i)$, $blocking\_time(\tau_i)$, $preemption\_time(\tau_i)$
D2	$priority(\tau_i)$	
D3	Threads execution order in $dependencies\_list$	Missed precedency constraints
D4	Current processor speed	
D5	$start\_time(\tau_i)$, $end\_time(\tau_i)$	$suspending\_time(\tau_i)$

Table 5: Metrics monitored on POSIX/uniprocessor RTEMS by MONANO

4 Related work

Previous research on scheduling anomalies has mostly focused on identifying and presenting different types of scheduling anomalies in both uniprocessor and multiprocessor systems [5, 6, 7, 8, 9, 10, 11, 12, 13, 14].

However, detecting anomalies also requires to detect events occuring when scheduling anomalies are raised. Many works have investigated how to monitor events in real-time systems and some of them could be applied to detect scheduling anomalies.

First, monitoring tools devoted to stream-based systems were developed by the community. For example, Copilot [17, 18] monitors systems by regularly capturing values (called samples) of variables of the system. The overall values of a given variable constitute a data stream on which verifications can be applied. As Copilot, RTLola [19, 20] also operates on data streams and monitors them. However, in the case of RTLola, the software components charged to monitor are generated from a specification written in a formal language. StreamLAB [21] is another framework example using RTLola for monitoring purposes.

R2U2(Realizable, Responsive, Unobtrusive Unit) [22, 23] is focusing on the monitoring of security properties for Unmanned Aerial Systems (UAS) built on FPGA. The objective is to detect security attacks.

Hili [24] proposed a model-based approach to monitor real-time systems during their runtime. The approach provides a means to integrate and configure various monitors.

Yibing [25] identifies variations between predicted behavior and monitored behavior. The method is using a digital twin.

Finally, Reinier [26] proposed MuSADET, a tool that looks for timing anomalies in event traces for real-time systems. The framework classifies anomalies with metrics between event arrivals.

As seen above, many authors have carried out technics for monitoring real-time systems during their execution, but without, most of the time, focusing specifically on the detection of scheduling anomalies.

In the contrary, we propose in this article a monitoring tool specifically devoted to detecting scheduling anomalies of a real-time system during runtime.

5 Conclusion

This article focuses on scheduling anomalies in real-time systems. The context in which an anomaly may occur have been studied and presented in [2]. In [2], a model of scheduling anomalies composed of software architecture and runtime constraints has been proposed. From this model, we proposed in this article a monitoring tool called MONANO which is able to detect runtime constraints leading to real-time scheduling anomalies. We are experimenting MONANO in Cheddar [15]. MONANO comes with a benchmark prototyped in RTEMS. We are currently running experiments to evaluate MONANO performance and intrusivity.

To fully evaluate MONANO, the ability to handle *false positive* and *false negative* anomalies is necessary to avoid inappropriate callback invocations. For now, *false positive* and *false negative* anomalies are not taken into account. A *false positive* result would occurs if an analysis gives an invalid positive result (an anomaly is detected whereas it should not be detected), and a *false negative* result would occurs if an analysis gives a invalid negative result (an anomaly is not detected whereas it should be detected). In the current state of our work, we are not able to provide such an analysis. In our future work, we plan to experiment with more use cases to improve dynamic anomaly analysis to include *false positive* and *false negative* detection.

For this article, we focused on uniprocessor systems. We also expect to improve the MONANO tool by integrating the detection of scheduling anomalies in multiprocessor systems.

Finally, another future work would be to investigate how MONANO could be used to monitor other real-time properties.

6 Artefact

MONANO and its companion benchmark are available at `http://beru.univ-brest.fr/svn/CHEDDAR/trunk/src/framework/scheduling_anomalies`

References

[1] L. Almeida, P. Pedreiras, and R. Marau, "Traffic scheduling anomalies in temporal partitions," in *From Model-Driven Design to Resource Management for Distributed Embedded Systems: IFIP TC 10 Working Conference on Distributed and Parallel Embedded Systems (DIPES 2006), October 11–13, 2006, Braga, Portugal*, pp. 95–104, Springer, 2006.

[2] B. Djika, F. Singhoff, A. Plantec, and G. E. Kouamou, "Work-in-progress: Models and tools to detect real-time scheduling anomalies," in *2021 IEEE Real-Time Systems Symposium (RTSS)*, pp. 540–543, IEEE, 2021.

[3] A. Burns and R. Davis, "Mixed criticality systems-a review," *Department of Computer Science, University of York, Tech. Rep*, pp. 1–69, 2013.

[4] G. Phavorin, P. Richard, J. Goossens, T. Chapeaux, and C. Maiza, "Scheduling with preemption delays: anomalies and issues," in *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, pp. 109–118, 2015.

[5] R. Graham, "Bounds on the performance of scheduling algorithms," *Computer and job scheduling theory*, pp. 165–227, 1976.

[6] M. Richard, P. Richard, E. Grolleau, and F. Cottet, "Contraintes de précédences et ordonnancement monoprocesseur," in *Proc. Real Time and Embedded Systems (RTS 2002)*, pp. 121–138, 2002.

[7] S. Pailler, *Analyse hors ligne d'ordonnançabilité d'applications temps réels comportant des tâches conditionnelles et sporadiques*. PhD thesis, Poitiers, 2006.

[8] R. Ha and J. W. Liu, "Validating timing constraints in multiprocessor and distributed real-time systems," in *14th international conference on distributed computing systems*, pp. 162–171, IEEE, 1994.

[9] F. Ridouard, P. Richard, F. Cottet, and K. Traore, "Some results on scheduling tasks with self-suspensions," *Journal of Embedded Computing*, vol. 2, no. 3-4, pp. 301–312, 2006.

[10] G. C. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*, vol. 24. Springer Science & Business Media, 2011.

[11] P. Richard, "On the complexity of scheduling real-time tasks with self-suspensions on one processor," in *15th Euromicro Conference on Real-Time Systems, 2003. Proceedings.*, pp. 187–194, IEEE, 2003.

[12] J. Goossens, "Introduction à l'ordonnancement temps réel multiprocesseur," in *École d'été" Temps réel"*, 2007.

[13] B. Andersson and J. Jonsson, "Preemptive multiprocessor scheduling anomalies," in *Proceedings 16th International Parallel and Distributed Processing Symposium*, pp. 8–pp, IEEE, 2002.

[14] B. Andersson, "Static-priority scheduling on multiprocessors.," 2004.

[15] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, "Cheddar: a flexible real time scheduling framework," in *Proceedings of the 2004 annual ACM SIGAda international conference on Ada: The engineering of correct and reliable software for real-time & distributed systems using Ada and related technologies*, pp. 1–8, 2004.

[16] C. Pagetti, D. Saussié, R. Gratia, E. Noulard, and P. Siron, "The rosace case study: From simulink specification to multi/many-core execution," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 309–318, IEEE, 2014.

[17] I. Perez, F. Dedden, and A. Goodloe, “Copilot 3,” tech. rep., 2020.

[18] L. Pike, A. Goodloe, R. Morisset, and S. Niller, “Copilot: A hard real-time runtime monitor,” in *International Conference on Runtime Verification*, pp. 345–359, Springer, 2010.

[19] P. Faymonville, B. Finkbeiner, M. Schwenger, and H. Torfah, “Real-time stream-based monitoring,” *arXiv preprint arXiv:1711.03829*, 2017.

[20] J. Baumeister, B. Finkbeiner, S. Gumhold, and M. Schledjewski, “Real-time visualization of stream-based monitoring data,” in *International Conference on Runtime Verification*, pp. 325–335, Springer, 2022.

[21] P. Faymonville, B. Finkbeiner, M. Schledjewski, M. Schwenger, M. Stenger, L. Tentrup, and H. Torfah, “Streamlab: stream-based monitoring of cyber-physical systems,” in *International Conference on Computer Aided Verification*, pp. 421–431, Springer, 2019.

[22] J. Schumann, P. Moosbrugger, and K. Y. Rozier, “Runtime analysis with r2u2: a tool exhibition report,” in *Runtime Verification: 16th International Conference, RV 2016, Madrid, Spain, September 23–30, 2016, Proceedings 7*, pp. 504–509, Springer, 2016.

[23] P. Moosbrugger, K. Y. Rozier, and J. Schumann, “R2u2: monitoring and diagnosis of security threats for unmanned aerial systems,” *Formal Methods in System Design*, vol. 51, pp. 31–61, 2017.

[24] N. Hili, M. Bagherzadeh, K. Jahed, and J. Dingel, “A model-based architecture for interactive run-time monitoring,” *Software and Systems Modeling*, vol. 19, pp. 959–981, 2020.

[25] Y. Li, Z. Tao, L. Wang, B. Du, J. Guo, and S. Pang, “Digital twin-based job shop anomaly detection and dynamic scheduling,” *Robotics and Computer-Integrated Manufacturing*, vol. 79, p. 102443, 2023.

[26] R. Torres Labrada, “Multi-signal anomaly detection for real-time embedded systems,” Master’s thesis, University of Waterloo, 2020.