SMT-based Repairing Real-time Task Specifications

Anand Yeolekar
TCS Research, India & TUM, Germany
anand.yeolekar@tcs.com

Ravindra Metta
TCS Research, India & TUM, Germany
r.metta@tcs.com

Samarjit Chakraborty
Dept. of CS, UNC Chapel Hill, USA
samarjit@cs.unc.edu

***Abstract*—When addressing timing issues in real-time systems, approaches for systematic timing debugging and repair have been missing due to (i) Lack of available feedback: most timing analysis techniques, being closed-form analytical techniques, are unable to provide root cause information when a timing property is violated, which is critical for identifying an appropriate repair, and (ii) Pessimism in the analysis: existing schedulability analysis techniques tend to make worst case assumptions in the presence of non-determinism introduced by real-world factors such as release jitter, or sporadic tasks. To address this gap, we propose an SMT encoding of task runs for *exact* debugging of timing violations, and a procedure to iteratively repair a given task specification. We demonstrate the utility of this procedure by repairing example task sets scheduled under global non-preemptive earliest-deadline-first scheduling, a common choice for many safety-critical systems.**

***Index Terms*—task, schedulability, repair, specification, SMT**

I. Introduction

For safety-critical embedded systems, *timing correctness* is as important as functional correctness of the system's software implementation: delayed output of a task can lead to system failure. While automated debugging and repair of software is well established and actively studied [1]–[3] for functional correctness, similar efforts have been missing for timing correctness. Existing techniques to address timing errors do not treat this as a *debug-and-repair* problem, and instead rely on ad-hoc approaches to fix timing violations, *e.g.,* by inserting idle times between tasks. We present one of the first systematic attempts towards automated debugging and repair for timing correctness, in particular repairing the task specification.

As safety-critical systems, from autonomous cars and robots to medical devices, embrace multicore architectures with larger volumes of software code, ensuring timing correctness is an increasing challenge. Real-world embedded systems are typically composed of a mix of periodic (*e.g.,* modeling time-triggered components) and sporadic tasks (*e.g.,* modeling interrupts), with release jitter, and execution times bounded by best- and worst-case values. Tasks are deployed on multicore processors, and the scheduler implementation may decide on resolving ties and processor-to-job mapping. These factors introduce *non-determinism* in the scheduling of task instances, leading to a large number of feasible task runs, and a timing violation can occur along *any* of these runs.

In the event of a timing violation, the system designer might have to *debug* the task specification (or parameters of the task model) by identifying possible causes and take remedial actions. For instance, to avoid deadline miss, one may adjust the offset of one task, or tweak the period of another task. Depending on the constraints for debugging, this can quickly become a non-trivial exercise. Hence, automated timing debugging could be highly useful to the designer, but there has been little work on automatically repairing the task parameters. *In this work, we address this* ***timing debugging*** *problem: detecting a timing violation and automatically repairing or modifying user-specified task parameters.*

In particular, deadline misses are unacceptable in hard real-time systems [4], as they may lead to critical faults. Therefore, debugging depends on the ability to precisely identify the cause and location of the timing violation. Most existing schedulability analysis techniques are *pessimistic* in real-world settings; thus there are limitations in extending such analyses to revise task parameters. With this in mind, a sub-goal of our work is to introduce an *exact* schedulability analysis for a class of task sets capable of modeling realistic systems, that can be *extended* for the main goal of repairing the task specification.

A. Proposed approach

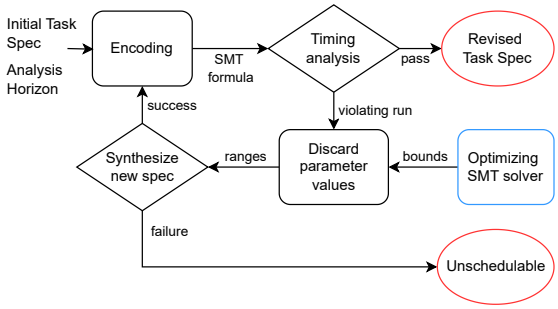

Fig. 1. High-level approach to repair task specification

Figure 1 shows our high-level approach to repairing task specifications. Given the task set and an analysis horizon (*e.g.,* the task hyperperiod), we encode runs of the task set under the given scheduling policy as a set of satisfiability-modulo-theory (SMT) constraints, along with the timing property to be analyzed *i.e.,* task deadlines. On timing violation, the SMT solver reports a *witness*, *i.e.,* an assignment to the variables of the formula describing the task run. This witness provides valuable insights into the circumstances leading to the deadline miss, aka **the debug phase**. Careful analysis of the witness allows us to discover subsets of values of task parameters that will *continue* to yield deadline misses. We then synthesize new values of user-specified task parameters, after discarding these subsets, aka **the repair phase**. This iterative *refinement* of the task set parameter space continues till we either discover parameters that guarantee timing satisfaction, or the parameter space is exhausted, indicating that the task specification cannot be debugged any further.

978-3-9826741-0-0/DATE25/© 2025 EDAA

According to a recent survey [5], 27% of the real-world automotive systems are scheduled under earliest deadline first (EDF) policy, with 80% systems deployed on multicore processors. Around 74% of the systems surveyed have at least two different kinds of task activation, with 30-40% consisting a mix of periodic and sporadic tasks. Further, avionics and automotive safety-critical domains had 79% and 56% hard timing constraints, respectively. Non-preemptive (NP) scheduling has several advantages such as reduced run-time and design overheads, more accurate estimation of task worst-case execution time [6]–[9]. Therefore, while our approach is general and applicable to a wide variety of task models and scheduling policies, for concreteness, we demonstrate our timing debugging and repair approach for the setting with NP-EDF scheduling on multicore processors, with task sets consisting of a mix of periodic and sporadic tasks, for which known analyses are pessimistic. The **main contributions** of this work are:

1) **Timing Debugging:** An *exact* SMT encoding of runs for task sets consisting of a mix of periodic and sporadic tasks, scheduled under multicore global NP-EDF,
2) **Specification Repair:** A refinement procedure for task offsets and periods that discovers and discards intervals of parameter values leading to timing violation, and
3) **State space pruning:** A DAG to capture static scheduling order over spawned task instances, significantly pruning the sequences of jobs explored by the solver.

B. Timing Debugging: A motivating example

Consider an example task set specified in Table I, scheduled under NP-EDF, on a dual core processor, with all offsets set to 0. Our timing analysis reveals a run leading to a deadline miss (deadline equals period), illustrated in Fig. 2. The first instances of tasks T_2 and T_3, released at time 0, are scheduled on processor 0 and 1, and terminate at time 17 and 15, respectively. The first instances of T_0 and T_1 are released at time 1 (release delayed due to jitter). T_0 is eventually scheduled at time 15 (marked in red) on processor 1, missing deadline at time 10.

TABLE I
EXAMPLE TASK SET

Task	Jitter	Period	Exec. time
T_0	1	10	[1,2]
T_1	1	30	[6,8]
T_2	2	60	[15,17]
T_3	2	60	[15,20]

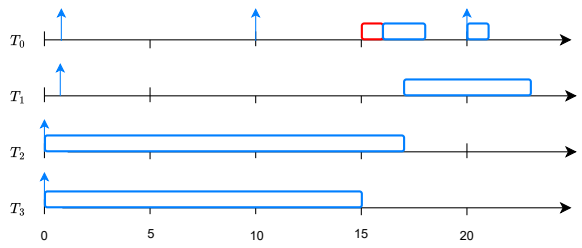

Fig. 2. Task set run leading to deadline miss

Refining offsets. Observe that for task T_0, if we choose any offset value in the interval [0,8], with offsets of other tasks unchanged, the first instance of T_0 *continues* to miss its deadline. For example, if we set offset of T_0 to 8, the deadline of the first instance of T_0 is pushed to 18 time units, but both processor cores can be occupied up to time 17 (T_2 and T_3 execute up to time 17), and with T_0 requiring 2 time units in the worst-case, misses its deadline at 18. Similarly, for task T_1, if we choose any value of offset in the interval [0,29], the same run reappears. We can discover *intervals* of offsets leading to timing violation by analyzing this run. The task set is schedulable if we choose new offsets as 2, 15, 0, 12.

Refining periods. If we assign a new value to the period of T_0 from the interval [10,18], with periods of other tasks unchanged, T_0 *continues* to miss deadline: in this run, T_0 can be made to wait up to time 17 (after T_2 and T_3 terminate), causing T_0 to terminate at time 19 in the worst case, thereby missing deadline. Similarly, selecting period of tasks T_1, T_2, T_3 in the intervals [30,60], [60,120] and [60,120], respectively, admits the same timing violation. The task set is schedulable, if we choose the new periods as 20, 30, 30, 40.

Our approach enables **exploration** of the space of task parameters, revising combinations of parameters in the process to achieve timing safety. This is of practical use, as design constraints might permit revising *specific parameters of specific tasks*, and each revision can have a beneficial or adverse impact on other system requirements such as control performance.

II. Related Work

Existing techniques to address timing errors rely on domain-specific solutions *e.g.,* by adjusting the execution environment of the system, or by adjusting the higher-level control application itself. Further, many existing schedulability analysis techniques either restrict the class of tasksets that can be analyzed, or become pessimistic in real-world settings. As a result, systematic debug-and-repair for timing is missing.

Overcoming timing errors. Some of the earlier attempts to overcome deadline misses adjusted scheduler behaviour, such as relaxing work conservation. For example, Lee [10] use knowledge of future task release patterns to insert idle time such that higher priority tasks are not blocked. However, sporadic tasks or release jitter makes it difficult to predict future task release patterns. Nasri [11], [12] present an idle-time insertion policy under NP EDF and RM scheduling to improve schedulability, by augmenting the scheduler with heuristics to relax work conservation, applicable to loose-harmonic unicore task sets. In contrast, we propose to repair the task specification itself without any changes to the scheduler.

Seto [13], Bini [14], present analytical approaches to sensitivity analysis *i.e.,* for task computation time and period, to improve schedulability under FP or EDF scheduling. However, the approaches are restricted to analysis of periodic unicore tasksets. Roy [15] propose domain-specific modification of periods of control tasks such that the poles of the closed-loop system experience minimal shift, limited to periodic tasks. Nasri [16] determine safe periods for a set of tasks, given their worst-case execution times, under preemptive EDF and RM, limited to unicore tasks. Yeolekar [17] encode task sets as a C program, check for deadline miss using CBMC model checker and revise task offsets, but limited to unicore periodic tasks.

While the technique could be extended to handle multicore task sets including sporadic tasks, scalability will worsen further due to the state space explosion problem.

In other domains addressing repair, Bendik [18] relax clock constraints of timed automata to ensure reachability using an MILP formulation. Noguchi [19] repair inconsistent timing requirements expressed as logical implications with time constraints, using SMT. Cimatti [20] solve the dual problem of computing schedulable regions of parameters. Pozo [21] propose schedule synthesis and repair by reassigning frame offsets in the presence of network link failure using ILP. While similar in spirit, our approach admits non-determinism in system behaviour and presents a wider choice of repair options, as suited for real-world applications.

Schedulability analysis. Analytical techniques developed for checking schedulability *e.g.,* critical instant analysis, or demand-bounding, give pessimistic results [22] in the presence of non-determinism. Gu [23] expanded earlier approaches [24]–[25] to model tasks with release jitter and execution time variation using Uppaal. Pedro [26] check schedulability using SMT solver. These exact approaches work for classes of periodic tasks but do not repair timing violations.

Nasri [27] present a technique to check exact schedulability of NP tasks under unicore fixed-job-priority scheduling that scales to thousands of jobs, extended in [28] towards a sufficiency test that analyzes multicore task sets, for periodic tasks. Yalcinkaya [29] introduce exact schedulability analysis for multicore FP scheduling using Uppaal, for multicore periodic or sporadic tasksets, with release jitter and execution time variation. Burmyakov [30] introduce techniques to prune the state space explored for exact schedulability checking for multicore sporadic tasksets under preemptive global FP.

III. System Model and Encoding

In this section we present an *exact* encoding of the runs of a taskset, to be used as a sub-procedure for the main goal of repair. Let $\mathcal{T}_p$ be a finite set of periodic tasks, to be scheduled on a multicore processor with N identical cores. $\tau \in \mathcal{T}_p$ is specified as $\tau \coloneqq (i, P, B, W, O, J)$, where i is a unique task identifier, P is task period, B, W are the best- and worst-case execution times, O is the task offset, and J is the release jitter. We refer to task instances as jobs. We denote the release time of the j^{th} job spawned by task i as $r_{i,j}$. Due to jitter, the instant of release lies in the interval $r_{i,j} \in [jP_i + O_i, jP_i + O_i + J_i]$. We assume deadlines are *implicit*, *i.e.,* job deadline equals the period, and computed as $d_{i,j} = (j+1)P_i + O_i$. Let $s_{i,j}$, $e_{i,j}$ and $p_{i,j}$ denote the start time, end time and processor mapping of the job. We assume all job times are *discrete*, any available core can be assigned to a task, and different instances of a task can be assigned to different cores.

A **run** of the task set, denoted $\langle \ldots, (i, j, s, e, p), \ldots \rangle$, is a timed sequence of jobs respecting the scheduling policy and processor work conservation. The task specification admits **non-determinism** via release jitter, sporadic tasks, termination time of jobs within the interval $[B, W]$, selecting one of many ready-to-run equal-priority jobs, and allocating one of many available cores to a job. As a result, multiple orderings of task instances are possible, leading to **many feasible runs** of the task set, and a deadline miss could occur along any of these. A run is **safe** if each job in the run meets its deadline. The task set is **schedulable** if every run of the task set is safe. Tasks spawn jobs up to a user-specified **analysis horizon** H or the hyperperiod H=LCM(periods) (or, H=2 $\times$ LCM(periods)+max(offset) when offsets are unequal). The goal is to encode all feasible task runs as solutions of a logical formula. Symbolic variables r, s, e, p are initially constrained as:

$$\begin{aligned} \forall (i,j) : jP_i + O_i \le r_{i,j} \le jP_i + O_i + J_i \\ \wedge r_{i,j} \le s_{i,j} \wedge B_i \le e_{i,j} - s_{i,j} \le W_i \wedge 1 \le p_{i,j} \le N \end{aligned} \tag{1}$$

A. Job precedence graph as a preprocessing step

Let $G = (V, E)$ be a graph, V is the set of jobs spawned up to time H. $(v, v') \in E$ is a directed edge from v to v' iff v is scheduled *no later than* v' in *all feasible* runs of the task set. Under NP-EDF, if a job has (strictly) earlier deadline compared with another and is released *no later than* the other (jitter included), then we can add an edge between the two:

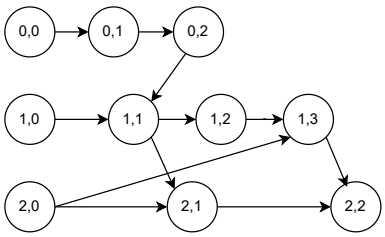

Fig. 3. Partial DAG for taskset of Table I

$$(v_{i,j}, v_{m,n}) \in E \text{ iff} : d_{i,j} < d_{m,n} \wedge jP_i + O_i + J_i \le nP_m + O_m \tag{2}$$

Fig. 3 illustrates a statically constructed partial DAG for the taskset of Table I. Edges of G ensure the destination job can be scheduled no earlier than the source, across all runs:

$$\forall (v_{i,j}, v_{m,n}) \in E : s_{i,j} \le s_{m,n} \wedge (p_{i,j} = p_{m,n} \Rightarrow e_{i,j} \le s_{m,n}) \tag{3}$$

These constraints significantly prune the sequences of jobs explored by the SMT solver. We define the **interference set** $I_{i,j}$ as the set of jobs that are *not connected* to $v_{i,j}$ in G:

$$v_{m,n} \in I_{i,j} \text{ iff} : \neg path(v_{i,j}, v_{m,n}) \ \wedge \ \neg path(v_{m,n}, v_{i,j}) \tag{4}$$

The interference set comprises of jobs that *could* contend with a given job when exploring runs of the task set.

B. Constraints for exact encoding of task runs

The s, e times of jobs are minimally constrained so far, thus we need to add more constraints to ensure correct runs.

Non-overlap: Jobs scheduled on same processor should not overlap in execution. The set of jobs that could precede a job (i, j) on the same processor is no larger than $I_{i,j} \cup A_{i,j}$, where $A_{i,j}$ is the set of ancestors whose deadline overlaps with $r_{i,j}$:

$$v_{p,q} \in A_{i,j} \text{ iff} : \ path(v_{p,q}, v_{i,j}) \wedge jP_i + O_i < d_{p,q} \tag{5}$$

These ancestors could be scheduled on different cores and thus can precede $v_{i,j}$. Thus, to prevent overlap, we construct:

$$\begin{aligned} \forall (i,j), \forall (m,n) \in I_{i,j} \cup A_{i,j} : (p_{i,j} = p_{m,n}) \Rightarrow \\ (e_{i,j} \le s_{m,n} \vee e_{m,n} \le s_{i,j}) \end{aligned} \tag{6}$$

Scheduling order: should be preserved globally (*i.e.,* across processors). Under NP-EDF, contention amongst jobs is resolved by selecting one with *earliest* deadline, leaving a non-deterministic choice when deadlines match:

$$\forall (i,j), \forall (m,n) \in I_{i,j} : s_{i,j} < s_{m,n} \Rightarrow \\ (d_{i,j} \leq d_{m,n} \vee s_{i,j} < r_{m,n}) \tag{7}$$

Work conservation: no processor core should idle in the presence of a ready job. We introduce a three-step encoding: **(a)** when a job waits after release, ensure *all* cores are busy, and schedule this job at the earliest available free core, **(b)** detect processor core idling, and **(c)** ensure jobs scheduled post-idling are also released post the idling.

Encoding step (a): For each job (i,j) having non-zero waiting time, ensure a job was scheduled *prior* to (i,j) and terminated *no earlier than* the time instant $s_{i,j}$, on *every* core, by enumerating over candidate jobs $I_{i,j} \cup A_{i,j}$:

$$\forall (i,j), (m,n) \in I_{i,j} \cup A_{i,j} : \; r_{i,j} < s_{i,j} \Rightarrow \\ \bigwedge_{q=1}^{q=N} (\bigvee (p_{m,n} = q \wedge s_{m,n} < s_{i,j} \wedge s_{i,j} \leq e_{m,n})) \tag{8}$$

Observe that Eqn. 8 ensures there is a job on *every* core, and in combination with the non-overlap constraint of Eqn. 6, job (i,j) was indeed scheduled on the earliest free core (the constraint is satisfied only for the minimum $e_{m,n}$). When multiple cores are free at the time instant $s_{i,j}$, a non-deterministic choice is available to the solver in assigning a value to $p_{i,j}$.

Encoding step (b): We use booleans $c_{i,j}$ to track if (i,j) was scheduled immediately post the preceding job:

$$\forall (i,j), \forall (m,n) \in I_{i,j} \cup A_{i,j} : c_{i,j} \Leftrightarrow \bigvee (e_{m,n} = s_{i,j}) \tag{9}$$

Thus, core $p_{i,j}$ idles *prior* to instant $s_{i,j}$ iff $c_{i,j}$ is FALSE.

Encoding step (c): The set of jobs that could be scheduled after the core idling interval *i.e.,* after the instant $s_{i,j}$, is jobs from $I_{i,j}$, and successors of (i,j) in the precedence graph G. These should be constrained to be released post idling:

$$\forall (i,j), \forall (m,n) \in I_{i,j} \cup \text{SUCC}_{i,j} : \\ (\neg c_{i,j} \wedge s_{i,j} < s_{m,n}) \Rightarrow s_{i,j} \leq r_{m,n} \tag{10}$$

Eqns. 8-10 form necessary and sufficient constraints to ensure that the idling interval of a processor core does not overlap with the waiting time of any job, achieving work conservation.

Next, we model **sporadic tasks** as follows. Let $\mathcal{T}_s$ be a finite set of sporadic tasks, $\tau \in \mathcal{T}_s$ is specified as $\tau : (i, P, B, W, O)$, where P is the minimum inter-arrival time between consecutive jobs, thus $r_{i,j} + P_i \leq r_{i,j+1}$. Since the instant of release lies in the interval $jP_i + O_i \leq r_{i,j} < \infty$, *at most* $\lceil (H - O_i)/P_i \rceil$ instances of τ_i can occur within horizon H. We track *valid* sporadic jobs with new boolean variables: $o_{i,j} \Leftrightarrow r_{i,j} < H$. Instances released after H are ignored. All constraints constructed above are fully applicable to sporadic jobs, but need to be qualified by the respective o variables. For example, deadline (assumed to be implicit) is computed as: $o_{i,j} \Rightarrow (d_{i,j} = r_{i,j} + P_i)$. Sporadic jobs increase the size of interference sets and cause significant blowup in the state space. We omit the full description of constraints for brevity.

Next, we compose the constraints in Eqns. 1,3,6-10 with those for the sporadic tasks to obtain an **exact** encoding of all (feasible) runs of the task set $\mathcal{T} = \mathcal{T}_p \cup \mathcal{T}_s$. This encoding, denoted ϕ_{sys}, can be analyzed by an off-the-shelf SMT solver.

*Theorem 1 (**Encoding**):* Every run of $\mathcal{T}$ exists as a solution of ϕ_{sys}. Conversely, every solution maps to a valid run.

Proof Sketch: The equivalence between runs of $\mathcal{T}$ and solutions of ϕ_{sys} can be established by mapping the constraints to scheduler actions and decisions, including the non-deterministic choices available to the scheduler, covering every run of $\mathcal{T}$. ■

Lastly, we encode timing violation *e.g.,* deadline miss as:

$$\phi_{err} \coloneqq \forall (i,j) : \; \bigvee (e_{i,j} > d_{i,j}) \tag{11}$$

If the solver returns a witness for $\phi_{sys} \wedge \phi_{err}$ *i.e.,* a satisfying assignment to the symbolic variables, we get a timing violation trace. $\mathcal{T}$ is **schedulable** if and only if $\phi_{sys} \wedge \phi_{err}$ is unsatisfiable.

IV. Repairing The Task Specification

When a task set violates timing requirement, it might be possible to revise a parameter such as offset or period, to meet a higher objective *e.g.,* control performance. However current schedulability tests provide little insight to system designers to improve the situation. Our approach repairs the task specification via formal synthesis of new (safe) values of task parameters by utilizing the encoding in Sec. III to discover and discard intervals of parameters leading to timing violation.

A. Refining the space of task offsets

We parse the witness reported by the solver to extract the task run $\sigma : \langle (i_1, j_1, s, e, p), \ldots, (i_n, j_n, s, e, p) \rangle$, and truncate to the last job that missed deadline: $e_{i_n,j_n} > d_{i_n,j_n}$. Suppose, we pick a task k, increment its offset O_k step by step, and observe the impact on σ. Our aim is to *retain* the (observed) timing violation while varying the offset, with other parameters *unchanged*. The offset "play" takes advantage of slack, if any, in σ, and the flexibility of reassigning the jitter and execution time of relevant jobs, to discover **intervals of values** yielding timing violation, enabling **debugging** of the task offset parameter.

We synthesize these intervals by identifying a *contiguous* neighbourhood [14], [17] of the current offset value that led to timing violation, generalizing earlier approaches to our broader setting of multicore task sets with mix of periodic and sporadic tasks. Incrementing the offset of a task proportionately *shifts* the instances of this task, along with their deadlines, within σ. The increment can continue as long as:

- there is a timing violation, such as $e_{i_n,j_n} > d_{i_n,j_n}$
- σ remains a feasible task run by "readjusting" job times
- the sequence of jobs in σ is unchanged

The last condition is a conservative requirement, as it may be possible to continue to obtain new error traces, however this cannot be guaranteed in general. Further, the original processor allocation of σ can be *relaxed*, as we are interested in *any* (feasible) processor allocation that continues to yield timing violation. When σ contains instances of sporadic tasks, their o variables must be set to TRUE during the increment. Together, these conditions and observations allow us to encode **interval**

synthesis as an optimization-modulo-theories query, where the goal is to pick a witness that **maximizes** the offset:

$$\overline{O}_k = \text{maximize } \mathbf{O}_k \text{ sub.to. } \phi_{sys} \wedge \phi_\sigma \wedge \phi_{spo} \wedge e_{i_n,j_n} > d_{i_n,j_n} \wedge \mathbf{O}_k \leq P_k \quad (12)$$

Here, $\overline{O}_k$ is the new (maximized) offset of τ_k with all other parameters unchanged; ϕ_{sys} is the set constraints from Eqns. 1-11; $\mathbf{O}_k$ is a new symbolic variable in ϕ_{sys} replacing the constant O_k; ϕ_σ encodes the sequence of jobs $s_{i_1,j_1} \leq ... \leq s_{i_n,j_n}$ from σ; ϕ_{spo} sets o variables to TRUE for sporadic instances in σ; and we assume P_k as the upper bound on the maximization.

*Lemma 1 (**Completeness**):* By construction of Eqn. 12, each choice of offset within $[O_k, \overline{O}_k]$ leads to timing violation.

These per-task intervals can be composed to form an n-dimension box $[O_1, \overline{O}_1] \times \cdots \times [O_n, \overline{O}_n]$ in the space of task offsets. The composition preserves the timing violation, thus the box can be marked for **rejection** in the offset space.

*Theorem 2 (**Composition**):* Assigning any value to task offsets from the n-D box leads to timing violation.

Proof Sketch: Intuitively, maximization preserves the job sequence and deadline miss in σ, and selecting (new) offset values from within the box further shifts release and start times of jobs, yet respects their **relative** order, thereby maintaining the sequence and timing violation. For simplicity, consider a 2-D box $[O_1, \overline{O}_1] \times [O_2, \overline{O}_2]$ based on σ. Suppose we pick a point $[O'_1, O_2], O_1 \leq O'_1 \leq \overline{O}_1$ in this box and construct a new precedence graph G_1. Observe that:

- G_1 **preserves** all edges of G via Eqns. 3,7 (part of ϕ_{sys})
- Incoming edges to τ_1 vertices **can** get added (via Eqn. 2)

Edges impose fixed ordering of jobs, thus G_1 admits a subset of runs of G, while **retaining** σ (via Lem. 1). Now, pick another point $[O'_1, O'_2], O_2 \leq O'_2 \leq \overline{O}_2$, and construct a new G_2, then:

- Incoming edges to τ_2 vertices can get added to G_2 wrt G_1
- Outgoing edges from τ_2 vertices can get removed, wrt G_1

Removing edges relaxes fixed ordering between jobs, expanding the existing the set of runs, while adding edges prunes runs. Consider three arbitrary jobs $(x, y), (2, z), (v, w)$, in σ, with $s_{x,y} < s_{2,z} < s_{v,w}$. Then, σ can be **excluded** from the runs of G_2 by blocking the sequence under two cases: either the edge $(2, z) \rightarrow (x, y)$, or $(v, w) \rightarrow (2, z)$, gets admitted in G_2. In the first case, outgoing edges from τ_2 vertices can be removed but can't be added (since O'_2 only delays these jobs). In the second case, since $s_{2,z} < s_{v,w}$ was witnessed in σ, O'_2 is within safe maximization bounds (via Lem. 1) to retain this order. Moreover, setting O_v to O'_v only further delays $s_{v,w}$, guaranteeing the order. Together, this implies σ can't be blocked in G_2. Thus, while the sets of runs admitted by G, G_1 and G_2 differ, the timing violation is preserved in the box. ■

B. Refining task periods

Following the offset refinement approach above, we can discover intervals of task periods that preserve the timing violation. We reuse Eqn. 12 to maximize over the task periods:

$$\overline{P}_k = \text{maximize } \mathbf{P}_k \text{ sub.to. } \phi_{sys} \wedge \phi_\sigma \wedge \phi_{spo} \wedge e_{i_n,j_n} > d_{i_n,j_n} \wedge \mathbf{P}_k \leq 2 \times P_k \quad (13)$$

where, $\overline{P}_k$ is the new (maximized) period of task k, $\mathbf{P}_k$ is a new symbolic variable introduced in ϕ_{sys} that replaces the constant P_k with other parameters unchanged, and $2 \times P_k$ as the upper bound on the maximization (assuming control task, selecting period any more will degrade control performance). Computationally, modifying the period parameter shifts the j^{th} job in σ by an order of j, compared to modifying the offset that shifts each job equally.

Lem. 1 applies to the case of task periods. Intervals obtained in this way can be composed as $[P_1, \overline{P}_1] \times \cdots \times [P_n, \overline{P}_n]$, to obtain an n-D box preserving the timing violation, similar to Thm. 2. Soundness of n-D box of intervals of periods again builds on Eqn. 7. Informally, with P_1 set to $\overline{P}_1$ and the scheduling constraint of Eqn. 7 holding (via Eqn. 13), jobs of τ_1 experience a shift (including deadline), yet σ is preserved. Then, increasing P_m (up to $\overline{P}_m$) only *relaxes* the consequent terms (namely, $d_{m,n}$ and $r_{m,n}$) in Eqn. 7, preserving it's satisfaction. Thus, while task periods change by selecting new values from the box, the job sequence and deadline miss of σ is reproducible. Consequently, any choice of periods from the n-D box will lead to timing violation.

V. Experiments

We implemented a Python-based tool STAR, that accepts as input a task specification, choice of parameter to refine, and H (optional). STAR refines the parameter space by iteratively discovering boxes (using an optimizing SMT solver) and synthesizing new parameter values (SMT query), till the schedulability check (SMT query) reports *unsat*, indicating timing safety, or the parameter synthesis fails (solver returns *unsat*) indicating the boxes cover the entire parameter space *i.e.,* unschedulability. Experiments were run on a laptop with i7/16GB. We used Z3-3.4.6 [31], Yices-2.6.2 [32], and OptiMathSAT-1.6.3 [33], with Z3/Yices for checking schedulability, and Z3/OptiMathSAT for optimization queries. We report best-of-two times.

A. Experimental evaluation against UTOR

TABLE II
OFFSET REFINEMENT: STAR VS. UTOR

Task Set	Tasks	Refinements	Result	STAR	UTOR
R1-R7	≤ 7	≤ 6	Sched	$\approx$ 5s	$\approx$ 120s
U1, U2	3	≤ 3	Unsched	$\approx$ 1s	$\approx$ 120s

We compare STAR against a model checking approach [17], called UTOR (Unicore Task Offset Refinement), that supports offset refinement for periodic tasks with jitter and execution time variation similar to STAR's supported task model, in Table II. UTOR analyzed 9 (synthetic) task sets containing maximum 7 tasks. Of these 9 task sets, 7 could be successfully repaired (R1-R7); 2 task sets (U1, U2) are unschedulable (parameter space exhausted). While both tools successfully analyzed the task sets, STAR outperformed UTOR by a significant margin:

- UTOR uses a C array representation to encode task runs, leading to state space explosion for the model checker.
- STAR uses a DAG to encode static ordering and interfering jobs, pruning a large number of invalid runs.

B. Evaluation of offset repair on PapaBench

We evaluated STAR on PapaBench [34], a real-world task set having 8 periodic and 4 interrupt tasks. UTOR's exact analysis using model checking tends to perform exponentially worse with increase in state space. Therefore, to systematically study the scalability of our SMT encoding combined with a DAG, we made 6 variants, P1 to P6 in Table III, of the PapaBench task set by (i)removing the interrupt tasks from the set (as UTOR does not support sporadic tasks) and some tasks with small execution time *e.g link_fbw_send* (for tractable analysis), (ii)we ceiled all the WCET values to the nearest millisecond, and (iii)assigned different values to offsets, jitter, and execution budget, such that the non-determinism increases in magnitude from P1 to P6. Further, all these six variants have at least one choice of parameter values that makes them schedulable.

For example, for task set P1, we set BCET = WCET for each task, and jitter to 0. This causes the jobs in P1 to execute in the same fixed order, except those with equal priority. For task set P2, we set BCET $=$ WCET$/2$. This causes each job's completion time to vary, potentially affecting the execution order of subsequent jobs, leading to a significant rise in the number of possible job sequences. Therefore, the solver explores non-deterministic job termination within the interval [WCET/2, WCET]. The state space begins to grow exponentially with each such relaxation. Similarly, non-zero jitter (set to maximum 5% of task period) causes the release time of each job to vary in the interval [0,J], causing state space explosion.

TABLE III
EVALUATION ON PAPABENCH

Taskset	Tasks	Refs	STAR
P1	7	2	3.5s
P2	7	2	4.9s
P3	7	11	20.7s
P4	7	2	7.7s
P5	7	26	1m
P6	7	6	18.6s

UTOR timed out on *all* the task sets P1 to P6 (and hence not shown in the table) with a 30min timeout. In contrast, STAR could repair all task sets. Notice that, even though P5's state space is "less" than that of P6, STAR took more time for P5. This is due to larger number of refinements (26) required for finding a safe assignment to offsets of P5, compared to lesser refinements (6) in case of P6. STAR's analysis is sensitive to the state space, the specific error traces reported by the solver (affecting box size), and the number of refinements required.

C. Repairing multicore mixed task sets

We chose tasks from [34] for repairing realistic tasksets and tweaked their WCET and periods such that the tasksets are not schedulable initially, but can be repaired by revising offsets or periods. All task times were set at microsecond precision, two interrupt tasks were modeled as sporadic, jitter was varied up to 5% of the task's period and BCET from 50%–100% of WCET. Table IV reports the experimental evaluation for $N = 2$, 4, and 6-core. Timeout for STAR was set to 30 mins. Column T is the task set name, R#T is the number of tasks whose parameter – offset or period – was revised, #R is the no. of refinements required to discover a schedulable parameter choice, and Time is the end-to-end tool time in seconds. #R also indicates number of boxes (or regions) discovered by STAR.

TABLE IV
EVALUATION ON MULTICORE MIXED TASK SETS

Task Specification			Offset			Period		
T	#Proc	#Tasks	R#T	#R	Time	R#T	#R	Time
M1	2	14	13	1	7.69	3	1	47.73
M2	2	14	13	1	9.82	3	1	22.08
M3	2	14	11	1	19.02	2	1	398.6
M4	2	14	12	1	17.27	8	1	190.5
M5	4	10	8	4	233.47	5	1	14.05
M6	4	14	12	3	179.05	3	1	187.23
M7	6	12	10	1	34.3	4	1	45.1
M8	6	14	12	2	408.7	4	1	400.9

In the **2-core** case, rows M1–M4, STAR could successfully repair the specification in one refinement, but with increasing time. This is due to the increasing non-determinism associated with the tasks. STAR revised offsets of 11–13 tasks out of the 14 tasks, and periods of 2-8 tasks. However, period refinement takes longer time: when task periods are revised, the time horizon $H = LCM$ of tasks was found to increase, leading to larger number of jobs spawned, which in turn leads to exponentially more number of job sequences to be analyzed. Further, the revised task periods were about 20% to 50% more than original periods.

In the **4-core** case, rows M5–M6, overall repair time was higher for offset refinement due to larger number of refinements; the maximization query gets significantly costlier with more non-determinism in the system.

In the **6-core** case, row M7–M8, the impact of increasing H on period refinement is more pronounced. Further, refining offsets is also impacted due to larger number of cores contributing to increase in state space.

Summary: STAR is able to successfully repair up to 14 tasks (12 periodic, 2 sporadic) scheduled on up to 6 cores within few refinements, under 7mins, suiting offline repair, particularly for safety-critical task sets. Each refinement step involves a schedulability check, an optimization query for each parameter, and an SMT-based synthesis of new values. *Exact* multicore schedulability analysis remains hard: *e.g.,* [29] reports scalability up to 3 sporadic tasks on 2 cores and up to 15 periodic tasks (no sporadic tasks) on 8 cores, and [30] report issues with both time taken and space consumed for up to 7 sporadic tasks on up to 3 cores. STAR found it difficult to scale beyond 6 cores and 4 sporadic tasks.

VI. CONCLUSIONS AND FUTURE WORK

We have proposed an *exact timing debugging and repair* approach that synthesizes safe values for task parameters, using an SMT encoding combined with a DAG to capture static scheduling precedence. Our prototype tool implementation successfully debugged and repaired tasksets with up to 14 tasks scheduled on 6 cores. The tool can detect and discard precise ranges of task parameters leading to timing violation. In the future, we would like to address (i) repair with the goal of minimizing changes to the task specification, (ii) learning-based approaches to guess the best repair candidate parameter, and (iii) an abstraction-refinement approach [35] for scalability.

REFERENCES

[1] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 01, jan 2019.

[2] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, nov 2019.

[3] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A survey of learning-based automated program repair," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 2, dec 2023.

[4] Z. Sun, M. Guo, and X. Liu, "A survey of real-time scheduling on multiprocessor systems," in *39th National Conference of Theoretical Computer Science NCTCS*, ser. Communications in Computer and Information Science, 2021.

[5] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. I. Davis, "A comprehensive survey of industry practice in real-time systems," *Real Time Systems*, vol. 58, no. 3, 2022.

[6] T. P. Baker, "Stack-based scheduling for realtime processes," *Real-Time Systems*, vol. 3, no. 1, 1991.

[7] G. C. Buttazzo, M. Bertogna, and G. Yao, "Limited preemptive scheduling for real-time systems. a survey," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, 2013.

[8] C. Lee, J. Hahn, Y. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. Kim, "Analysis of cache-related preemption delay in fixed-priority preemtive scheduling," *IEEE Transactions on Computers*, vol. 47, no. 6, 1998.

[9] K. Jeffay, D. Stanat, and C. Martel, "On non-preemptive scheduling of period and sporadic tasks," in *[1991] Proceedings Twelfth Real-Time Systems Symposium*, 1991.

[10] H. Lee and J. Lee, "Limited non-preemptive edf scheduling for a real-time system with symmetry multiprocessors," *Symmetry*, vol. 12, no. 1, 2020.

[11] M. Nasri and G. Fohler, "Non-work-conserving non-preemptive scheduling: Motivations, challenges, and potential solutions," in *28th Euromicro Conference on Real-Time Systems (ECRTS)*, 2016.

[12] M. Nasri and G. . Fohler, "Non-work-conserving scheduling of non-preemptive hard real-time tasks based on fixed priorities," in *Proceedings of the 23rd International Conference on Real Time and Networks Systems (RTNS)*, 2015.

[13] D. Seto, J. P. Lehoczky, and L. Sha, "Task period selection and schedulability in real-time systems," in *19th IEEE Real-Time Systems Symposium (RTSS)*, 1998.

[14] E. Bini, M. Di Natale, and G. Buttazzo, "Sensitivity analysis for fixed-priority real-time systems," in *18th Euromicro Conference on Real-Time Systems (ECRTS)*, 2006.

[15] D. Roy, C. Hobbs, J. H. Anderson, M. Caccamo, and S. Chakraborty, "Timing debugging for cyber-physical systems," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021.

[16] M. Nasri, "On flexible and robust parameter assignment for periodic real-time components," *SIGBED Reviews*, vol. 14, no. 3, 2017.

[17] A. Yeolekar, R. Metta, R. Venkatesh, and S. Chakraborty, "Refining task specifications using model checking," in *IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2018.

[18] J. Bendík, A. Sencan, E. A. Gol, and I. Černá, "Timed automata relaxation for reachability," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2021.

[19] R. Noguchi, O. Sankur, T. Jéron, N. Markey, and D. Mentré, "Repairing real-time requirements," in *Automated Technology for Verification and Analysis*, A. Bouajjani, L. Holík, and Z. Wu, Eds., 2022.

[20] A. Cimatti, L. Palopoli, and Y. Ramadian, "Symbolic computation of schedulability regions using parametric timed automata," in *2008 Real-Time Systems Symposium*, 2008, pp. 80–89.

[21] F. Pozo, G. Rodriguez-Navas, and H. Hansson, "Schedule reparability: Enhancing time-triggered network recovery upon link failures," in *International Conference on Embedded and Real-Time Computing Systems and Applications*, 2018.

[22] N. Guan, W. Yi, Z. Gu, Q. Deng, and G. Yu, "New schedulability test conditions for non-preemptive scheduling on multiprocessor platforms," in *2008 Real-Time Systems Symposium*, 2008, pp. 137–146.

[23] Z. G. Gu, Z. Wang, H. Chen, and H. Cai, "A model-checking approach to schedulability analysis of global multiprocessor scheduling with fixed offsets," *International Journal of Embedded Systems (IJES)*, vol. 6, no. 2-3, 2014.

[24] E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, "Schedulability analysis of fixed-priority systems using timed automata," *Theoretical Computer Science (TCS)*, vol. 354, no. 2, 2006.

[25] N. Guan, Z. Gu, M. Lv, Q. Deng, and G. Yu, "Schedulability analysis of global fixed-priority or edf multiprocessor scheduling with symbolic model-checking," in *Proc. of 11th International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, 2008.

[26] A. de Matos Pedro, D. Pereira, L. M. Pinho, and J. S. Pinto, "Smt-based schedulability analysis using rmtl," *SIGBED Reviews*, vol. 14, no. 3, 2017.

[27] M. Nasri and B. B. Brandenburg, "An exact and sustainable analysis of non-preemptive scheduling," in *Real-Time Systems Symposium (RTSS)*, 2017.

[28] M. Nasri, G. Nelissen, and B. B. Brandenburg, "A response-time analysis for non-preemptive job sets under global scheduling," in *30th Euromicro Conference on Real-Time Systems (ECRTS)*, vol. 106, 2018.

[29] B. Yalcinkaya, M. Nasri, and B. B. Brandenburg, "An exact schedulability test for non-preemptive self-suspending real-time tasks," in *Design, Automation & Test in Europe Conference & Exhibition, (DATE)*, 2019.

[30] A. Burmyakov, E. Bini, and E. Tovar, "An exact schedulability test for global FP using state space pruning," in *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, 2015.

[31] N. Bjørner, A.-D. Phan, and L. Fleckenstein, "νz - an optimizing smt solver," in *21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2015, pp. 194–199.

[32] B. Dutertre, "Yices 2.2," in *Computer-Aided Verification (CAV)*, July 2014.

[33] R. Sebastiani and P. Trentin, "Optimathsat: A tool for optimization modulo theories," *Journal of Automated Reasoning*, 2018.

[34] W. Lunniss, S. Altmeyer, and R. Davis, "A comparison between fixed priority and edf scheduling accounting for cache related pre-emption delays," *Leibniz Transactions on Embedded Systems*, vol. 1, no. 1, 2014.

[35] A. Yeolekar, R. Metta, C. Hobbs, and S. Chakraborty, "Checking scheduling-induced violations of control safety properties," in *20th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, 2022.